Nancy-Université
Université Henri Poincaré

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

Toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

➢ Contact SCD Nancy 1 : theses.sciences@scd.uhp-nancy.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université

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

Réalisation de surface : ambiguïté et déterminisme

Surface realisation: ambiguity and determinism

THÈSE

version 1.0.1

présentée et soutenue publiquement le 14 novembre 2007

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Eric Kow

Composition du jury

Rapporteurs :	John Carroll	Professeur, Université de Sussex, Brighton
	Patrick Saint-Dizier	Directeur de Recherche CNRS, IRIT Toulouse
Examinateurs :	Dominique Méry	Professeur, Université Henri Poincaré, LORIA Nancy
	Eric De La Clergerie	Chargé de Recherche, INRIA Rocquencourt
	Claire Gardent	Directrice de Recherche CNRS, LORIA Nancy

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

Loria

Remerciements

Let's not get sentimental here. This thesis is the result of six years hanging out in Nancy, with the good people of LORIA. I want you to meet the people that made it happen.

Claire Gardent



Four years working with Claire and I still don't how she does it: how she gets right to the heart of the matter, or how she keeps everything so simple. I just hope some of it has rubbed off, and that her time coaxing this computer geek out into the research world will have been worth her while. Thanks for everything, Claire.

Patrick Blackburn



Patrick gave this thesis a heaping dose of extra polish. He is the man that welcomed me at the train station when I first arrived in Nancy, and he is one of the people that encouraged me to stick around, when I was still an *ingénieur* trying to figure out what to do with myself. Patrick is a fun person to watch and to learn from, particularly because of the freedom he enjoys from dogma or preconceived notions. And thanks, Patrick, for reminding me to stay alive.

Hélène Manuelian



Without Hélène, the French summary of this thesis would not have been possible. Trust me, you don't learn to write like that by watching *Loft Story*. This an big help on the administrative front, but to be honest, my real reason to say thanks is for being somebody to look up to. Hélène has taught me a lot about working with people. She is my favourite example of thoughtful consideration. What I admire particularly is that the consideration does not just come from some knee-jerk niceness reflex, but from an acute and mindful awareness of her surroundings. "Make it easy for others." I'll try.

Thanks also to...

Laurent Romary for bringing me to Nancy and growing me out of my first phases of stupid.

Joseph Le Roux for help on polarity filtering, enlightening discussions and just being a good example.

Bertrand Gaiffe for patiently teaching me about chart parsing and hashing ideas out with me.

Carlos Areces for giving me a model of clear and unpretentious writing to aspire to.

B224 for good chats, times and fights; the things that friends are made of.



Benoît Crabbé and Djamé Seddah for many nuggets of advice and general wisdom from the era of Langue et Dialogue *thésards*.



The Proofreading Brigade for catching the many clumsinessessesses of my writing. Sébastien Hinderer, Ania Kupsc, Jackie Lai, Michael Leiseca and Yannick Parmentier, I salute you!

The inevitable forgotten for the many ways they have helped me. Sorry, and thanks.

That's all

So, merci, merci de tout coeur.



Eric
2007-11-27

Contents

Remerciements (Acknowledgements) **i**

Contents **iii**

Ambiguïté et détérminisme **iv**

Introduction **2**

I Background **5**

1 Realisation algorithms **7**
1.1 Syntactic tree traversal 8
1.2 Search 16
1.3 Sharing intermediate results 17
1.4 Summary of the main issues 38

2 Flat semantics with holes **41**
2.1 Flat semantics 41
2.2 Logical-form equivalence 43
2.3 The case for a flat semantics 49
2.4 Intersective modifiers 51
2.5 Summary of flat semantics 57

3 Tree Adjoining Grammar **59**
3.1 From TAG to FB-LTAG 59
3.2 TAG Derivations 64
3.3 FB-LTAG augmented with L_U flat semantics 65
3.4 Generation with TAG 73

4 GenI and SemFraG **77**
4.1 GenI 77
4.2 SemFraG 84
4.3 Related NLG systems for TAG 86

II Contributions **89**

5 Polarity filtering **91**
5.1 Polarised intuitions 91

5.2 Building polarity automata 96
5.3 Chart generation with polarity automata 108
5.4 Extensions 108
5.5 Evaluation 118
5.6 Related work in lexical disambiguation 121

6 Paraphrase selection **127**
6.1 Contextual appropriateness 128
6.2 Selection mechanism 129
6.3 Evaluation 132
6.4 Possible extensions 135
6.5 Related work in paraphrase selection 138

7 Reducing overgeneration **147**
7.1 Overgeneration 147
7.2 Grammar debugging 148
7.3 An incremental approach 149
7.4 Evaluation and results 155
7.5 Possible extensions 157
7.6 Related work 159

8 Conclusion **165**
8.1 Summary 165
8.2 Future work 166
8.3 Putting GenI to work 167

A SemFraG families **172**

B Tree properties from SemFRaG **175**

C Deductive realisation and unification **181**
C.1 Kay1996 with unification 181
C.2 GenI with unification 182

D GenI pseudocode **183**
D.1 Lexical selection 183
D.2 Realisation proper 184
D.3 Helper functions 185

Bibliography **187**

Ambiguïté et déterminisme

This chapter presents a summary of the thesis, in French.
Ce chapitre présente un résumé français de la thèse.

La génération de langue naturelle (GLN) consiste à traduire un but communicatif de nature abstraite en langue naturelle. Le module de réalisation de surface est une petite partie du générateur de langue naturelle et sa tâche est relativement clairement définie : étant donné une grammaire et une représentation du sens (la plupart du temps une forme logique), il doit produire les chaînes que la grammaire associe à la sémantique. La réalisation de surface est une des tâches les plus concrètes de la génération et donc une des plus faciles. C'est d'ailleurs la raison pour laquelle la plus grande partie de la recherche en génération de textes porte sur la réalisation de surface et qu'il existe un certain nombre de réalisateurs de surface de bonne qualité et réutilisables comme RealPro, fuf et kpml. Ces réalisateurs ont tous été intégrés pour la construction de systèmes de génération de langue naturelle.

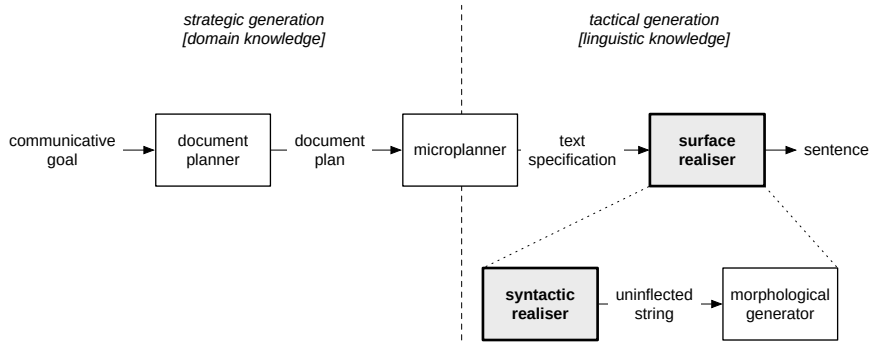
Bien que la réalisation de surface soit facile, elle ne peut pas être considérée comme un problème totalement résolu. En particulier, le traitement de la paraphrase pose encore des problèmes. Nous entendons par paraphrase le fait qu'il existe souvent plus d'une façon d'exprimer la même chose. Cette caractéristique des langues a pour conséquence la très grande variété d'énoncés possibles, et constitue la raison pour laquelle nous sommes capables d'exprimer des nuances de sens subtiles. Malheureusement, c'est aussi un cauchemar combinatoire. La thèse qui est résumée ici traite de la façon dont un module de réalisation de surface doit gérer la paraphrase, que nous appellerons abusivement ambiguïté pour conserver le parallèle avec l'analyse (le parsing).

La thèse, comme le résumé que nous en faisons ici, se structure de façon standard autour de deux grandes parties. Nous présentons tout d'abord l'état de l'art (Chapitres 1–4) puis notre contribution au domaine (Chapitres 5–7). Cette deuxième partie s'articule autour de trois thèmes qui sont les suivants :

1. L'utilisation de techniques de « filtrage par polarité » pour réduire l'espace de recherche du module de réalisation.
2. Un mécanisme de sélection de la paraphrase, permettant au réalisateur de renvoyer le meilleur résultat par rapport aux critères (descriptions linguistiques) donnés par l'utilisateur.
3. Un processus semi-automatisé de débuggage de la grammaire, utilisant les mêmes descriptions linguistiques pour localiser les causes de surgénération dans la grammaire.

F-1 Algorithmes de réalisation de surface

La génération de langue naturelle est généralement vue comme un enchaînement de tâches (on parle traditionnellement de structure en pipeline) tel que l'illustre la figure ci-dessous. Ces tâches sont regroupées dans deux modules distincts : le premier est le module stratégique, qui détermine « quoi dire ? » sur la base de connaissances du domaine, le deuxième est le module tactique, répondant à « comment le dire? ». Ce dernier fonctionne grâce aux connaissances linguistiques. En réalité, on doit faire des distinctions plus fines. A la fin des années 90, il est devenu clair que certaines tâches devaient utiliser à la fois les connaissances du domaine (donc extralinguistiques) et les connaissances linguistiques. On a alors dû utiliser un composant intermédiaire, qu'on a appelé module de microplanification. Dans notre thèse, nous allons postuler que la réalisation de surface a lieu à la fin du processus (comme c'est la plupart du temps le cas), et qu'elle utilise en entrée la sortie du module de microplanification.



Les différents algorithmes de réalisation de surface peuvent se différencier sur trois aspects : le parcours de l'arbre syntaxique, l'exploration des espaces de recherche, et le stockage des résultats intermédiaires. Ce sont ces différents aspects de la réalisation que nous présentons maintenant.

F-1.1 Parcours de l'arbre

On peut considérer la réalisation de surface comme un processus de découverte d'un arbre syntaxique correspondant à une sémantique d'entrée [Shieber *et al.*, 1990]. Ce processus peut être abordé par une stratégie descendante, ascendante ou mixte. Chacune de ces stratégies pose des problèmes pratiques spécifiques. L'utilisation d'algorithmes descendants nous expose à la récursivité à gauche ; les algorithmes ascendants demandent trop de restrictions du formalisme grammatical pour être complets et sont trop non-déterministes pour être utiles en pratique. Les stratégies mixtes comme la génération dirigée par la tête sémantique (semantic head driven generation ou SHDGA) [Shieber *et al.*, 1990] sont de loin les meilleures, et c'est pourquoi nous allons nous centrer sur cette façon de faire.

SHDGA peut être vu comme une adaptation du parsing coin gauche (left-corner parsing) dans laquelle au lieu de chercher le coin gauche de la phrase, on cherche sa tête sémantique. La tête sémantique d'une règle est le nœud fils qui a la même sémantique que le nœud père. Précisions toutefois que toutes

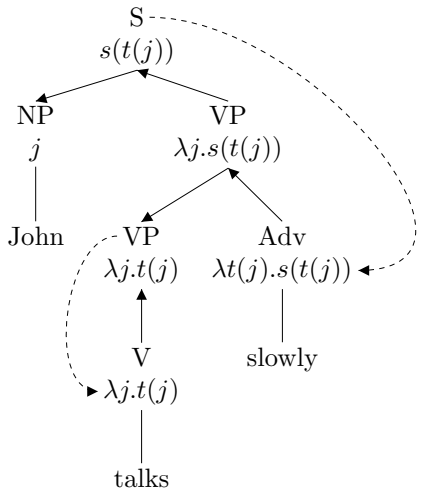


FIG. 1: parcours SHDGA

les règles n'ont pas forcément une tête sémantique. Certaines règles ont même parfois plusieurs nœuds fils qui partagent la sémantique du nœud père, mais nous allons mettre ces cas de côté pour l'instant. Dans SHDGA, la grammaire est pré-traitée et divisée entre les règles enchaînées (celles qui ont une tête sémantique) et les règles non-enchaînées (celles qui n'en ont pas).

Le traitement commence au symbole de départ de la grammaire. Ensuite, il trouve un nœud pivot et traite récursivement ses fils. Le pivot est le nœud père d'une règle non enchaînée dont la partie gauche correspond au but courant. La sélection du pivot remplit le même rôle que la phase de scan dans un analyseur coin-gauche : elle décide de l'endroit à partir duquel on commence à remonter l'arbre d'analyse. Ce processus de remontée (ou de connexion) est lui aussi récursif : on sélectionne une règle enchaînée, on unifie sa tête sémantique avec le nœud courant, on traite ses autres fils et ensuite on remonte encore jusqu'à une autre règle enchaînée ; on ne s'arrête que lorsque le but et le nœud courant se correspondent. Ce parcours est illustré dans la figure 1, qui montre la réalisation de *slowly(talk(john))* par la grammaire ci-dessous :

c1.	S(S)	$\rightarrow$ NP(X) VP($\lambda X.S$)
c2.	VP($\lambda X.S$)	$\rightarrow$ V($\lambda X.S$)
c3.	VP($\lambda X.S$)	$\rightarrow$ VP($\lambda X.V$) Adv($\lambda V.S$)
c4.	V($\lambda X.talk(X)$)	$\rightarrow$ talks
c5.	NP($john$)	$\rightarrow$ John
c6.	Adv($\lambda V.slowly(V)$)	$\rightarrow$ slowly

F-1.2 Recherche

En un sens, le choix d'une bonne stratégie de parcours de l'arbre aide les algorithmes de réalisation de surface à éviter les choix non déterministes. Par

exemple, les approches dirigées par les têtes de la réalisation sont avantageuses parce qu'elles manipulent seulement des nœuds avec une sémantique complètement instanciée (et évitent ainsi le non-déterminisme qui peut résulter des différentes tentatives d'instanciation de la sémantique). La langue naturelle étant intrinsèquement ambiguë, il est clair que quel que soit le parcours d'arbre qu'on adopte, il y aura toujours à gérer une part de non-déterminisme. C'est là que le choix de la stratégie de recherche devient important. Pour pouvoir faire ce choix, nous devons répondre aux trois questions suivantes :

1. Devrions-nous retourner une seule solution, n solutions, ou toutes les solutions ? Si nous les voulons toutes, comment pourrions-nous les trier ?
2. Comment l'arbre de recherche (qu'on ne doit pas confondre avec l'arbre syntaxique) doit-il être exploré ? Parmi les stratégies d'exploration de l'arbre de recherche, on trouve les recherches dites « en largeur d'abord », en « profondeur d'abord », ou « gourmandes ».
3. Les choix que nous faisons sont-ils définitifs ? Un choix extrême serait de ne pas prendre d'engagement, quel qu'il soit, et d'autoriser tout l'espace de recherche à être exploré. L'autre attitude extrême serait de se tenir coûte que coûte aux choix qu'on a faits. Les deux solutions représentent en fait différents degrés d'élagage.

F-1.3 Stockage des résultats intermédiaires

Jusqu'à maintenant, nous avons parlé d'éviter le non-déterminisme (dans son aspect de parcours de l'arbre) et de faire les meilleurs choix quand il survient (dans son aspect recherche). Le troisième aspect de la réalisation consiste à trouver un moyen de se relever des inévitables mauvais choix que nous aurons faits. Un tel mécanisme est le retour arrière (backtracking). Si notre stratégie de recherche rencontre une impasse, on devrait simplement pouvoir revenir sur nos pas jusqu'au choix le plus récent et essayer un autre choix. Le retour arrière peut pourtant s'avérer extrêmement inefficace ; il peut inutilement recalculer des choix déjà éliminés. Une alternative courante au backtracking est la méthode de programmation dynamique appelée analyse tabulaire ou « chart parsing ». Les analyseurs qui utilisent cette méthode stockent des résultats intermédiaires dans une structure de données et les utilisent pour construire d'autre résultats intermédiaires qui sont à leur tour stockés et utilisés pour construire des résultats toujours plus complets.

La génération tabulaire (ou « chart generation ») est une adaptation de cette technique à la réalisation de surface. Une question qui se pose en génération tabulaire est de trouver le mécanisme adapté pour indexer les résultats intermédiaires. L'analyse tabulaire utilise des indices de position dans la chaîne, mais cela n'est pas possible en génération parce que l'entrée n'est pas une chaîne. Une approche très citée de ce problème a été introduite par [Kay, 1996], et a été incorporée dans beaucoup de générateurs tabulaires depuis [Carroll *et al.*, 1999; Striegnitz, 2000; White, 2004]. L'approche de départ s'appuie sur l'utilisation d'une sémantique plate. Une formule de sémantique plate est un ensemble de littéraux où chacun consiste en un prédicat et des indices. La formule qui suit, par exemple, est une formule de sémantique plate.

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

Il existe plusieurs formalismes de sémantique plate, mais ils ont au moins cela en commun. L'idée centrale de la méthode est que les indices sémantiques comme r ou j peuvent aussi servir d'indices pour la génération tabulaire dans la mesure où les arcs actifs et inactifs doivent se combiner seulement autour d'un indice sémantique commun. C'est cette approche que nous utilisons dans la thèse.

F-1.4 Synthèse

Nous pouvons donc récapituler maintenant les trois aspects sur lesquels les algorithmes de réalisation de surface peuvent se différencier, ainsi que les choix que nous ferons dans notre travail :

1. La façon de parcourir l'arbre syntaxique : pour notre part, nous utiliserons une méthode dirigée par les têtes.
2. La stratégie de recherche : nous choisissons de retourner tous les résultats, et nous nous contentons de techniques d'élagage sûres.
3. Le stockage de résultats intermédiaires : nous utiliserons une génération tabulaire combinée à une sémantique plate.

F-2 Sémantique plates à trous

Les langages de représentation sémantiques doivent gérer la récursivité d'une manière ou d'une autre. En effet, les expressions en langue naturelle peuvent être d'une longueur indéterminée et une expression peut être construite à partir d'une autre expression. Plus important encore, le sens d'une expression peut être construit à partir du sens des autres expressions, et ce, à une profondeur quelconque. Par exemple, considérons la phrase "A dog barks." En logique du premier ordre, on peut la représenter ainsi : $\exists d(dog(d) \wedge bark(d))$; comme nous le montrons dans l'exemple ci-dessous, cette phrase et sa sémantique peuvent être imbriquées indéfiniment.

1. A dog barks.
   $\exists d(dog(d) \wedge bark(d))$
2. A man complains that a dog barks.
   $\exists m(man(m) \wedge complain(m, \exists d(dog(d) \wedge bark(d))))$
3. A neighbour says that a man complains that a dog barks.
   $\exists n(neighbour(n) \wedge say(n, \exists m(man(m) \wedge complain(m, \exists d(dog(d) \wedge bark(d))))))$
4. ...

Les représentations en sémantique plate sont elles aussi récursives, bien qu'elles expriment la récursivité de manière moins directe. Au lieu d'inclure les autres formules sémantiques, la sémantique plate y *réfère*. N'importe quel langage de représentation sémantique peut être aplati. Nous montrons ci-dessous une formule récursive exprimée en logique du premier ordre et dans une représentation hypothétique en sémantique plate :

(1) Ernest considers buying a dog.
   $\exists d(dog(d) \wedge consider(ern, buy(ern, d)))$
   l1:dog(d), l2:buy(ern,d), l3:consider(ern,l2), l4:and(l2, l3), l5:exists(d,l4)

F-2.1 L_U : une application de la sémantique à trous

Le langage de représentation sémantique utilisé dans cette thèse est L_U [Gardent and Kallmeyer, 2003]. Il s'agit d'une reformulation de la Logique des Prédicats « débranchée » (Predicate Logic Unplugged, désormais PLU) où les variables lambda sont remplacées par des variables d'unification. La PLU est une application de la sémantique à trous [Bos, 1995]. Il s'agit d'un système permettant d'aplatir les langages de représentation sémantique et d'y introduire une sous-spécification (ainsi, Bos montre comment on peut « débrancher » la logique du premier ordre et la DRT). Le langage L_U simplifie quelque peu la PLU et ajoute des variables d'unification pour permettre la construction sémantique :

Définition 1 (formule syntaxique L_U)**.** Soit (i) I_{var} un ensemble de variables d'unification et I_{con} un ensemble de constantes ; (ii) L_{var} un ensemble de variables d'unification « étiquettes » (label unification variables) et L_{con} un ensemble de constantes « étiquettes » (label constants) (iii) H un ensemble de constantes « trous » et (iv) R un ensemble de relations n-aires sur $I_{var} \cup I_{con}$.

Etant donné $l \in L_{var} \cup L_{con}$, $h \in H$, $i_1, \ldots, i_n \in I_{var} \cup I_{con} \cup H$ et $R^n \in R$, les formules d'unifications (désormais UF pour unifying formulas) de L_U sont définies de la manière suivante :

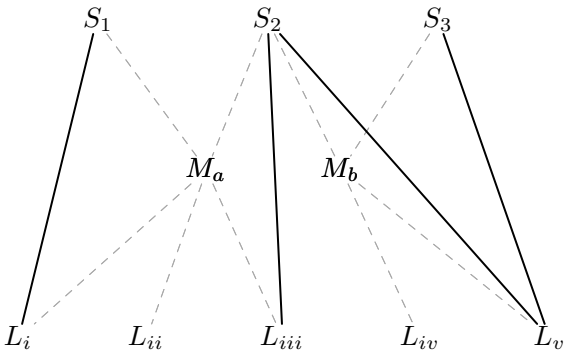


FIG. 2: Chaînes, sens et formes logiques

1. $l : R^n(i_1, \ldots, i_n)$ est une UF de L_U
2. $h \geq l$ est une UF de L_U
3. Si ϕ est une UF de L_U et ψ est une UF de L_U, alors ϕ, ψ est une UF de L_U
4. rien d'autre n'est une UF de L_U

F-2.2 Problème de l'équivalence des formes logiques

Le problème de l'équivalence des formes logiques a été mentionné pour les premières fois dans [Appelt, 1987] et [Shieber, 1988]. Il est impossible à traiter (à moins de résoudre le problème de la représentation des connaissances en intelligence artificielle) et il est l'une des raisons principales à l'utilisation d'une sémantique plate. Le problème part de considérations somme toute assez banales. On considère la grammaire comme une façon d'exprimer la relation entre des chaînes et une ou plusieurs formes logiques. Les formes logiques sont censées être une approximation relativement fiable du sens des chaînes. La relation entre les chaînes, les formes logiques et les sens peut se résumer de la manière suivante :

- Une chaîne peut avoir plus d'un sens ;
- Un sens peut être réalisé par plus d'une chaîne ;
- Un sens peut être représenté par plus d'une forme logique ;
- D'un autre côté, une forme logique représente un seul sens ;
- En pratique, on attend de la grammaire qu'elle associe une chaîne avec une seule forme logique (par sens). On appelle ceci la forme logique canonique de la chaîne.

La fonction d'un analyseur est de calculer le sens de chaque chaîne, en construisant la(les) forme(s) logique(s) canonique(s). Les systèmes de génération ont une fonction inverse qui consiste à dériver une chaîne à partir d'une représentation du sens. Mais ce n'est pas si simple. Si l'entrée ne correspond pas à une forme logique reconnue par la grammaire, nous n'obtenons pas de sortie. Considérons la figure 2. Si la grammaire n'associe pas L_{ii} à une chaîne et que le

générateur la reçoit en entrée, il ne produira aucun résultat. Plus concrètement, dans la grammaire jouet ci-dessous, la phrase "John has a cat" est associée à la sémantique $(\exists x.white(cat(x)) \wedge has(john, x))$. Si on avait fourni une entrée légèrement différente, par exemple $\exists x.has(john, x) \wedge white(cat(x))$, on n'aurait pas pu obtenir de sortie à partir de la grammaire, bien que les deux formes soient réellement sémantiquement équivalentes.

t1	S(S)	→ NP(X) VP($\lambda X.S$)
t2	VP($\lambda X.S$)	→ V($\lambda Y \lambda X.V$) NP($\lambda Y \lambda V.S$)
t3	NP($\lambda X \lambda C.S$)	→ Det($\lambda X \lambda R \lambda C.S$) N($\lambda X.R$)
t4	N($\lambda X.A \wedge N$)	→ Adj($\lambda X.A$) N($\lambda X.N$)
t5	NP($john$)	→ John
t6	N($\lambda X.cat(X)$)	→ cat
t7	Det($\lambda X \lambda R \lambda C.\exists X.(R \wedge C)$)	→ a
t8	Adj($\lambda X.white(X)$)	→ white
t9	V($\lambda Y \lambda X.has(X, Y)$)	→ has

Le problème de l'équivalence des formes logiques (désormais LFE pour Logical Form Equivalence) consiste simplement à trouver le moyen de générer du texte à partir de toutes les formes logiques qui ont le même sens. Le point crucial est que décider l'équivalence logique est indécidable en logique du premier ordre. Des algorithmes plausibles et pertinents (comme la conversion à la forme normale) ne contournent pas ce problème fondamental.

F-2.3 Le cas de la sémantique plate

La sémantique plate peut être utilisée comme une stratégie d'évitement du problème de l'équivalence des formes logiques. Si calculer l'équivalence logique est indécidable dans le cas général, on se tourne vers une notion plus modeste de l'équivalence qui est utile dans la plupart des cas concrets. Ce qu'on peut faire consiste à autoriser la commutativité et l'associativité de la conjonction. Calculer ces équivalences avec une sémantique plate peut être fait à moindre coût. On trie simplement les littéraux et on vérifie que les formes logiques triées sont identiques syntaxiquement.

Modifieurs intersectifs

Malheureusement, la commutativité et l'associativité de la conjonction ont un prix : l'ambiguïté de l'ordre des mots. Le manque de contraintes sur l'ordre des mots est problématique pour ce qu'on appelle les modifieurs intersectifs, c'est-à-dire pour les cas où plusieurs modifieurs affectent la même entité. Si une chaîne contient un mot avec k modifieurs, le réalisateur va produire 2^k versions de cette chaîne, soit une par sous-ensemble de modifieurs. Par exemple, voici les $2^3 = 8$ sous-ensembles possibles de modifieurs dans "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

Une solution proposée à ce problème est d'adopter une stratégie de réalisation en deux phases [Carroll *et al.*, 1999] : dans la première phase, les structures syntaxiques complètes sont construites sans les modifieurs ; dans la deuxième phase, les modifieurs sont insérés. La clé de cette technique réside dans le fait qu'elle évite d'insérer des modifieurs dans des structures syntaxiques *incomplètes*, ce qui serait une perte de temps. Ceci ne résoud pas effectivement le problème des modifieurs intersectifs, mais c'est réellement utile pour en contenir les effets négatifs en pratique. Adopter cette stratégie requiert que le réalisateur de surface supporte la notion d'adjonction, c'est à dire l'insertion d'une structure syntaxique au milieu d'une autre. Comme nous allons le voir plus loin, ceci est réalisé tout-à-fait naturellement avec le formalisme des grammaires d'arbres adjoints (grammaires TAG).

F-2.4 Synthèse

Les langages de représentation sémantique sont intrinsèquement récursifs. Les langages de sémantique plate ne sont pas moins récursifs, mais ils utilisent un mécanisme de pointage pour exprimer cette récursivité. De telles représentations sont particulièrement intéressantes pour la réalisation de surface parce qu'elles autorisent la commutativité et l'associativité de la conjonction. Ceci permet de traiter le problème de l'équivalence de la forme logique, au moins dans les cas les plus pratiques. Bien entendu, autoriser la commutativité et l'associativité de la conjonction présente des inconvénients, et particulièrement pose le problème des modifieurs intersectifs. Pour éviter cela, nous utilisons la technique de l'insertion différée des modifieurs.

F-3 Grammaires d'arbres adjoints basées sur l'unification

Les grammaires d'arbres adjoints (Tree Adjoining Grammars ou TAG) sont un formalisme qui génère des langages légèrement contextuels (mildly context-sensitive language). Ceci signifie que ce formalisme peut générer certains langages que les grammaires hors contextes ne peuvent pas générer, mais qu'il ne peut pas pour autant générer tous les langages contextuels. Ainsi, les TAG peuvent décrire le langage $a^n b^n c^n$ (soit le langage comprenant l'ensemble des chaînes consistant en des a suivis par le même nombre de b et de c), mais elles ne peuvent pas décrire le langage $a^n b^n c^n d^n e^e$. En d'autres termes, une grammaire TAG est plus puissante qu'une CFG, mais elle n'est pas *trop* puissante (elle peut toujours être analysée en temps polynomial, $\mathcal{O}(n^6)$, pour être précis). On utilisera les TAG lexicalisées basées sur l'unification (Feature-Based Lexicalised TAG ou FB-LTAG), une variante des TAG qui en conserve les propriétés formelles.

Une FB-LTAG comprend un ensemble d'arbres élémentaires et deux opérations permettant de combiner ces arbres entre eux, l'opération de substitution et l'opération d'adjonction. Les arbres résultant d'une de ces opérations sont appelés arbres dérivés. Les arbres élémentaires sont lexicalisés, c'est-à-dire qu'ils sont explicitement associés à une composante lexicale (lemme ou une forme fléchie). Leurs nœuds sont étiquetés par deux structures de traits appelées *top* et *bottom*. Un arbre élémentaire est soit initial, soit auxiliaire. Un arbre initial est un arbre dont les nœuds feuilles sont soit des nœuds terminaux, soit des nœuds dits de substitution (marqués par $\downarrow$). Un arbre auxiliaire est un arbre dont l'un des nœuds feuilles est un nœud pied (marqué par $\star$) étiqueté par la même catégorie que le nœud racine.

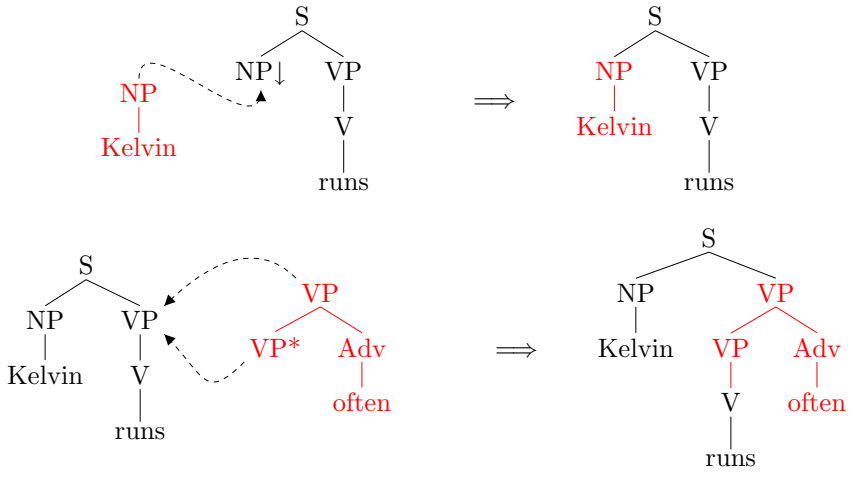


FIG. 3: Substitution et adjonction TAG

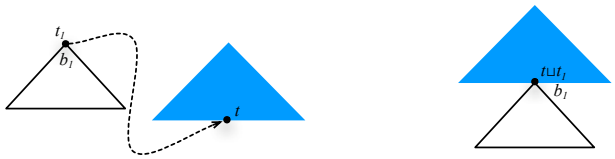


FIG. 4: Substitution TAG avec structures de traits

L'opération de substitution permet d'insérer un arbre élémentaire ou dérivé τ_δ dans un arbre initial τ_α : le nœud racine de τ_δ est alors identifié avec un nœud de substitution dans τ_α et leurs traits *top* sont unifiés ($top_{\tau_\alpha} = top_{\tau_\delta}$).

L'opération d'adjonction permet d'insérer un arbre auxiliaire τ_β dans un arbre quelconque τ_α à un nœud n : les traits *top* et *bottom* du nœud n où se fait l'adjonction sont alors unifiés avec les traits *top* du nœud racine de l'arbre auxiliaire et les traits *bottom* de son nœud pied respectivement ($top_n = top_{Root_{\tau_\beta}}$ et $bottom_n = bottom_{Foot_{\tau_\beta}}$). En fin de dérivation, les traits *top* et *bottom* de chaque nœud de l'arbre dérivé produit sont unifiés.

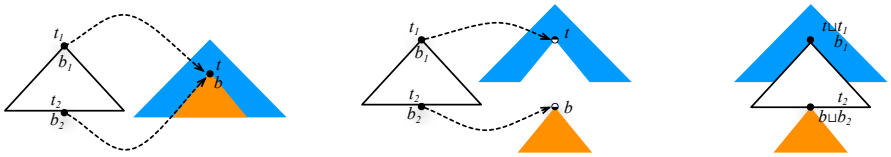


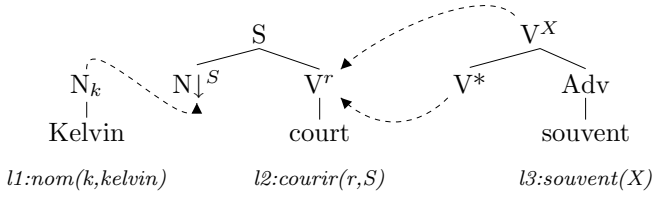
FIG. 5: Adjonction TAG avec structures de traits

F-3.1 Les FB-LTAG augmentées d'une sémantique plate L_U

Le lien entre structure syntaxique et représentation sémantique se fait de la façon illustrée par la figure ci-dessous. Chaque arbre élémentaire est associé avec une représentation sémantique où les arguments manquants sont des variables d'unification. Ces variables apparaissent en outre sur certains nœuds de l'arbre et sont instanciées par le biais des substitutions et des adjonctions (section 3.3). Ainsi, dans la dérivation de "Kelvin court souvent" illustrée ci-dessous, k s'unifie avec S et r avec X si bien que la représentation sémantique finale est *l1:nom(k,kelvin), l2:courir(r,k), l3:souvent(r).*

F-3.2 Synthèse

Le formalisme FB-LTAG est une petite extension du formalisme TAG, un formalisme grammatical légèrement contextuel. Il peut être combiné avec L_U, le langage sémantique utilisé dans cette thèse, en codant la sémantique dans des structures de traits.



F-4 GenI et SemFRaG

GenI est un réalisateur de surface développé par Carlos Areces et Claire Gardent [Areces, 2003]. Il utilise une grammaire FB-LTAG, une sémantique L_U et un algorithme ascendant de « chart generation ». Dans ce chapitre, nous présentons l'algorithme de départ, développé dans le cadre de l'action de recherche concertée INRIA GenI.
SemFRaG est une grammaire FB-LTAG pour le français doublée d'une sémantique L_U [Gardent, 2006]. Elle a été utilisée à la fois pour l'analyse avec une construction sémantique [Parmentier, 2007] et pour la réalisation de surface avec GenI.
La plus grande partie du travail accompli pour mettre en œuvre ce réalisateur a tourné autour du développement de la grammaire. Dans ce résumé, nous nous cantonnons à la description de GenI ; en revanche, dans la thèse, nous décrivons brièvement les systèmes de génération qui lui sont apparentés en fin de chapitre.

F-4.1 GenI

GenI utilise un algorithme ascendante de génération tabulaire optimisé pour les grammaires d'arbres adjoints. Nous illustrons son fonctionnement ci-dessous avec un court exemple :

Supposons que la sémantique donnée en entrée soit *l2:courir(r,k), l1:nom(k, kelvin), l3:souvent(r)*. L'algorithme procède en quatre étapes : une phase de sélection lexicale, une phase de substitution, une d'adjonction et une d'extraction.

Dans un premier temps, les arbres élémentaires dont la sémantique subsume une partie de l'entrée sont sélectionnés (c'est ce que nous appellerons la phase de sélection lexicale).

La sélection lexicale se fait donc à partir (i) d'une sémantique d'entrée et (ii) d'une grammaire FB-LTAG. La grammaire consiste en un ensemble d'items lexicaux regroupant un arbre élémentaire et une sémantique lexicale. Plus précisément :

Définition 2 (item lexical)**.** Un item lexical est une paire $\langle T, S\rangle$, où T est un arbre élémentaire FB-LTAG et S une formule L_U. Les variables d'unification dans l'item lexical ont une portée sur la totalité de l'item, soit T et S.

Pour notre exemple, les arbres sélectionnés seront (entre autres) les arbres de “Kelvin, court” et “souvent”.

La deuxième étape (que nous appelons phase de substitution) consiste à explorer systématiquement les possibilités de combinaisons par substitution. Pour l'exemple considéré, cette exploration permettra de substituer l'arbre pour "Kelvin" dans l'arbre pour "court".

Notez que dans le tableau ci-dessous, les lettres 'k', 'c' et 's' représentent les arbres élémentaires qui correspondent respectivement à "Kelvin", "court" et "souvent".

Combinaison	Agenda	Charte	AgendaA
	k,c		s
	c,	k	s
↓(c,k)	kc	c,k	s
		c,k,kc	s

La troisième étape (phase d'adjonction) permet de combiner les arbres produits par adjonction. C'est à ce stade que l'arbre pour "souvent" sera adjoint à l'arbre dérivé pour "Jean court". En dernier ressort (phase d'extraction), les chaînes étiquetant les items couvrant la sémantique donnée en entrée sont produites en l'occurrence : "Kelvin court souvent".

Combinaison	Agenda	Charte	Résultats
	k,kc	s	
	kc	s	
⋆(kc,s)	kcs	s	
		s	kcs

F-4.2 SemFRaG

SemFRaG est une grammaire noyau pour le français. Elle combine une syntaxe FB-LTAG et une sémantique L_U. Le but de SemFRaG est de servir de grammaire paraphrastique, c'est-à-dire une grammaire associant des réalisations syntaxiques différentes, avec le même sens, à une forme logique unique. Par exemple, les phrases "Jean aime Marie" et "Marie est aimé par Jean" auront la même représentation sémantique (*l1:aimer(e), l1:agent(e,j), l1:patient(e,m)*). Pour pouvoir nous faire une idée de ses capacités de paraphrase, nous avons construit un suite de tests comprenant plus de 80 cas. Nous avons obtenu un total de 1582 phrases, avec une moyenne de 18 paraphrases par cas.

F-4.3 Synthèse

GenI est un réalisateur de surface fondé sur le formalisme FB-LTAG augmenté d'une sémantique L_U. La version de base de l'algorithme utilise une stratégie en deux phases de génération tabulaire qui construit des expressions syntaxiquement complètes et ajoute les modifieurs selon les besoins. Ceci conclut notre survol de l'état de l'art. Dans les trois chapitres suivants de la thèse, nous présentons notre propre extension de l'algorithme noyau dont il a été question ici.

F-5 Filtrage par polarité

La réalisation de surface est une tâche complexe pour deux raisons : l'ambiguïté lexicale et le manque de contraintes sur l'ordre des mots. Dans la première partie de notre thèse, nous avons présenté des techniques pour gérer cette complexité en général, ainsi que des techniques pour traiter le manque de contraintes sur l'ordre des mots (e.g. l'insertion différée des modifieurs). Dans ce chapitre, nous nous concentrons sur l'ambiguïté lexicale qui constitue l'autre source de la complexité de la tâche de réalisation. Ce chapitre propose une façon de limiter les effets de l'ambiguïté lexicale. L'idée est inspirée de l'« étiquetage électrostatique » présenté par [Perrier, 2003], qui consiste essentiellement à élaguer l'espace de recherche initial en appliquant un filtre global sur la première combinaison possible d'items lexicaux.

F-5.1 Ambiguïté lexicale

Dans la réalisation de surface, l'ambiguïté lexicale entre en jeu après l'étape de sélection lexicale. A ce moment-là, on a accédé à l'ensemble d'items lexicaux dont la sémantique subsume (une partie de) la sémantique d'entrée. Par exemple, étant donné la sémantique :

l0:tableau(t), l1:coût(c,t,g), l2:élevé(g).

Un réalisateur de surface combiné à une grammaire et un lexique donnés devrait sélectionner les items lexicaux suivants (voir la figure 6).

- $\tau_{peinture}$ ou $\tau_{tableau}$ pour *l0:tableau(t)*;
- τ_{cout} (le nom) ou τ_{coute} pour *l1:coût(c,t,g)*;
- τ_{eleve} ou τ_{cher} pour *l2:élevé(g)*.

Cette sélection est ici ambiguë dans le sens où il y a plus d'un item disponible par partie de la sémantique d'entrée. A ce stade, le nombre de combinaisons *a priori* possibles est

$$\prod_{1 \leq i \leq n} a_i$$

où a_i est le degré d'ambiguité lexicale du i-ème littéral, et n, le nombre de littéraux dans la sémantique donnée en entrée. En d'autres termes, l'espace de recherche est exponentiel par rapport au nombre de littéraux.

F-5.2 Automates de polarité

Le filtrage par polarités [Perrier, 2003] repose sur l'observation qu'un grand nombre des séquences *a priori* possibles ne peuvent pas être valides au plan syntaxique. Supposons par exemple, que la représentation sémantique donnée en entrée soit *l0:tableau(t), l1:cout(c,t,g), l2:élevé(g).* Pour cet ensemble de littéraux, les arbres TAG sélectionnés pourront inclure ceux indiqués dans le tableau ci-dessous. A partir de ce tableau, certaines séquences mènent à une phrase bien formée, d'autres non. Ainsi, la combinaison $\tau_{tableau}$, τ_{coute}, τ_{cher} permettra de générer "Le tableau coûte cher"; $\tau_{tableau}$, τ_{cout}, $\tau_{est\ eleve}$ permettra de générer "Le coût du tableau est élevé". Mais la séquence $\tau_{est\ eleve}$ $\tau_{peinture}$, τ_{coute} ne peut pas mener à une phrase bien formée puisqu'elle contient deux verbes conjugués et un seul argument.

l0:tableau(t)	*l1:cout(c,t,g)*	*l2:élevé(g)*
$\tau_{tableau}$	τ_{cout}	$\tau_{est\ eleve}$
$\tau_{peinture}$	τ_{coute}	τ_{cher}

Afin de détecter les séquences invalides, l'information combinatoire présente de façon implicite dans la grammaire par le biais en particulier des nœuds de substitution, des nœuds pieds et des nœuds racines, est associée de façon explicite avec chaque arbre élémentaire sous la forme de polarités. Une polarité se compose d'une étiquette et d'un nombre relatif, sa charge. Elle reflète le potentiel combinatoire de l'arbre auquel elle est associée. Ainsi dans l'exemple ci-dessus, les arbres τ_{coute} et $\tau_{est\ eleve}$ seront associés à la polarité `-1gn` ce qui, intuitivement, signifie que ces arbres ont besoin d'un arbre GN (Groupe Nominal) pour être complets. En contrepartie, les arbres GN auront la polarité `+1gn`, ce qui signifie qu'ils procurent un GN.

l0:tableau(t)	*l1:cout(c,t,g)*	*l2:grand(g)*
$\tau_{tableau}$ `+1gn`	τ_{cout} `0gn`	$\tau_{est\ eleve}$ `-1gn`
$\tau_{peinture}$ `+1gn`	τ_{coute} `-1gn`	τ_{cher} `0gn`

Toute combinaison d'arbres lexicaux dont la charge totale n'est pas nulle est nécessairement syntaxiquement invalide : soit certains besoins ne sont pas satisfaits, soit certaines ressources ne sont pas utilisées. Par exemple, la combinaison τ_{cout}, $\tau_{tableau}$, τ_{cher} a une charge de `+1gn` et est donc éliminée. En revanche, la séquence τ_{cout}, $\tau_{tableau}$, $\tau_{est\ eleve}$ a une charge nulle ce qui sans

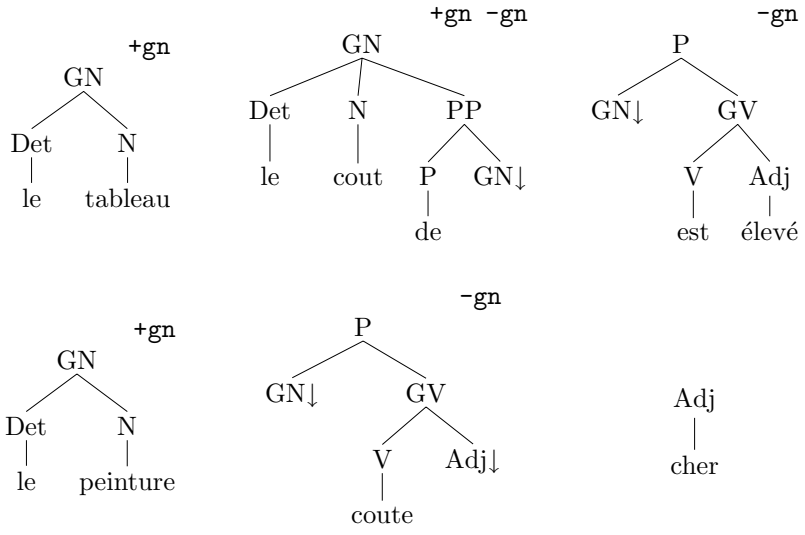


FIG. 6: Calcul de la polarité des arbres sélectionnés lexicalement

garantir sa validité syntaxique, ne l'invalide pas — cette séquence sera donc conservée et explorée par le générateur.

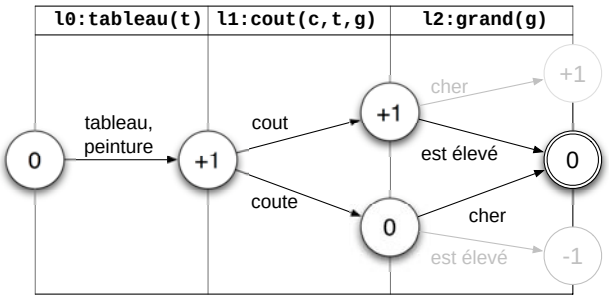


FIG. 7: Automate de polarités après minimisation

Le filtrage par polarités est implémenté comme dans [Perrier, 2003] par le biais d'un automate à états finis. Cet automate encode toutes les séquences possibles d'items lexicaux couvrant la sémantique d'entrée. Chaque transition est étiquetée avec le nom d'un arbre élémentaire réalisant une partie de la sémantique d'entrée et chaque état avec la polarité cumulée des transitions menant à cet état. L'état final de l'automate a une charge nulle et une minimisation [Hopcroft and Ullman, 1979] de l'automate est exécutée qui élimine de l'automate tous les états non finaux. L'automate résultant représente ainsi uniquement les combinaisons d'items lexicaux qui couvrent la sémantique donnée en entrée et dont la charge totale est nulle. La figure 7 montre l'automate final pour l'exemple discuté ci-dessus.

Pour préserver la factorisation permise par l'usage de la charte pendant la génération, le filtrage par polarité doit en outre être intégré avec l'algorithme de réalisation. En effet, certains chemins dans l'ensemble des chemins définis par l'automate peuvent avoir des parties communes. Pour éviter de calculer plusieurs fois ces parties communes, chaque arbre lexical est annoté avec l'ensemble des chemins auquel il appartient. Pendant la phase de réalisation, deux items ne sont comparés que si l'intersection de leur ensemble de chemins n'est pas vide (ils apparaissent dans le même chemin). Le résultat d'une combinaison est étiqueté avec l'intersection des étiquettes des constituants combinés. De cette façon, les items lexicaux apparaissant dans différents chemins de l'automate ne sont introduits qu'une seule fois dans la charte et la factorisation des arbres élémentaires ou dérivés communs à plusieurs chemins peut être assurée.

F-5.3 Évaluation du filtrage par polarités

Il y a deux questions pratiques qui doivent être posées à propos de l'utilisation de l'automate de polarité. Il faut d'abord savoir à quel point le filtrage est efficace dans la réduction des effets de l'ambiguïté lexicale, autrement dit dans l'élimination de certaines combinaisons lexicales. Il faut ensuite s'assurer que la dépense supplémentaire consistant à construire l'automate de polarité vaut la peine d'être faite.

Clairement, le filtrage par polarité réduit l'effet de l'ambiguïté lexicale. Il y a deux façons de le mesurer, d'abord en regardant l'effet direct sur le nombre

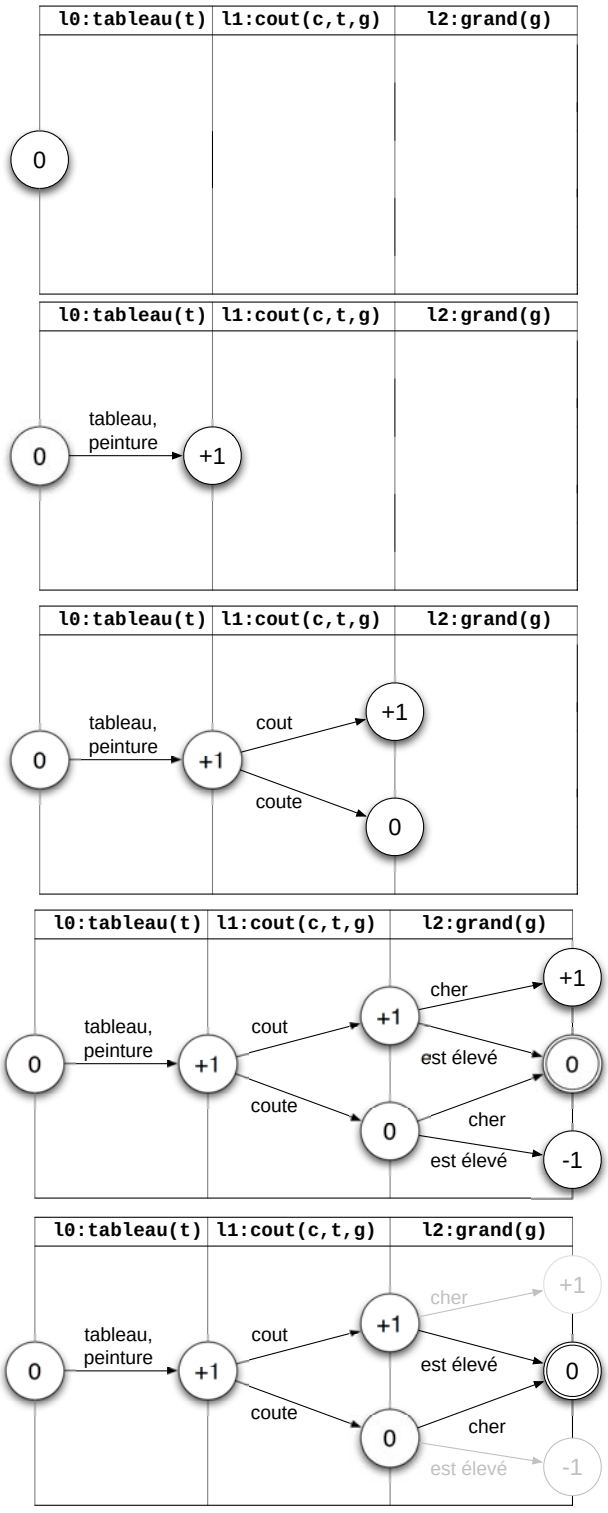


FIG. 8: Construction d'un automate de polarités

de combinaisons lexicales, ensuite en mesurant des artefacts indirects comme le nombre d'arbres dérivés construits par le réalisateur de surface. Ci-dessous, nous présentons les résultats concernant notre test « le pire » (le plus grand, comprenant 8 littéraux, sans compter les rôles thématiques et permettant 231 paraphrases) :

	sans filtrage	**avec filtrage**
combinaisons lexicales	2 436 672	4 136
substitutions	26 149	3 284
adjonctions	5 014	630

Pour expliquer comment nous arrivons au calcul du nombre de combinaisons lexicales sans filtrage, nous devons préciser qu'au départ de la construction de l'automate, toutes les polarités sont neutralisées. Cette première version de l'automate est appelée automate source (seed automaton). Le nombre de combinaisons sans filtrage correspond au nombre de chemins que contient l'automate source. Chaque chemin correspond à une combinaison d'items lexicaux. Pour obtenir le nombre de combinaisons lexicales avec filtrage, nous avons compté les chemins de notre automate de polarité, une fois que des valeurs ont été assignées aux polarités.

La comparaison de ces deux nombres fournit une estimation de l'utilité du filtre de polarité. Dans notre suite de tests, l'automate source a pour chaque cas entre 1 et 2 436 672 chemins. L'automate final a entre 1 et 4 136 chemins. Ceci représente une réduction importante de l'espace de recherche (puisque nous avons éliminé 99,9% des combinaisons).

Là encore, l'effet concret de l'élimination de ces chemins n'est pas clair. En effet, en pratique, un réalisateur de surface naïf pourrait peut-être rapidement élaguer les parties de l'espace de recherche qui impliquent des combinaisons que le filtrage par polarité aurait éliminé; aussi, l'importante réduction du nombre de combinaisons lexicales pourrait ne pas signifier grand chose. Une présentation plus convaincante du bénéfice du filtrage par polarité ne mesurerait pas seulement combien de combinaisons lexicales nous avons éliminées, mais quelles sont les conséquences de cette élimination pour la réalisation de surface. C'est à ce moment-là que le comptage des substitutions et des adjonctions entre en jeu. Chacun de ces comptages sert à estimer la quantité de travail effectivement produite par le réalisateur de surface. Sans le filtrage, il a effectué 26 149 substitutions et 5014 adjonctions. Avec le filtrage, il tombe à 3284 substitutions et 630 adjonctions. En bref, le filtrage par polarité a un impact réel à la fois sur la taille de l'espace de recherche, et sur ce que le réalisateur explore réellement.

Nous savons donc maintenant que le filtrage par polarité peut apporter une bonne réduction de l'espace de recherche dans la réalisation de surface. Une question autrement plus compliquée est la suivante : est-ce que le coût de la construction, de la minimisation et de l'intersection de ces automates compense les gains que nous faisons? En effet, les temps de réalisation avec et sans filtrage sont comparables pour la majorité des cas de la suite de tests. Pourtant, si on observe les résultats du réalisateur pour la phrase la plus complexe de la suite de tests, on constate que le filtrage par polarité permet de rendre la réalisation de surface plus rapide à 98,4%, et produit un résultat en 25,21 secondes CPU au lieu de 1615,7.

F-6 Sélection de paraphrases

Dans le chapitre précédent, nous avons exploré une technique permettant de surmonter l'ambiguïté lexicale. Pourtant, on pourrait dire que l'ambiguïté lexicale ne devrait pas être possible, qu'après tout, elle révèle un défaut fondamental dans la conception du générateur, que la grammaire ou le formalisme sémantique d'entrée ne sont pas assez précis. Dans notre cas, pourtant, l'ambiguïté lexicale est totalement délibérée. C'est bien plus une caractéristique souhaitée qu'un défaut. Elle existe parce que le réalisateur de surface et la grammaire sont conçus pour traiter les paraphrases syntaxiques qui ont la même sémantique.

Par exemple, une phrase comme "Jean aime Marie" aura la sémantique suivante : *l1:jean(j) l2:aimer(e,j,m) l3:marie(m)*, et ses nombreuses paraphrases seront :

(3) a. Marie est aimée par Jean
b. C'est Jean qui aime Marie
c. C'est Jean de qui est aimée Marie
d. C'est par Jean que Marie est aimée
e. C'est par Jean qu'est aimée Marie
f. C'est Jean dont est aimée Marie
g. C'est Marie que Jean aime
h. ...

Malgré tout, avoir une longue liste de paraphrases possibles pour une sémantique donnée n'est pas particulièrement utile pour les applications pratiques de la génération. Dans ce chapitre, nous présentons une technique pour que le réalisateur de surface ne donne qu'une seule phrase par sémantique d'entrée. L'approche que nous utilisons a été construite autour de l'intuition que la sélection de paraphrase est étroitement liée à la sélection lexicale. Elle consiste essentiellement à enrichir l'entrée avec un ensemble de propriétés linguistiques (comme par exemple préciser que *l2:aimer(e,j,m)* doit être réalisée à la forme active) qui contrôle le choix initial des items lexicaux et par conséquent, les paraphrases produites.

F-6.1 Paraphrases contextuellement (in)appropriées

Avant d'explorer le mécanisme de sélection en détail, il est important d'examiner plus attentivement nos objectifs. Pour commencer, il y a deux mauvaises raisons pour vouloir produire une seule paraphrase. La première raison consisterait à dire que cela pourrait être utile en soi, ou que les applications de GLN requièrent seulement une sortie. Si c'était le cas, on pourrait simplement générer toutes les sorties et en choisir une au hasard. La deuxième mauvaise raison est une raison d'efficacité. Alors qu'on pourrait penser que l'efficacité est un but important, une motivation incontestable pour arriver à ne produire qu'une seule paraphrase, elle ne constitue pas la principale raison à notre façon d'agir.

Notre motif principal peut être illustré par cet exemple, qui vient de [Halliday, 1978] et qu'on retrouve dans la littérature sur KPML [Bateman *et al.*, 1992] :

(4) Now comes the President here. It's the window he's stepping through to wave to the crowd. On his victory his opponent congratulates him. What they are shaking now is hands. A speech is going to be made by him. "Gentleman and ladies. That you are confident in me honours me. We shall, hereby pledge I, turn this country into a place, in which what people do safely will be live, and the ones who grow up happily will be able to be their children."

Le passage ci-dessus peut nous sembler être de l'anglais parfaitement raisonnable au premier abord, mais quand on le lit attentivement, on s'aperçoit que ce texte est vraiment bizarre. On peut identifier les problèmes suivants [Halliday, 1985] :

- Le circonstant de temps "now" a été marqué thématiquement dans la première phrase, là où "here" aurait été plus naturel.
- La forme clivée "It's the window he's..." est utilisée dans la deuxième phrase, bien qu'on n'exprime pas de contraste.
- Dans la dernière phrase, "A speech is going to be made by him", on utilise la forme passive, alors que l'actif aurait été meilleur ou tout au moins, plus naturel.

Aucun des problèmes cités n'est, à proprement parler, une erreur de grammaire. Le passage ci-dessus est composé de phrases techniquement correctes, bien que contextuellement inappropriées. Ces phrases ne correspondent tout simplement pas aux besoins du contexte (qui inclut mais ne se limite pas au cotexte et aux connaissances de l'utilisateur).

Un réalisateur de surface devrait par conséquent fournir des outils pour éliminer toute paraphrase contextuellement inappropriée de la sortie qu'il fournit. Mieux encore, il devrait avoir pour but de produire *la meilleure paraphrase pour une entrée donnée*. La raison pour laquelle nous insistons sur le fait de produire une seule paraphrase, et non les quatre ou cinq meilleures repose sur le principe affirmant que « tous les choix linguistiques ont un sens » tel qu'on le trouve dans la linguistique systémique fonctionnelle (Systemic Functional Linguistic) [Winograd, 1983]. Chaque forme syntaxique est choisie délibérément. En s'assurant qu'on peut produire au plus une seule paraphrase, on s'assure que tous les choix qui sont faits sont explicites et accessibles à l'utilisateur. Notons simplement qu'être capable de se restreindre à une seule paraphrase ne nous oblige pas à le faire. Idéalement le mécanisme de sélection des paraphrases devrait autoriser l'utilisateur à faire des choix explicites parmi des propositions ou à laisser certaines décisions au réalisateur (SURGE, notamment, possède cette flexibilité [Elhadad and Robin, 1999]). Dans tous les cas, cette option de rendre les choix explicites devrait toujours être disponible.

F-6.2 Mécanisme de sélection

La sélection de paraphrase peut largement être réalisée en contraignant la sélection lexicale. Nous allons étendre le processus de sélection lexicale et la sémantique d'entrée pour exprimer de telles contraintes, mais avant d'entrer dans les détails, esquissons un scénario. Considérons la sémantique d'entrée ci-dessous, et imaginons une grammaire hypothétique qui l'associe aux trois paraphrases données :

(5) *l1:give(e,j,b,m), l2:john(j), l3:mary(m)*
John gave the book to Mary.
Mary was given the book by John.
The book was given to Mary by John.

Pour restreindre la sélection, on pourrait demander que le littéral *l1:give(e,j,b,m)* soit réalisé par un verbe à la forme passive. On ré-écrit alors la sémantique d'entrée de la manière suivante avec les conséquences attendues.

(6) *l1:give(e,j,b,m)[PassiveForm], l2:john(j), l3:mary(m)*
~~John gave the book to Mary.~~
Mary was given the book by John.
The book was given to Mary by John.

Ajouter des propriétés supplémentaires à la sémantique d'entrée limite simplement un peu plus la sortie résultante. Nous restreignons encore le littéral et nous obtenons :

(7) *l1:give(e,j,b,m)[PassiveForm, CanonicalToObject], l2:john(j), l3:mary(m)*
~~John gave the book to Mary.~~
~~Mary was given the book by John.~~
The book was given to Mary by John.

Les étiquettes avec lesquelles nous avons enrichi la sémantique d'entrée sont appelées les propriétés d'arbre. Nous allons voir maintenant comment elles sont utilisées exactement, et d'où elles viennent en pratique. A partir de ces idées, nous allons aussi voir comment les propriétés d'arbre peuvent être utilisées pour restreindre la sémantique de façon à ce que nous produisions au plus *une* sortie.

F-6.3 Items lexicaux enrichis et sémantique d'entrée

Ces extensions par propriété d'arbre requièrent qu'on introduise des versions enrichies du lexique et de la sémantique d'entrée, les deux tenant compte des propriétés d'arbre. L'idée de base repose sur le fait que les linguistes utilisent les propriétés d'arbre pour décrire les items lexicaux et que le réalisateur de surface les utilise pour filtrer la sélection lexicale.

Définition 3 (Propriété d'arbre)**.** Une propriété d'arbre est un identifiant. Des propriétés d'arbre peuvent être par exemple *PassiveForm* (on demande à ce que la phrase soit réalisée au passif) et *CanonicalToObject* (on demande à ce que la phrase soit réalisée avec l'objet canonique introduit par «to»).

Définition 4 (Item lexical enrichi)**.** Un item lexical enrichi est un triplet $\langle T, S, LTP \rangle$. T et S sont respectivement l'arbre élémentaire et la sémantique lexicale telle qu'elle est décrite dans la définition 2 (page xvi). LTP est un ensemble de propriétés d'arbre.

Définition 5 (Sémantique d'entrée enrichie)**.** Une sémantique d'entrée enrichie est un ensemble de littéraux enrichis de la forme $L[tp_1, \ldots, tp_n]$ où L est un littéral L_U saturé et $tp_1, \ldots, tp_n$ un ensemble de propriétés (qui peut être vide). Pour simplifier la notation, nous n'écrivons pas les crochets quand l'ensemble de propriétés d'arbre est vide.

Définition 6 (Sémantique d'entrée simple)**.** La sémantique d'entrée simple est ce qu'il reste quand on a enlevé toutes les propriétés d'arbre de la sémantique enrichie. Etant donné une sémantique d'entrée enrichie, ES, on dira que la sémantique d'entrée simple est l'ensemble des littéraux de la forme L_i où $L_i[tp_1, \ldots, tp_n] \in ES$

F-6.4 Sélection lexicale enrichie

Prendre en compte les propriétés d'arbre consiste à filtrer les items lexicaux de manière à ce que seuls ceux qui présentent les propriétés d'arbre souhaitées soient retenus. Etant donné une sémantique enrichie ES, on instancie le lexique comme on a l'habitude de le faire (section F-4.1) et on retourne l'ensemble des items lexicaux enrichis tel que pour chaque item $\langle T, S, LTP \rangle$:

- sa sémantique instanciée S est non vide et subsume la sémantique d'entrée simple ;
- pour chaque littéral enrichi $L[tp_1, \ldots, tp_n]$ de la sémantique d'entrée enrichie ES, si $L \in S$ alors $\{tp_1, \ldots, tp_n\} \subseteq LTP$.

F-6.5 D'où viennent les propriétés d'arbres (métagrammaires)

Le mécanisme de sélection d'arbre requiert que chaque item lexical dans la grammaire soit associé avec un ensemble de propriétés d'arbre. Il y a plusieurs moyens de parvenir à ce but intermédiaire. Une solution possible pourrait être l'annotation manuelle, mais ce n'est ni souhaitable ni nécessaire. Ce n'est pas souhaitable parce que les grammaires TAG réalistes sont assez grosses pour rendre le processus sujet aux erreurs et encombrant. Ce n'est par ailleurs pas nécessaire dans le cadre de SemFRaG parce que les annotations sont déjà codées dans une autre ressource, la métagrammaire à partir de laquelle SemFRaG a été compilée.

Les métagrammaires sont des représentations hautement factorisées de la grammaire qui peuvent alors être compilées dans des formes plus explicites et familières. L'utilisation des métagrammaires est motivée par la quantité de redondances qui peuvent être trouvées dans une grammaire TAG typique. Le système qu'on utilise ici est le compilateur de métagrammaire XMG décrit dans [Crabbé and Duchier, 2004]. Nous allons maintenant entrer dans les détails concernant ce qui forme une telle métagrammaire. Au départ, une métagrammaire XMG consiste en des fragments d'arbres nommés (un fragment d'arbre est un ensemble de contraintes de dominance et de précédence linéaire) qui sont combinés par des disjonctions et des conjonctions. Pour avoir une idée de ce à quoi peut ressembler une métagrammaire, voici un exemple de fragments combinés pour générer des arbres de la famille des verbes transitifs. Ils sont combinés dans la figure 9.

$$Subject \wedge \\ ((ActiveForm \wedge Object) \vee (PassiveForm \wedge CAgent))$$

En termes XMG, un arbre élémentaire peut être vu comme une conjonction de fragments. Comme nous pouvons le voir dans l'exemple ci-dessus, les fragments d'arbres ont des noms qui ressemblent énormément aux propriétés

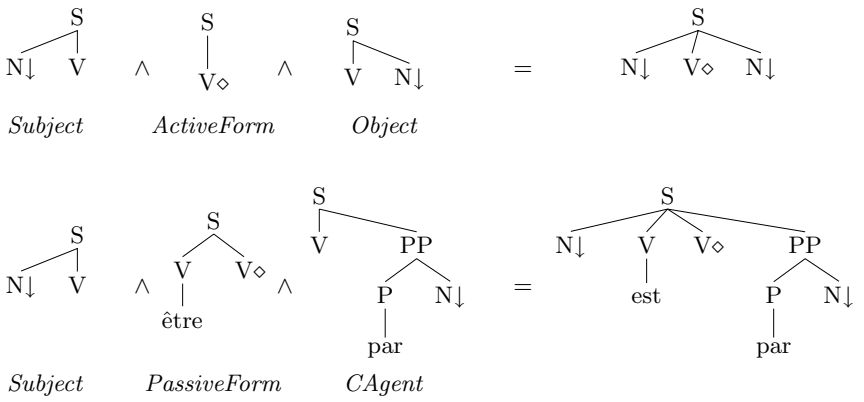


FIG. 9: Fragments d'arbres XMG (simplifiés de SEMFRAG)

d'arbres qu'on a utilisées dans nos exemples. Ceci est lié au fait que dans la grammaire SEMFRAG, on se sert des identifiants XMG pour trouver les propriétés d'arbre. L'ensemble des propriétés d'arbres possédé par un arbre élémentaire est l'ensemble des noms de fragments d'arbres à partir duquel cet arbre élémentaire a été construit par la métagrammaire.

F-6.6 Produire au plus une sortie

Plus il y a de propriétés d'arbre utilisées pour enrichir la sémantique d'entrée, moins on sélectionne d'items lexicaux. En allant jusqu'au bout de cette logique, on pourrait tout à fait ajouter assez de propriétés d'arbre pour que chaque littéral soit réalisé par au plus un item lexical (On précise « au plus » à cause des items dont la sémantique est composée de plusieurs littéraux). La plupart du temps, avoir une sélection lexicale débarrassée de toute ambiguïté est une condition suffisante pour garantir que le réalisateur de surface retourne une paraphrase unique, les exceptions étant les entrées où l'absence de contrainte sur l'ordre des mots entre en jeu (se reporter au cas des modifieurs intersectifs). Les propriétés d'arbres associées avec chaque item lexical doivent permettre de l'identifier *de façon unique*. En d'autres termes, chaque item lexical enrichi doit être associé avec ce qu'on appelle un identifiant d'arbre.

Définition 7 (Identifiant d'arbre)**.** Dans une grammaire FB-LTAG, un identifiant d'arbre est un ensemble de propriétés d'arbre I. Si dans un ensemble d'entrées lexicales il y a seulement un item lexical $\langle T, S, LTP\rangle$ tel que $LTP \subseteq I$, on dit que l'identifiant d'arbre est unique dans cet ensemble.

Ce qui est important ici n'est pas seulement le fait que les identifiants d'arbre soient uniques. Après tout, les arbres élémentaires produits par XMG on déjà des noms uniques (comme par exemple `Tn0Vn1-387`) qui pourraient techniquement nous permettre de parvenir au même résultat. Mais ceux-ci ne sont pas aussi pratiques que les identifiants d'arbre, étant donné qu'ils sont totalement arbitraires, et qu'ils ne sont pas dotés de la signification linguistique

des propriétés d'arbre. La raison pour laquelle la signification linguistique a de l'importance est qu'elle nous amène plus près de l'objectif consistant à choisir des paraphrases contextuellement appropriées. Sommairement, nous avons besoin de moyens de représenter l'ensemble des alternatives linguistiques et de faire un choix pour chacune d'elles. Nous pensons que les disjonctions dans la métagrammaire servent de mécanisme pour représenter ces alternatives et que les propriétés d'arbres nous fournissent le mécanisme pour faire notre choix.

Les exemples 8a–8c illustrent l'utilisation des propriétés d'arbre pour la sélection de paraphrase dans SemFraG. On se reportera à l'annexe B pour la liste complète des propriétés d'arbre de la grammaire.

(8) a. *l1:jean(j)[ProperName]*
l2:aimer(e,j,m)[CanonicalNominalSubject, ActiveVerbForm, CanonicalNominalObject]
l3:marie(m)[ProperName]
Jean aime Marie
~~Marie est aimé de Jean~~

b. *l1:le(c)[Det]*
l1:chien(c)[Noun]
l2:dormir(e1,c)[RelativeSubject]
l3:ronfle(e2,c)[CanonicalSubject]
Le chien qui dort ronfle
~~Le chien qui ronfle dort~~

c. *l1:jean(j)[ProperName]*
l2:promettre(e1,j,m,e2)[CanonicalNominalSubject, ActiveVerbForm, CompletiveObject]
l3:marie(m)[ProperName]
l4:partir(e2,j)[InfinitivalVerb]
Jean promet à Marie de partir
~~Jean promet à Marie qu'il partira~~

F-6.7 Evaluation

Afin d'évaluer d'une part, la capacité paraphrastique du réalisateur et d'autre part, l'impact des annotations de contrôle sur le non-déterminisme, nous avons utilisé une suite de tests graduée. Cette suite a été construite en (i) analysant un ensemble de phrases et (ii) sélectionnant pour chaque phrase la représentation sémantique correcte[1]. Le résultat est une suite de 80 représentations sémantiques choisies pour illustrer les différents types de paraphrases grammaticales décrites par la grammaire utilisée c'est à dire :

- les variations grammaticales dans la réalisation des arguments (clivés, cliticisation, extraction, inversion du sujet, etc.) et la forme du verbe (passive/active, impersonnelle, etc.)
- les variations dans la réalisation des modificateurs (anté- vs. post-posés, adjectif vs. subordonnée relative, épithète vs. attribut, etc.)
- les variations permises par une équivalence morpho-dérivationnelle (ex : arrivée/arriver)

[1] L'analyseur peut donner plusieurs analyses et donc souvent plusieurs représentations sémantiques dont certaines représentent correctement le sens de la phrase analysée , d'autres non.

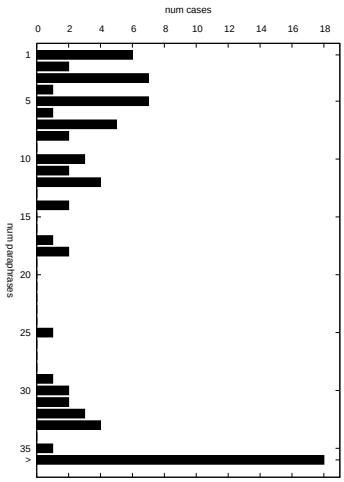


FIG. 10: Distribution de la complexité paraphrastique

Les 80 cas sélectionnés donnent lieu à la génération par GENI de 1 528 phrases distinctes soit un taux de paraphrases moyen par entrée de 18 avec une variation allant de 1 à plus de 50 paraphrases par représentation sémantique. La figure 10 donne une description plus détaillée de la distribution du taux de variation paraphrastique. Plus généralement, 42% des phrases avec un verbe fini ont une à 3 paraphrases (cas des verbes intransitifs), 44% 4 à 28 paraphrases (verbes prenant deux arguments) et 13% acceptent plus de 30 paraphrases (verbes à trois arguments). Pour les phrases contenant deux verbes finis, le ratio est de 5% des cas ayant 1 à 3 paraphrases, 36% des cas ayant entre 4 et 14 paraphrases et 59% plus de 14 paraphrases. Enfin les phrases contenant plus de 3 verbes finis acceptent toutes plus de 20 paraphrases.

Afin de vérifier que l'utilisation des profils suffit à assurer le déterminisme, nous avons calculé le nombre de cas où deux paraphrases d'un même contenu partagent le même profil. Pour ce faire, nous avons étiqueté de façon automatique les 1 528 paraphrases produites par GENI à partir de la suite de test, avec leur profils (le profil d'une paraphrase est l'ensemble des profils associés aux arbres élémentaires utilisés pour construire l'arbre dérivé de cette paraphrase). Nous avons ensuite comparé, pour chaque entrée de la suite de tests, les profils de toutes les paires de paraphrases correspondantes et compté le nombre de fois où une paire de paraphrases partage le même profil.

Cette manipulation montre que pour les 1 528 paraphrases considérées, le profil échoue à refléter la différence entre deux paraphrases dans moins de 2% des cas. L'analyse des données fautives révèle que les cas posant problème sont les paires paraphrastiques impliquant uniquement (i) une variation d'ordre des arguments (Ex. " Jean donne une pomme à Marie / Jean donne à Marie une pomme ") ou (ii) une variation de position pour un modificateur (Ex. "Jean donne ce soir une pomme à Marie /Jean donne une pomme ce soir à Marie

/ Jean donne une pomme à Marie ce soir"). Le premier cas peut être résolu en modifiant la grammaire de façon à expliciter la différence dans le profil des arbres élémentaires correspondant, le second en imposant un ordre canonique sur l'adjonction des modificateurs.

F-7 Réduction de la surgénération

Une grammaire générative devrait produire toutes les phrases de la langue qu'elle décrit et seulement ces phrases. En pratique cependant, les grammaires surgénèrent et sous-génèrent en même temps. La sous-génération a lieu quand une grammaire ne reconnaît pas toutes les chaînes dans la langue cible. Cela signifie qu'on peut échouer dans l'analyse de phrases grammaticales ou dans la production de paraphrases valides pour une sémantique d'entrée. La surgénération, d'un autre côté, a lieu quand la grammaire autorise trop de chaînes. Cela signifie que les réalisateurs de surface comme GenI ne vont pas seulement produire des paraphrases valides, mais aussi des phrases assez peu naturelles, des chaînes qui n'appartiennent pas à la langue ou des chaînes qui ne peuvent pas être associées à la sémantique d'entrée. La surgénération peut aussi amener à ce que les analyseurs *acceptent* de telles chaînes ou choisissent la mauvaise analyse pour une phrase licite. En bref, les problèmes de sous-génération ou de surgénération peuvent arriver en analyse comme en génération. Ici, nous nous concentrons sur la surgénération.

On peut identifier plusieurs causes à la surgénération. Le fait est aujourd'hui bien connu : la création de grammaire est une tâche très complexe. En particulier, il est facile d'oublier ou de mal formuler une contrainte et ainsi d'autoriser une combinaison illicite (donc indirectement une chaîne illicite). De plus, une grammaire computationnelle est un objet gigantesque et prévoir les interactions décrites même par une grammaire de taille moyenne est très difficile pour ne pas dire impossible.

Dans notre cas, la grammaire est compilée à partir d'une spécification plus abstraite, une métagrammaire. Ceci nous aide à parvenir rapidement à une bonne couverture (on évite la sous-génération) ; cependant, avec un haut niveau d'abstraction, on court le risque d'autoriser plus de structures élémentaires que ce qu'on souhaitait. Les arbres élémentaires dans XMG sont construits en combinant des classes. Parfois, les classes se combinent de façon inattendue ou non-souhaitée, produisant ainsi des arbres qui ne devraient vraiment pas être dans la grammaire, des arbres qui vont aussi amener à la surgénération.

C'est pourquoi un réalisateur de surface qui produit toutes les chaînes associées à une sémantique donnée est un outil précieux : il permet de vérifier ces prédictions sur des exemples concrets. Un analyseur est un excellent outil pour détecter la sous-génération – il suffit de lui donner une phrase et d'observer ce qui se produit – mais comme le montre [Boguraev *et al.*, 1988] , il est beaucoup moins pratique pour détecter la surgénération, parce qu'on se retrouve alors à devoir inventer des phrases agrammaticales pour le tester. Avoir un générateur évite de devoir imaginer de telles phrases.

F-7.1 Débuggage de grammaire

La grammaire que nous avons à débugger est SemFRaG, la grammaire française de type FB-LTAG décrite dans le chapitre 4. Nous avons observé avec SemFRaG que les phrases fausses pour une sémantique d'entrée présentent toutes le même type d'erreur. Ceci nous a amené à penser que c'est un petit nombre de failles qui provoque un grand nombre d'erreurs et qu'observer systématiquement ce que les phrases fausses avaient en commun nous révélerait ces failles. L'idée qui sous-tend notre correction est la suivante : nous allons

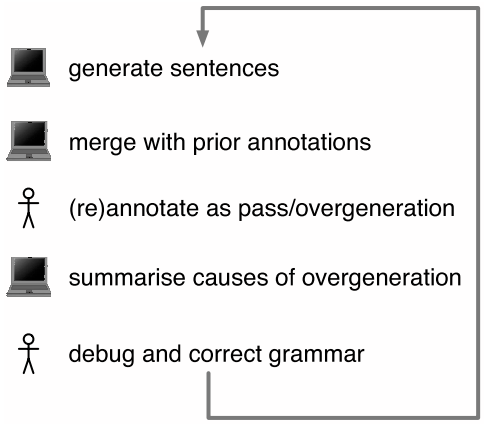


FIG. 11: Infrastructure de test

d'abord annoter (manuellement) la sortie du réalisateur de surface comme étant fausse ou non, et ensuite nous allons utiliser les données annotées pour faire ressortir automatiquement les items (e.g., les arbres élémentaires ou les propriétés d'arbre) qui apparaissent systématiquement dans les cas de surgénération. Plus précisément, la procédure qu'on définit pour réduire la surgénération peut être schématisée de la façon suivante :

1. La réalisation de surface est appliquée à une suite de tests graduelle de sémantique d'entrée et produit un historique détaillé des dérivations associées à chaque entrée de la suite de tests.
2. Les sorties obtenues par l'historique sont classées (à la main) dans deux catégories de phrases qu'on appelle : PASS (les phrases qu'on accepte) et OVERGENERATION (celles qui ne devraient pas être générées, parce qu'elles n'appartiennent pas à la langue ou ne peuvent pas être associées à la sémantique d'entrée).
3. La sortie annotée est utilisée pour produire automatiquement un résumé appelé *liste de suspects* ; ce résumé contient la liste des arbres TAG ou des étapes de la dérivation qui semblent causer la surgénération ; on les reconnaît parce qu'ils apparaissent seulement dans des cas de surgénération.
4. La grammaire est corrigée et on la ré-exécute sur les données.
5. Les résultats de la dérivation sont comparés avec les précédents et toutes les formes de divergence entre eux (phrases générées en plus ou en moins) sont signalées.

D'une certaine manière, cette approche consistant à produire une sortie et à corriger la grammaire à l'aide d'un historique des dérivations doit être déjà largement répandue en génération. Notre apport porte sur les trois points suivants :

– Le caractère systématique et incrémentale de notre approche ;

– Le haut niveau d'automatisation qui augmente notre rendement en focalisant l'attention des utilisateurs humains sur la correction de la grammaire plus que sur les détails inutiles ;
– La production du résumé des opérations permettant d'identifier facilement la source de l'erreur.

Nous détaillons maintenant ces trois points :

F-7.2 Une approche incrémentale

Les premières expériences avec SemFRaG ont montré que la grammaire surgénère énormément à la fois parce qu'elle a été développée au départ pour l'analyse, et aussi parce qu'elle est compilée à partir d'une spécification abstraite. En effet, pour certaines entrées, le réalisateur produit plus de 4 000 paraphrases, dont une grande proportion relève de la surgénération. Plus généralement, le nombre de sorties pour une entrée donnée varie de 0 à 4 908 avec une moyenne de 201 sorties par entrée (la médiane est de 25).

Pour éviter d'avoir à annoter manuellement un grand nombre de données, nous nous sommes fondé sur une suite de tests graduelle (que nous décrivons au chapitre 6) et nous avons traité les données en commençant par les plus simples pour terminer par les plus complexes. Ceci signifie que nous avons réduit la surgénération de façon incrémentale pour chaque cas, diminuant ainsi le nombre de sorties à annoter dans le test suivant.

Nous nous sommes contraint à travailler selon une méthode incrémentale très stricte avec l'aide d'une infrastructure de tests alternant les annotations manuelles et la production de sorties générées automatiquement.

Trois points méritent d'être soulignés. Tout d'abord, la liste de suspects est produite automatiquement grâce à l'annotation de l'historique des dérivations. Ainsi, en dehors de l'annotation de l'historique des dérivations qui est manuelle, l'identification des suspects est totalement automatisée. Ensuite, un test de régression est utilisé pour vérifier que les corrections faites à la grammaire n'affectent pas sa couverture (autrement dit, on vérifie que toutes les phrases appartenant à la catégorie pass sont toujours produites). Enfin, l'infrastructure de test procure un environnement confortable pour le linguiste permettant de visualiser, de modifier et d'exécuter la grammaire sur les entrées qu'il a étudiées.

L'historique des dérivations (étapes 1 et 3)

L'historique des dérivation produit par GenI contient des informations détaillées sur chacune des dérivations associées à une entrée donnée. Plus précisément, pour chaque chaîne générée, l'historique des dérivations montre l'arbre de dérivation associé et la famille d'arbre, l'identifiant d'arbre et les propriétés d'arbre associées à chaque arbre élémentaire composant cet arbre de dérivation.

L'historique des dérivations (figure 12) nous montre d'où proviennent les phrases, certes, mais ce qui nous manque cruellement est de savoir si ces phrases sont correctes ou non. Pour en extraire l'information nous montrant plus directement les causes probables de la surgénéraion, nous annotons manuellement cet historique, remplaçant pour chaque phrase l'étiquette `output` par l'étiquette `pass` si la phrase et correcte ou par l'étiquette `overgeneration` si elle ne l'est pas (nous avons été relativement tolérant dans nos annotations, en ce sens que nous avons marqué des phrases comme étant correctes lorsqu'elles étaient en

```
output: c'est paul demander jean qu'il vient
venir:Tn0V:n5 <-(a)- demander:Tn0Vs1int
venir:Tn0V:n4 <-(s)- paul:Tpropername
demander:Tn0Vs1int:n3 <-(s)- jean:Tpropername

output: c'est paul que demander jean il vient
venir:Tn0V:n8 <-(a)- demander:Tn0Vs1int
venir:Tn0V:n4 <-(s)- paul:Tpropername
demander:Tn0Vs1int:n3 <-(s)- jean:Tpropername

demander Tn0Vs1int-6199
  CanonicalSententialObjectInterrogativeFiniteWithoutComplementizer
  InvertedNominalSubject SententialInterrogative
venir Tn0V-6686
  CleftObject ImpersonalSubject NonInvertedNominalSubject
  activeVerbMorphology
jean Tpropername-2472
paul Tpropername-2472
========================================================================
output: c'est jean qui demande il vient paul
demander:Tn0Vs1int:n10 <-(s)- venir:Tn0V
demander:Tn0Vs1int:n4 <-(s)- jean:Tpropername
venir:Tn0V:n4 <-(s)- paul:Tpropername

demander Tn0Vs1int-6182
  CanonicalSententialObjectInterrogativeFiniteWithoutComplementizer
  CleftSubject NonInvertedNominalSubject SententialInterrogative
venir Tn0V-6678
  CanonicalObject ImpersonalSubject NonInvertedNominalSubject
jean Tpropername-2472
paul Tpropername-2472
```

FIG. 12: Un historique de dérivation

fait douteuses). Ce travail peut sembler fastidieux, mais il est en réalité plus tolérable qu'on pourrait le penser. En effet, le processus d'annotation est morcelé grâce à la nature incrémentale de notre approche, ce qui le rend bien plus gérable. Chaque test ne demande qu'un petit nombre de phrases soit annoté, et de plus, à chaque itération de notre infrastructure de test, le réel effort intellectuel réside dans le fait de devoir trouver comment corriger la grammaire. Comparées à cela, les quelques minutes passées à annoter une centaine de phrases sont négligeables.

Ceci dit, ce n'est pas en se plongeant dans ces historiques que nous allons découvrir les bugs de notre grammaire. En effet, ces historiques sont aussi longs que redondants. L'exemple ci-dessus montre trois phrases et fait une demi page ! La plus grande utilité de ces historiques, selon nous, réside dans le fait qu'ils

```
<suspects-report> ::= <test-case>*
<test-case> ::= (<lemma-item> <EOL>)*
                (<derivation-item> <EOL>)*

<lemma-item> ::= <lemma> <EOL>
                 <family-item> <EOL>
                 (<tree-item> <EOL>)*
<lemma>       ::= <string>
<family-item> ::= <tree-family> "(all)"? <tree-property>*
<tree-item>   ::= " [" <tree-number> "]" <tree-property>*

<derivation-item> ::= <tree-1> <arrow> <tree-2>
<tree-1>  ::= <tree-id> ":" <node-number>
<tree-2>  ::= <tree-id>
<arrow>   ::= "<-(" <op> ")-"
<op>      ::= "s" | "a"

<tree-id>       ::= <lemma> ':' <tree-family> '-' <tree-number>
<tree-family>   ::= <identifier>
<tree-number>   ::= <number>
<tree-property> ::= <identifier>
<node-number>   ::= <number>
```

FIG. 13: EBNF pour les listes de suspects

servent à produire les listes de suspects.

Les listes de suspects (étape 4)

Pour faire un usage complet de l'historique des dérivations, nous devons résumer et synthétiser l'information qu'il contient. Nous le faisons automatiquement en utilisant un script qui lit le fichier de dérivation annoté et produit une liste de suspects bien plus courte. Cette liste identifie des « suspects », c'est-à-dire les causes probables de la surgénération. Il ne se contente pas de résumer l'historique des dérivations, mais il en extrait l'information importante. Pour chaque sémantique d'entrée, la liste présente les arbres, les ensembles d'arbres ou les items de dérivation qui apparaissent *seulement* dans les cas de surgénération, c'est-à-dire, les chaînes associées avec cette entrée et qui ont été marquées avec l'étiquette OVERGENERATION.

La figure 13 montre l'allure d'une liste de suspects sous forme de grammaire EBNF. Comme nous pouvons le voir, la liste nous montre les sources de la surgénération pour chaque cas de la suite de tests. Les tests sont indépendants les uns des autres. Pour plus de simplicité, nous dirons que quand une chose (par exemple, une propriété d'arbre), apparaît systématiquement dans les phrases de surgénération, elle est «mauvaise». La liste donne donc pour chaque cas :

- les mauvaises propriétés d'arbres,
- les mauvaises dérivations,
- les mauvais arbres élémentaires,
- les mauvaises familles.

Les éléments cités se retrouvent dans la liste des suspects groupés sous deux items différents que nous décrivons maintenant : l'item lemme et l'item dérivation.

Item lemme Chaque item lemme (lemma item) consiste en une lemme et un ensemble d'arbres élémentaires qui possèdent des mauvaises propriétés d'arbres. Pour simplifier la présentation, nous supposons qu'un lemme est associé à seulement une famille d'arbres. (Les réelles listes de suspects acceptent et traitent plus d'une famille par lemme ; chaque famille distincte pour le lemme produit un item lemme distinct). L'item lemme se divise en deux parties : l'information sur la famille complète, et l'information individuelle sur les arbres à l'intérieur de la famille. Si tous les arbres élémentaires associés à cette famille (i.e. ceux qui sont utilisés dans le processus de génération pour le cas considéré et associés à cette famille) sont mauvais, nous le signalons en affichant `(all)`. Ceci suggère à l'utilisateur que tous les problèmes concernant cette famille pourraient venir d'une même source. De plus, nous faisons la liste de toutes les mauvaises propriétés d'arbres qui apparaissent dans tous les mauvais arbres élémentaires de la famille. Cela n'est pas strictement nécessaire, mais évite des redondances dans les listes de suspects.

Les mauvaises propriétés d'arbres restantes sont listées à côté de l'arbre élémentaire dans lequel elles apparaissent. Par exemple une ligne de la forme :

```
[649] CanonicalGenitive dePassive
```

indique que les propriétés d'arbre `CanonicalGenitive` et `dePassive` sont mauvaises et apparaissent dans l'arbre élémentaire numéro 649.

Nous affichons seulement le numéro de l'arbre parce que nous référons à des arbres de la même famille (par convention, les arbres produits par XMG sont associés à un numéro d'arbre). Notons que nous pouvons parfois afficher un numéro d'arbre sans propriétés d'arbre. Ceci arrive quand les mauvaises propriétés d'arbres associées à cet arbre sont communes à toute la famille et ont déjà été affichées.

Item dérivation Un item dérivation ($t_1^n \xleftarrow{Op} t_2$) consiste en un arc dans un arbre de dérivation TAG. Il indique que l'arbre t_2 a été inséré dans l'arbre élémentaire t_1 au nœud n grâce à l'opération TAG Op (qui sera soit une substitution soit une adjonction). Seulement les mauvais items dérivation sont donnés dans les listes de suspects. Ce qui ressort généralement de l'item dérivation c'est que l'un des deux arbres n'est pas assez contraint et qu'il est responsable d'une adjonction interdite.

Exemple: "Jean dit accepter/*C'est par Jean qui accepte qu'être dit". Nous montrons ici un exemple de fonctionnement de la liste de suspects. L'exemple ci-dessous indique que toutes les dérivations impliquant un arbre de type n0vn1 ancré par *dire* amènent à une surgénération et qu'il existe 6 arbres

de ce genre (les arbres 699 . . . 750). De plus, les propriétés d'arbre indiquent que tous ces arbres partagent les propriétés d'arbre `InfinitiveSubject` et `Passive`. L'examen des données montre que ces arbres se combinent avec une forme finie de "accepter" pour nous amener à une chaîne totalement agrammaticale comme "c'est par Jean qui accepte qu'être dire" au lieu de, par exemple, "Jean dit accepter".

Pour résumer, cet exemple indique que la grammaire n'est pas suffisamment contrainte pour bloquer la combinaison de la forme passive de l'infinitif des arbres n0Vn1 ancrés par "dire" avec des arbres associés par la grammaire à "accepter".

```
input t90
Lemma: dire
Tn0Vn1 (all) - InfinitiveSubject Passive
  [699] CanonicalCAgent Passive
  [746] CanonicalGenitive dePassive
  [702] CleftCAgentOne Passive
  [752] CleftDont dePassive
  [751] CleftGenitiveOne dePassive
  [750] RelativeGenitive dePassive
```

F-7.3 Le rapport de progression (étapes 5 et 2)

Utiliser les listes de suspects aide le linguiste à localiser les erreurs dans la grammaire qui conduisent à la surgénération. Pourtant, encore maintenant, toutes nos corrections de la grammaire n'ont pas le résultat attendu. Parfois, elles n'ont pas d'effet sur les erreurs et parfois elles en introduisent de nouvelles. Une partie de notre protocole consiste alors à exécuter l'infrastructure de test sur le même cas jusqu'à ce que toutes les phrases soient marquées PASS. Chaque fois que nous exécutons le réalisateur de surface, GENI produit un nouvel historique de dérivation non annoté. On utilise un petit script pour importer les annotations du précédent historique dans le nouveau. Le script produit aussi un rapport de progression qui permet à l'utilisateur de savoir au premier coup d'œil si ses modifications ont eu l'effet attendu.

```
New output?
  jean dit c'est l'homme volontaire qui part

Oops! We lost these passes:
  jean dit l'homme volontaire part

Hooray! no longer overgenerates:
  dit part l'homme volontaire jean
  dit jean part l'homme volontaire
```

F-7.4 Evaluation et resultats

Nous avons utilisé l'infrastructure de test pendant une période d'une semaine, approximativement pendant 12 heures d'affilée. Pendant cette période, nous avons effectué dix itérations de l'infrastructure de test et fait 13 modifications (30 lignes) dans la grammaire. Dans ce processus de révision de la

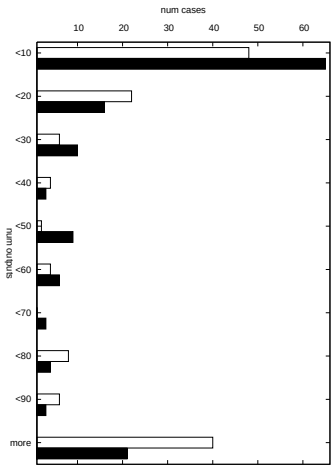


FIG. 14: Surgénération avant et après

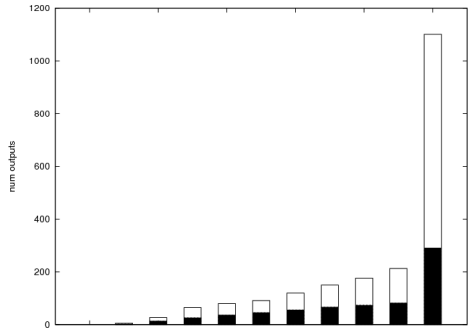


FIG. 15: Reductions de la surgénération

grammaire, nous avons étudié 40 cas (moins d'un tiers de la suite de tests) et annoté manuellement 1389 sorties de jugements de type « pass » ou « over-generation ». Sur la totalité des 140 cas de la suite de tests, la grammaire a produit 28 167 sorties (4 908 pour le cas le pire, et 201 en moyenne, et 25 de médiane). La grammaire révisée progresse de 70%. Elle produit 8434 phrases (201 pour le cas le pire, 60 de moyenne, et 12 de médiane). Ces nombres sont présentés dans le tableau ci-dessous pour plus de clarté :

	total	max	moyenne	médiane
avant	28167	4908	201	25
après	8434	710	60	12

Les réductions varient de 42% dans notre cas le plus faible à 74% dans le cas le plus important. On peut voir ces réductions dans la figure 15, où nous

nous concentrons seulement sur le nombre de sorties qui sont produites par chaque cas du test (sans regarder leur distribution dans la suite). Ces cas sont triés et groupés en fonction du nombre de sorties qu'ils produisent dans la nouvelle grammaire. Nous voyons dans le graphique combien de phrases sont produites en moyenne pour chaque groupe de tests, avant et après la correction, et plus particulièrement nous voyons que la différence entre les sorties est plus importante dans les cas les plus gros. L'idée centrale ici est que les sources de la surgénération vont se combiner et qu'éliminer ces sources peut avoir un impact important.

Par ailleurs, il est très bien de limiter la surgénération, mais seulement tant qu'on ne supprime pas des phrases linguistiquement valides au passage. La suite de tests a été produite semi-automatiquement, en analysant des phrases et en choisissant à la main les représentations sémantiques valides proposées en sortie. Pour plus de sécurité nous avons ré-analysé les phrases d'origine avec la nouvelle grammaire et avons réussi pour 136 des 140 phrases, soit pour 4 de moins qu'avec la grammaire d'origine. La différence était due à une contrainte trop restrictive et a été corrigée facilement.

La collecte des erreurs

Regardons maintenant les types d'erreur produisant la surgénération que nous avons trouvés.

Contraintes manquantes Ce n'est pas vraiment surprenant, mais la source principale de la surgénération était le manque de contraintes suffisantes pour bloquer les combinaisons d'arbres illicites. Par exemple, la grammaire générait la chaîne "devoir c'est Jean qui part" (au lieu de "c'est Jean qui doit partir") parce que l'arbre pour "devoir" n'était pas suffisamment contraint pour bloquer l'adjonction du nœud VP des phrases clivées. Dans de tels cas, nous avons éliminé la surgénération en ajoutant les contraintes nécessaires. Ces contraintes étaient les suivantes : CEST $=-$ sur le nœud pied de l'arbre "devoir" et CEST $=+$ sur le nœud VP de l'arbre de la forme clivée pour "partir".

Contraintes incomplètes et percolation de traits incorrecte Dans certains cas, nous avons trouvé que les contraintes étaient partiellement codées dans la grammaire, en ce qu'elles étaient données dans l'un des arbres de la combinaison, mais mal données voire pas données du tout dans tous les autres. Ainsi, par exemple, l'arbre des adjectifs était correctement contraint pour adjoindre à DET $=-$ les arbres N, mais la contrainte correspondante DET $=+$ sur le nœud racine du déterminant était manquante. Dans les autres cas, le trait était présent mais incorrectement percolé. Dans les deux cas, l'implémentation partielle de la contrainte empêchait un conflit dans l'unification, et donc provoquait une combinaison d'arbre qui n'aurait pas dû avoir lieu.

Arbres élémentaires illicites Un troisième type d'erreur était lié au fait que la grammaire a été produite semi-automatiquement à partir d'une description de grammaire abstraite. Dans certains cas, le linguiste n'a pas réussi à prévoir correctement les implications de sa description et à voir que l'arbre élémentaire ainsi produit par le compilateur était incorrect. Par exemple, on a dû

introduire une contrainte additionnelle dans la métagrammaire pour éliminer la formation d'arbres décrivant un verbe transitif avec un sujet impersonnel (en français, les verbes transitifs ne peuvent pas prendre de sujet impersonnel : en effet, la phrase « Des hommes aiment Marie » ne peut pas être paraphrasée par « Il aime Marie des hommes »).

Sémantique incorrecte Un type d'erreur plus complexe à traiter rassemble les cas où la sémantique n'est pas assez contrainte et autorise par conséquent des combinaisons illicites. Par exemple, dans la forme impérative, la grammaire échoue à contraindre que le premier argument soit YOU i.e., ait pour dénotation « l'auditeur ». Du coup, l'entrée permettant de produire des phrases comme "Jean demande si Paul part" génère aussi des sorties fausses comme "demande à Jean si Paul part". Clairement, dans de tels cas, c'est la sémantique associée par la métagrammaire à l'arbre élémentaire qui doit être modifiée.

Exceptions lexicales On sait que les généralisations grammaticales sont souvent sujettes à des exceptions lexicales. Par exemple, les verbes transitifs sont supposés pouvoir se passiver, mais pas les verbes de mesure comme "to weigh" (peser) qui sont pourtant transitifs. Comme on le fait habituellement en TAG, dans GenI, de telles exceptions sont notées dans le lexique de façon à bloquer la sélection de certains arbres (dans ce cas précis, tous les arbres passifs) pour les items lexicaux faisant partie des exceptions (ici, les verbes de mesure). Il arrive donc que la surgénération provienne parfois simplement de l'insuffisance des informations lexicales.

F-8 Conclusion

Le point de mire de cette thèse est la construction d'un réalisateur de surface réutilisable, réversible, efficace, et adapté au contexte. La réutilisabilité vient de l'utilisation d'une ressource et d'un langage d'entrée indépendants du domaine. La réversibilité vient du type de grammaire utilisé, dans notre cas, une FB-LTAG augmentée par L_U, qui a déjà prouvé son efficacité en analyse. L'efficacité et l'adaptation au contexte sont les caractéristiques sur lesquelles notre travail apporte des éléments nouveaux. Chacune de nos contributions constitue une extension du réalisateur de surface GenI. Alors que les extensions sont mutuellement dépendantes, et très différentes l'une de l'autre, elles sont liées par les thèmes communs de l'ambiguïté et du déterminisme.

Filtrage par polarité La première extension concerne le problème de l'ambiguïté lexicale, autrement dit le problème posé par le fait d'avoir plus d'un item lexical correspondant à chaque littéral de la sémantique. Cette ambiguïté doit rester une caractéristique de la grammaire parce qu'elle lui permet d'exprimer la même chose de plusieurs façons différentes. Pour conserver et gérer cette caractéristique, on doit s'assurer que le réalisateur de surface n'est pas enlisé dans des interactions entre des ensembles d'items lexicaux qui ne sont finalement pas faits pour aller ensemble. La solution consiste à polariser la grammaire pour que les items lexicaux « sachent » quelles ressources ils fournissent, et quels besoins ils ont. On ajoute ensuite une étape de filtrage pour éliminer tous les ensembles d'items avec des polarités non-neutres, c'est-à-dire avec des besoins et des ressources qui ne se correspondent pas. Ajouter ces filtres rend l'utilisation d'une grammaire ambiguë plus pratique pour la réalisation de surface.

Sélection des paraphrases Comme nous avons un réalisateur qui peut effectivement *utiliser* l'ambiguïté lexicale nous pouvons générer toutes les paraphrases pour une entrée donnée. Alors que les paraphrases sont (en théorie) toutes grammaticalement correctes, elles ne sont pas toutes adaptées à tous les contextes. Notre objectif suivant a donc été de rendre notre réalisateur plus proche de ceux qu'on dit orientés-génération, comme KPML ou SURGE, en augmentant sa capacité à s'adapter au contexte. Idéalement, on devrait être capable de générer à partir du même type de traits fonctionnels que ceux qui sont acceptés par de tels réalisateurs. Ce n'est pas encore possible avec GenI, mais nous avons fait un pas dans cette direction, en ajoutant la possibilité de présélectionner nos sorties à partir d'un ensemble de critères linguistiques appelés propriétés d'arbre. Ces propriétés sont utilisées pour restreindre la sélection lexicale. Elles apparaissent comme des annotations dans la sémantique d'entrée et dans la grammaire. Ajouter des propriétés d'arbre à une grammaire est une conséquence directe de la compilation de la grammaire à partir d'un formalisme métagrammatical plus abstrait. Ajouter ces annotation dans la sémantique d'entrée est une question que nous réservons pour des recherches futures.

Réduire la surgénération Penser que toute les sorties du générateur pourraient être grammaticales serait être optimiste. Les grammaires réelles surgé-

nèrent, et celles qui sont générées à partir de représentations abstraites surgénèrent même beaucoup. Ceci peut être problématique pour deux raisons (i) parce que l'ambiguïté non souhaitée rend la réalisation moins efficace, et (ii) parce que le problème de choisir une sortie appropriée au contexte est mélangé avec celui de choisir une sortie grammaticalement correcte. Cependant, la caractéristique qui rend nos grammaires sujettes à la surgénération (compilation à partir d'une représentation factorisée ⇒ structures syntaxiques non souhaitées), peut aussi être utilisée pour réduire la surgénération. L'idée est de ré-utiliser les propriétés d'arbres qu'on a employées pour la sélection des paraphrases. On génère toutes les chaînes qui sont associées avec la sémantique d'entrée, on isole les cas de surgénération, on récupère leurs propriétés d'arbre, et on identifie l'origine des propriétés d'arbre erronées dans la métagrammaire.

F-8.1 Perspectives

Dans les chapitres 5–7, nous suggérons des améliorations pour les techniques de filtrage, de sélection et de débuggage de la grammaire. En plus de ces améliorations, voici des perspectives à explorer en utilisant le réalisateur dans sa totalité.

Génération morphologique

Le manque de formes fléchies est un obstacle à l'utilisation du réalisateur dans de réelles applications, et particulièrement pour une langue comme le français. Ajouter de la génération morphologique nous permettrait de générer des phrases comme "Elle est aimée du garçon timide" au lieu de "Il être aimer de le garçon timide".

Planification de phrase

Il serait peut être utile d'intégrer une tâche de planification de phrase dans l'esprit de SPUD et INDIGEN.

Passage à l'échelle

Il reste encore du travail à faire pour rendre le processus de réalisation plus efficace. Deux éléments qui valent la peine d'être explorés sont l'amélioration du générateur tabulaire et du filtre de polarité.

Devenir plus orienté-génération

L'introduction de propriétés d'arbre peut nous aider à combler le fossé entre les réalisateurs réversibles comme GENI et ceux qui sont orientés-génération comme KPML et FUF. La prochaine question à se poser sera de savoir d'où les propriétés d'arbres doivent provenir. Une idée pourrait être de les collecter à partir d'un réseau de systèmes comme on le fait en linguistique systémique fonctionnelle.

Mettre GenI au travail

GenI sera mis en œuvre comme un module de génération effectif. Alexandre Denis l'utilise à l'intérieur d'un système de dialogue en français pour obtenir des retours sur la compréhension par le système des énoncés des utilisateurs. Par ailleurs, Luciana Benotti l'utilise dans un système de dialogue multilingue français/anglais pour des recherches sur l'accomodation de la présupposition [Beaver and Zeevat, 2007]. La génération n'est pas la problématique centrale de ces projets de recherches, mais nous espérons que le réalisateur va simplement... « travailler pour eux ».

Evaluation comparative

Il serait utile de voir le réalisateur de surface intégré dans un processus d'évaluation, et pourquoi pas en comparaison avec d'autres réalisateurs de surface. Cependant, avant de mettre tout cela en œuvre, nous aurions à définir exactement ce que nous souhaitons comparer et pourquoi.

Surface realisation: ambiguity and determinism

Eric Kow

2007-11-29

version 1.0.1

Contents

Remerciements (Acknowledgements) **i**

Contents **iii**

Ambiguïté et détérminisme **iv**

Introduction **2**

I Background **5**

1 Realisation algorithms **7**
1.1 Syntactic tree traversal 8
1.2 Search 16
1.3 Sharing intermediate results 17
1.4 Summary of the main issues 38

2 Flat semantics with holes **41**
2.1 Flat semantics 41
2.2 Logical-form equivalence 43
2.3 The case for a flat semantics 49
2.4 Intersective modifiers 51
2.5 Summary of flat semantics 57

3 Tree Adjoining Grammar **59**
3.1 From TAG to FB-LTAG 59
3.2 TAG Derivations 64
3.3 FB-LTAG augmented with L_U flat semantics 65
3.4 Generation with TAG 73

4 GenI and SemFraG **77**
4.1 GenI 77
4.2 SemFraG 84
4.3 Related NLG systems for TAG 86

II Contributions **89**

5 Polarity filtering **91**
5.1 Polarised intuitions 91

5.2 Building polarity automata 96
5.3 Chart generation with polarity automata 108
5.4 Extensions 108
5.5 Evaluation 118
5.6 Related work in lexical disambiguation 121

6 Paraphrase selection **127**
6.1 Contextual appropriateness 128
6.2 Selection mechanism 129
6.3 Evaluation 132
6.4 Possible extensions 135
6.5 Related work in paraphrase selection 138

7 Reducing overgeneration **147**
7.1 Overgeneration 147
7.2 Grammar debugging 148
7.3 An incremental approach 149
7.4 Evaluation and results 155
7.5 Possible extensions 157
7.6 Related work 159

8 Conclusion **165**
8.1 Summary 165
8.2 Future work 166
8.3 Putting GenI to work 167

A SemFraG families **172**

B Tree properties from SemFRaG **175**

C Deductive realisation and unification **181**
C.1 Kay1996 with unification 181
C.2 GenI with unification 182

D GenI pseudocode **183**
D.1 Lexical selection 183
D.2 Realisation proper 184
D.3 Helper functions 185

Bibliography **187**

Introduction

The fundamental goal of natural language generation (NLG) is to translate a communicative goal into natural language. A surface realiser is a small part of a natural language generator. Given a grammar and a meaning representation (often a logical form), the job of a surface realiser is produce the strings which are associated by the grammar with the semantics. Surface realisation is one of the most concrete and therefore one of the more straightforward tasks in generation. Indeed, much of generation research in the past has been about surface realisation and has borne a series of high-quality, reusable surface realisers like RealPro, fuf and kpml, which have all been used to build real world NLG systems.

But while the basics of the task are well understood, much remains to be done. One interesting issue is how a surface realiser should deal with natural language paraphrasing. Paraphrasing is possible because of the simple fact that there is often more than one way to say the same thing. This brings great variety to language, and it allows us to communicate subtle nuances in meaning. It is also a combinatorial nightmare. In this thesis, we are interested in the question of how a surface realiser should deal with paraphrasing.

Roadmap of the thesis

We follow a standard two part format: first some background information, and second our contributions. The background begins in Chapter 1 with a survey of the common surface realisation technologies: basically, head-driven traversal and chart generation. The next two chapters focus on our core assumptions: a flat semantics (namely the language L_U) and a Feature Based Lexicalised Tree Adjoining Grammar (FB-LTAG). Chapter 2 presents the key issues that led to widespread adoption of flat semantics, and provides some details on the language L_U. In Chapter 3, we turn to the question of syntax, what the FB-LTAG formalism is and how exactly we combine it with L_U for semantic construction. These three chapters provide the three major pieces of a surface realiser: an algorithm, an input semantics and a grammar. Chapter 4 assembles all of these pieces into the realiser, GenI.

One defining characteristic of GenI is that it was designed to make the most of a paraphrasing grammar. A paraphrasing grammar is one which offers more than one way to say the same thing, for example, "Ernest loves Charmaine" and "Charmaine is loved by Ernest", and moreover, associates all of them to the same meaning representation. The usual assumption in surface realisation work is that we need only return one result; after all, that is all we will need for a generation task. GenI turns this assumption on its head. It starts from

the perspective that one might actually want to return *all* of the results that the grammar has to offer. We explore this idea from three different angles, basically (i) doing it (ii) doing the opposite and (iii) making it useful. The first angle is that if we are going to return all results, we had better be efficient about it. Chapter 5 shows how we can use "polarity filtering" to cut down the search space of the realiser. The second angle is about making GenI behave more like the one-result realisers whilst staying true to its paraphrasing spirit. Chapter 6 thus provides a way of enriching the input with a description of additional linguistic requirements, allowing the user to preselect the one best result. This feeds right into our third angle. The language of describing linguistic requirements can be flipped around by asking the realiser to annotate every paraphrase with the linguistic requirements it fulfils. In Chapter 7, we see how to use these annotations to pinpoint the various flaws in our grammar.

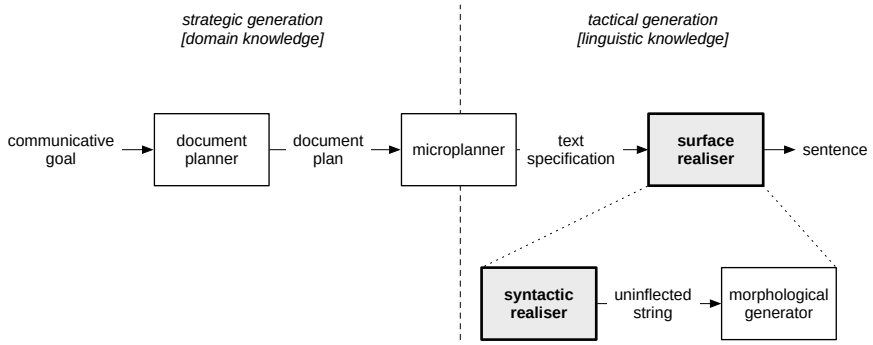
Part I

Background

Chapter 1

Realisation algorithms

Natural language generation is usually seen as consisting of a basic pipeline of tasks, shown in the figure below. 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. By the late 1990s, for example, 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.[1]



What we are particularly interested in are the syntactic aspects of realisation, basically, making sure that the words come out in the right order. We use the approach that reversible, phrase-structure based realisers have in common: build a syntactic tree which is associated by the grammar to the input semantics, and read the tree leaves to get an output string. Where the surface realisers vary is how they go about building this tree. There are three facets to the problem, which we now visit: traversal of the syntactic tree (Section 1.1), exploration of the search space (Section 1.2) and storing of intermediate results (Section 1.3).

[1] It is not a foregone conclusion that NLG requires a pipeline architecture; it is just that pipelines are simple and convenient, and they work well enough for the moment.

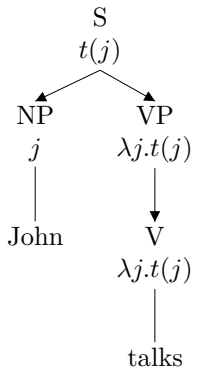


Figure 1.1: Top-down realisation of "John talks".

1.1 Syntactic tree traversal

We can see surface realisation as the process of "discovering" a syntactic tree that corresponds to an input semantics [Shieber *et al.*, 1990]. This can be approached using a top-down, bottom-up or even a mixed strategy. Each of these strategies has specific problems in practice: top-down algorithms are vulnerable to left-recursion, bottom-up ones require too many restrictions of the grammar formalism to be complete and are too non-deterministic to be useful in practice. The situation is considerably better for mixed strategies such as semantic head-driven generation [Shieber *et al.*, 1990], but as pointed out in [Gerdemann and Hinrichs, 1995] there also exist linguistically motivated grammars for which these algorithms fail to terminate. We now explore the three strategies.

About the notation In this chapter and in Chapter 2, we assume a unification grammar with a context free backbone. Unification variables will be written starting with a capital letter, as in Foo. The symbols λ and . should be treated as meaningless punctuation; they are introduced for mnemonic reasons.

1.1.1 Top-down realisation

We will start with a small example. As we can see, the grammar below associates the sentence "John talks" with the semantics *talks(john)*.

t1.	S(S)	$\rightarrow$ NP(X) VP($\lambda X.S$)
t2.	VP($\lambda X.S$)	$\rightarrow$ V($\lambda X.S$)
t3.	V($\lambda X.talk(X)$)	$\rightarrow$ talks
t4.	NP($john$)	$\rightarrow$ John

Suppose now that we give this semantics to a surface realiser. If it was using a top-down strategy, it would start from the root and work its way down

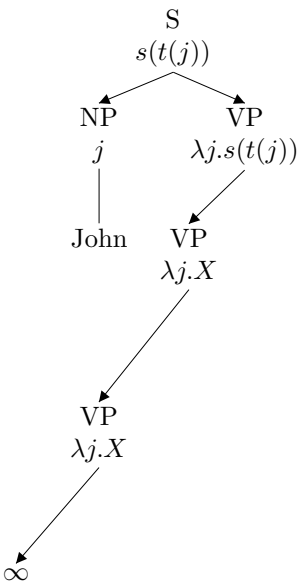


Figure 1.2: Unsuccessful top-down realisation of "John talks-slowly".

to the leaves. We can see this strategy at work in Figure 1.1. The semantic representations in this diagram are abbreviated (e.g. *t(j)* for *talk(john)*), and the arrows indicate the rough order of derivation.

The problem with top-down realisation is that left-recursive rules can send the algorithm into an infinite loop. For example, suppose we inserted the following rules into our grammar to account for adverbs:

t5. $\text{VP}(\lambda X.S) \rightarrow \text{VP}(\lambda X.V)\ \text{Adv}(\lambda V.S)$
t6. $\text{Adv}(\lambda V.\mathit{slowly}(V)) \rightarrow \text{slowly}$

Trying to realise *slowly(talk(john))* in a purely top-down fashion would be a bad idea because of the left-recursion inherent in rule `t5`. As we see in Figure 1.2, the realiser may attempt to process the left child of this rule by again selecting `t5`. It does so continuously because the `X` variable is unconstrained. If the realiser somehow "knew" to process the right child, that `VP` variable would have been constrained by `t6`. Or if the realiser had instead selected rule `t2` as the left child of `t5`, it would also have halted and yielded the sentence "John talks slowly".

Left-recursion is a well-known problem for NLP researchers. Some promising solutions had been found, solutions which consist either in automatically rewriting the left-recursion out of the grammar [Dymetman and Isabelle, 1988] or in changing the order in which nodes are expanded, starting with the nodes whose semantics is instantiated [Wedekind, 1988]. Indeed, these techniques would help for the case of "John talks slowly"; however, in the general case,

they are not adequate. As [Shieber *et al.*, 1990] points out, one could always find a linguistically plausible rule that is prone to "left"-recursion, for instance, this use of subcategorisation lists:[2]

t1b. $\text{VP}(\lambda X.S) \rightarrow \text{VP}(\lambda X.S, [])$
t2b. $\text{VP}(\lambda X.S, L) \rightarrow \text{V}(\lambda X.S, L)$
t3b. $\text{VP}(\lambda S, L) \rightarrow \text{VP}(\lambda X.Sem, X{:}L)\ \text{NP}(X)$

The VP rule above, `t3b`, generalises over verbs which accept a different number of complements. This rule accepts n complements if it accepts $n+1$ complements, or if there is a lexical item which also accepts n complements. Enforcing the number of complements is thus left to the lexical rules. For example, "teaches" takes a direct object, whereas "gives" takes both a direct object and an indirect object.

t4b. $\text{V}(\lambda S.talks(S), [])$ $\rightarrow$ talks
t5b. $\text{V}(\lambda D \lambda S.teach(S, D), [D])$ $\rightarrow$ teaches
t6b. $\text{V}(\lambda D \lambda I \lambda S.give(S, D, I), [I, D])$ $\rightarrow$ gives

Suppose we wanted to realise the semantics *slowly(teach(john,mary))*. With an improved top-down algorithm, where the first child to be selected is the one with an instantiated semantics, we would indeed avoid the left recursion on rule `t2`. On the other hand, we would still fall into infinite recursion whilst attempting to construct the subcategorisation list. We would just go on and on, adding uninstantiated variables into a never-ending list (Figure 1.3).

The subcategorisation example provides a useful insight into why the top-down approach is flawed. This infinite loop could easily have been avoided because information needed to constrain the realiser (the size of the subcategorisation list) is readily available in the lexicon. If for instance the realiser "knew" it was trying to realise "gives", it would have been forced to build a subcategorisation list of exactly size two and the whole issue of left recursion would not have crept up. What would considerably improve top-down realisation is some means of exploiting lexical information. Figure 1.4 shows how this semantics may be realised in an ideal world.

1.1.2 Bottom-up realisation

Bottom-up algorithms make much better use of lexical information; they build the syntactic tree up from the leaves to the root. Figure 1.5 shows what the bottom-up realisation of *slowly(talk(john))* might look like, using the same toy grammar from the previous section (repeated below for convenience).

b1. $\text{S}(S)$ $\rightarrow \text{NP}(X)\ \text{VP}(\lambda X.S)$
b2. $\text{VP}(\lambda X.S)$ $\rightarrow \text{V}(\lambda X.S)$
b3. $\text{VP}(\lambda X.S)$ $\rightarrow \text{VP}(\lambda X.V)\ \text{Adv}(\lambda V.S)$
b4. $\text{V}(\lambda X.talk(X))$ $\rightarrow$ talks
b5. $\text{NP}(john)$ $\rightarrow$ John
b6. $\text{Adv}(\lambda V.slowly(V))$ $\rightarrow$ slowly

The problem of infinite looping is gracefully avoided on `b3` because the VP semantics (V) is already instantiated by the time we try to use it.

[2] We can assume the standard notation for lists: [] is the empty list. If L is a list, then $X : L$ is a list. $[a, b, .., z]$ is shorthand for $a : b : \ldots : z : []$.

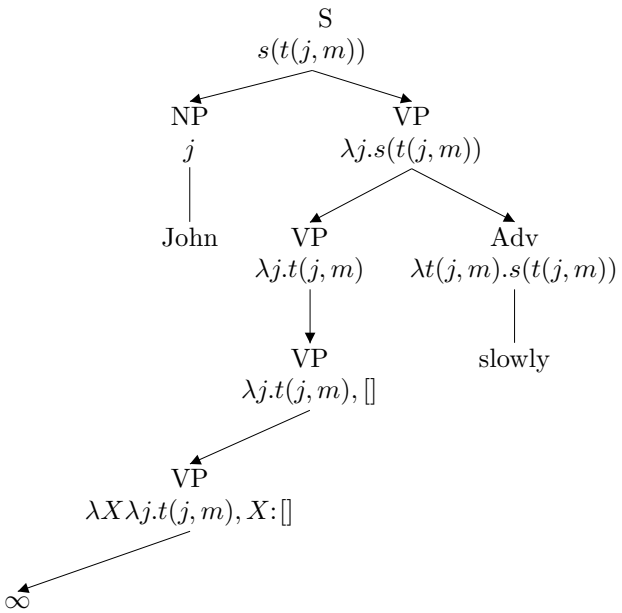


Figure 1.3: Infinite recursion on subcategorisation lists

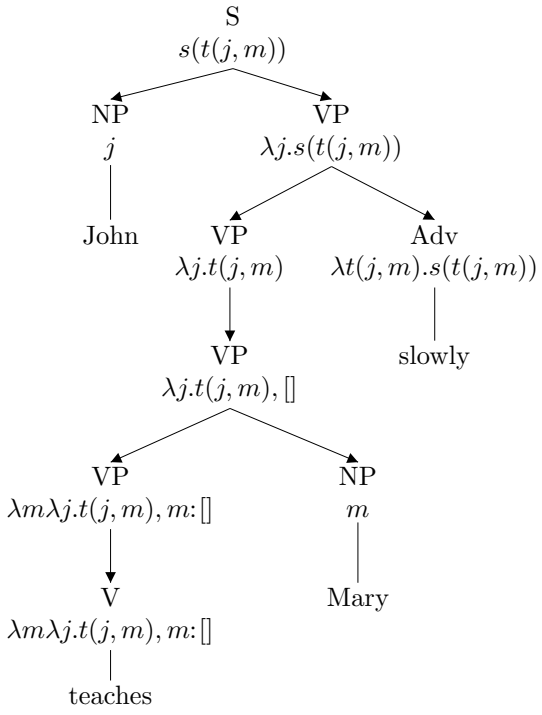


Figure 1.4: Subcategorisation on its best behaviour

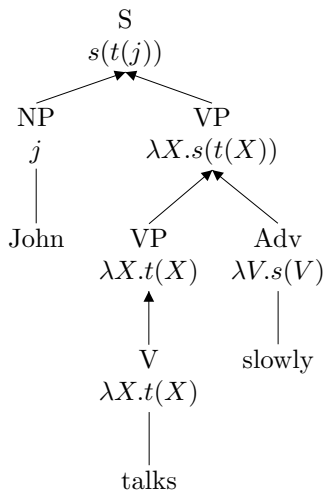


Figure 1.5: Bottom-up realisation of "John talks-slowly".

Semantic monotonicity

A commonly cited example of bottom-up realisation is [Shieber, 1988], which proposes a uniform architecture for parsing and generation. The uniform architecture uses an Earley-like algorithm, which is bottom-up in the sense that it is initialised with a list of lexical items rather than from the start symbol (see the clarification on Page 13). For parsing, the algorithm is constrained by string positions, but not for generation because string positions are meaningless when we do not yet know what the string is. Shieber claims that not using string positions causes the algorithm to no longer be goal-directed, that is, "many phrases are built that could not contribute in any way to a sentence with the appropriate meaning."

Consequently, he introduces an additional semantic filter for efficiency purposes. The filter retains only items whose semantics subsumes some part of the input semantics. For instance, given an input semantics *passionately(love(sonny, kait))*, an item with semantics *passionately(X)* would be admitted by the filter and so would *sonny*, but not *love(kait, sonny)*.

Completeness

Using this filter adds a requirement that the grammar be semantically monotonic, that is, the semantics of every child node of every rule must subsume the semantics of its parent node. The semantic monotonicity requirement is needed to guarantee completeness of the generation algorithm despite the semantic filter. Completeness here[3] is the property that any sentence the gram-

[3]Not to be confused with completeness as the sense of completeness/coherence from [Wedekind, 1988].

mar associates with the input semantics will be found by the generator.

Problem: idioms

As Shieber points out, the semantic monotonicity requirement is overly restrictive. For instance, it would not allow for grammar rules such as those in the fragment below:

b7. $\text{VP}(\lambda X \lambda Y.callup(X,Y)) \rightarrow \text{V}(\lambda X \lambda Y.callup(X,Y))\ \text{NP}(Y)\ \text{PP}(up)$
b8. $\text{V}(\lambda X \lambda Y, callup(X,Y)) \rightarrow \text{call}$
b9. $\text{PP}(up) \rightarrow \text{up}$

This fragment associates the sentence "The girl calls up the boy" with the semantics *callup(girl, boy)*. It cannot be used with Shieber's semantic filter because the semantics *up* does not subsume any part of the input semantics. More generally, semantic monotonicity is at odds with a natural treatment of idioms or multi-word expressions.

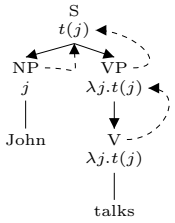
Non-termination

Note that completeness is not a guarantee of termination. For instance, `b10` is a perfectly (semantically) monotonic rule from that would provoke an infinite loop in a bottom-up algorithm [van Noord, 1989]:

b10. $\text{VP}(\lambda X.S) \rightarrow \text{to VP}(\lambda X.S)$

As van Noord points out, a reasonable grammar would also have syntactic information elsewhere to prevent a semantics like *like(kiss(john, mary))* from being associated with "John likes to to... to kiss Mary". A complete generator would find all sentences admitted by the grammar, whilst still falling into an infinite loop on some (ultimately ungrammatical) fragment.

An Earley remark



[Gerdemann, 1991] disputes the original observation by Shieber that the lack of string positions is responsible for the loss of goal-directedness. To see why, we should take a closer look at why we consider Shieber's uniform architecture to be bottom-up in the first place. The algorithm uses Earley-style traversal, which consists of two essential moves: top-down prediction and bottom-up completion. Prediction is the tentative instantiation of a child node in a grammar rule, and completion is the confirmation that it was successfully recognised. Earley traversal is a series of predictions, followed by a series of completions, a series of predictions and so forth until the full syntactic tree is built. This has an uncanny resemblance to top-down realisation, which makes it seem rather odd to say that Shieber is bottom-up.

The reason we can say this is precisely because the traversal includes both top-down and bottom-up elements. For context-free grammars, the top-down part is the most pertinent, whereas for unification grammars, it is mostly the bottom-up part that counts. And since [Shieber, 1988] is based on unification grammar, we say it is bottom-up. One can think of prediction as a sort of filter; no completion is possible unless a corresponding prediction was made to start the process. For context free grammars, this is a hard filter, which makes

Earley's algorithm equivalent to top-down traversal. Prediction in unification grammar is a soft filter; as we saw in Section 1.1.1, we do not always have all the information we need to fully instantiate a feature structure when we predict it, so we scrape by with a partly instantiated one. Only the "completed" feature structure is fully instantiated. To check if the completion was authorised by a prior prediction, we must verify that the predicted feature subsumes the completed one. The point here is that the more general the predicted features are (the less fully instantiated), the less useful the prediction filter is and the more bottom-up the algorithm becomes. There are two implications of this.

First, we must nuance our initial presentation, where we said that Shieber's algorithm is bottom-up because it is initialised from the set of lexical items. This is only half-true. Starting from lexical items does not have any effect by itself since completions have to be "authorised" by a prior prediction. Otherwise, they remain inert, waiting to be catalysed by the right predictions. With Earley-style traversal, starting from the lexical items is equivalent to starting from the single start symbol. Where the bottom-up nature of the algorithm really comes from is a set of overly lax prediction rules.

Second — this is the more important point — Shieber's semantic filter may not be necessary after all. To see this, it helps to consider why predictions may be overly lax in the first place. The killer feature behind Earley's algorithm is the use of tabulation to avoid non-termination (see Section 1.3.3.4). This only works for context free grammars, however. Tabulation or no, it is still possible for the prediction rule to fail to terminate with a unification grammar (for example, with a subcategorisation list). To get around this, [Shieber, 1985] proposed a notion of restrictors which eliminate the pathological non-terminating parts from feature structures before using them for prediction. In other words, we relax prediction in order to help it terminate. This is where [Gerdemann, 1991] comes in. The lack of string positions, he argues, is not the culprit behind this loss of goal-directedness. Rather, the problem at hand is the use of overly aggressive restrictors. "The algorithm, in fact, becomes more or less goal directed depending only on how much information is eliminated by the restrictor in the prediction step" taken from [Gerdemann and Hinrichs, 1995]. Gerdemann and Hinrichs go on to suggest that the semantic filter is not actually needed and can be replaced by a more careful choice of restrictors. This might make the question of semantic monotonicity moot. That said, the use of restrictors implies a better use of top-down information; the overall point stands that purely bottom-up traversal is not goal-directed enough.

1.1.3 Head-corner realisation

Top-down algorithms may fail to terminate if based on a left-recursive grammar. On the other hand, bottom-up algorithms may be too inefficient. The head-corner algorithms proposed in [van Noord, 1989] and [Shieber *et al.*, 1990] improve on this situation considerably. They eliminate the need for a semantic filter, and as an added bonus, avoid the non-determinism that comes from left-to-right evaluation. The algorithms can be seen as adaptations of left-corner parsing, except that instead of seeking the left corner of a phrase, they seek its semantic head. The semantic head of a rule is the daughter node that has the same semantics as its mother. Not all rules have a semantic head. Conversely, some rules even have multiple heads, but we will ignore them for the moment.

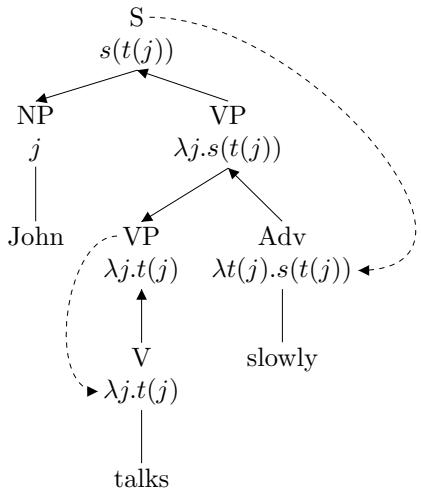


Figure 1.6: Semantic-head-driven generation of "John talks-slowly".

In the [Shieber *et al.*, 1990] variant of the algorithm, semantic-head-driven generation (SHDGA), the grammar is preprocessed and divided into chain rules (those that have a semantic head) and non-chain rules (those that do not). The processing begins at the start symbol of the grammar. Next, it finds a pivot node and recursively process its daughters. The pivot is a non-chain rule whose left hand side matches the current goal. Selecting a pivot serves a similar role as the scanning step in a left-corner parser; it establishes a location from which we begin climbing up the analysis tree. This climbing (or connecting) process is also recursive: we select a chain rule, unify its semantic head with the current node, process its other daughters and then climb up to yet another chain rule until the current node matches the goal. Figure 1.6 illustrates this process on the realisation of *slowly(talk(john))* using the same toy grammar from the previous two sections:

c1.	S(S)	$\rightarrow$ NP(X) VP($\lambda X.S$)
c2.	VP($\lambda X.S$)	$\rightarrow$ V($\lambda X.S$)
c3.	VP($\lambda X.S$)	$\rightarrow$ VP($\lambda X.V$) Adv($\lambda V.S$)
c4.	V($\lambda X.talk(X)$)	$\rightarrow$ talks
c5.	NP($john$)	$\rightarrow$ John
c6.	Adv($\lambda V.slowly(V)$)	$\rightarrow$ slowly

This algorithm does not require the kind of rule selection filter that Shieber's uniform architecture did. Therefore, it can be used on semantic non-monotonic grammars, which is clearly an improvement. This algorithm looks very much like top-down realisation with head-first traversal instead of left-to-right. But as we saw in Section 1.1.1, head-first selection cannot by itself guarantee termination on linguistically plausible grammars. This is where the bottom-up processing of SHDGA becomes useful; it avoids the non-termination problem by

exploiting lexical information. And since the bottom-up processing is only initiated from the top-down, the algorithm remains goal-directed. What makes this traversal strategy useful is the marriage of top-down and bottom-up traversal, and what makes this marriage possible is the use of head-first selection.

Non-termination

Unfortunately, SHDGA is also vulnerable to non-termination from plausible grammar rules. The problem here is that the top-down chain-rule processing phase may run into an infinite loop. A minimal example might be: [Shieber *et al.*, 1990]

$$\text{c7.} \quad \text{NBar(N)} \rightarrow \text{NBar(N) SBar(N)}$$

[Gerdemann, 1991] proposes an alternative head-driven algorithm to SHDGA. Instead of corner-based processing, he combines the Earley algorithm with head-first selection. This avoids some non-termination, through tabulation and careful management of restrictors, but as [Gerdemann and Hinrichs, 1995] show, the resulting algorithm has its own share of non-termination problems. Anyway, the point remains that head-first selection is inherently useful.

1.2 Search

Our presentation of tree traversal strategies has been largely focused on non-termination and goal-directness. When we say that an algorithm is goal-directed, we can think of it as saying that it avoids non-deterministic situations. For example, SHDGA avoids the non-determinism from uninstantiated semantics by using head-first selection. Natural language is inherently ambiguous, so it is clear that no matter what tree traversal we adopt, there is always going to be a certain amount of non-determinism to deal with. In this section, we touch on the strategies handling the non-deterministic choices we are forced to make.

1.2.1 Number of solutions

The first thing we must ask ourselves is how many solutions we want to return: just one? n solutions? all of them? If we want to return all solutions, the other aspects of the search problem become less crucial. They could still be worth exploring if we are also interested in ranking the solutions. Search also becomes relevant if we want to limit the number of results returned by the surface realiser, since we might as well return the best results we can.

1.2.2 Priority of choices

The search problem consists in choosing the best way to build a syntactic tree. Confusingly enough, the search space can also be viewed as a tree. The search tree represents the choices we make given an ambiguity in the grammar. Each node in the search tree corresponds to an expansion point in the syntactic tree; and each child node of the search tree represents an option that the grammar provides for performing this expansion. The more ambiguous the grammar, the

more ways we can perform this expansion and the more child nodes we have. This is where general search strategies come into play.

We could have a breadth-first search, which explores all the choices for a given expansion point before moving on their consequences. This involves keeping track of all possible syntactic structures at the same time. More commonly, we could have a depth-first search, which fully explores each choice (and its subchoices) and backtracks upon the first dead-end. Search strategies need not be blind either. For optimisation problems (where all solutions are not created equal), the search strategy can be crafted to favour choices that end up leading to "good" results. A simple example of this would be a greedy strategy, which always picks the locally best choice at each point. More sophisticated strategies are also possible.

1.2.3 Commitment and pruning

Another factor to consider is the degree of commitment we make to our choices. On one extreme, we make no commitment whatsoever, allowing for the entire search space to be explored (e.g. we allow for full backtracking of depth-first algorithms). On the other extreme, we could commit fully to our choices (e.g. no backtracking), which avoids a great deal of non-determinism, but also induces the risk of dead ends or suboptimal results. The two extremes represent different degrees of pruning, for which a wide range of strategies may also exist. For instance, [White, 2004] uses an "anytime searching" strategy, where highly improbable states are pruned away. This also induces some risk of dead ends or suboptimal states but is highly unlikely to do so. Another option still would be to use "safe" pruning methods which only cut away paths that we can guarantee will lead to a dead end. In this thesis, we only use "safe" pruning.

1.3 Sharing intermediate results

So far we have dealt with avoiding non-determinism (tree traversal) and making the best choices when it inevitably arises (search). But what happens if we make the wrong choices? A good search strategy is no guarantee the choices we make are always correct, so we need also a mechanism for recovering from error. One such mechanism would be backtracking. If our search strategy encounters a dead-end, we might simply go back to the most recent choice and try an alternative. But backtracking can be extremely wasteful: whenever the search algorithm reaches a dead end, it simply discards intermediate results only to the recompute them on a slightly different search path.

A popular alternative to backtracking is the dynamic programming method known as chart parsing. Chart parsers store intermediate results in some data structure and use them to build other intermediate results, which are in turn stored and used to build ever more complete results. If we ever make the wrong non-deterministic choice, we simply retrieve the intermediate results and use those to explore a different search path. The importance of this decision is not simply about dealing with "incorrect" non-deterministic choices. Some parsing and generation tasks might call for not just one result, but *all* the possible results (parses or sentences) that a grammar would associate with an input. The parsing algorithm is forced to explore the entire search space, which makes it all the more desirable not to waste intermediate results.

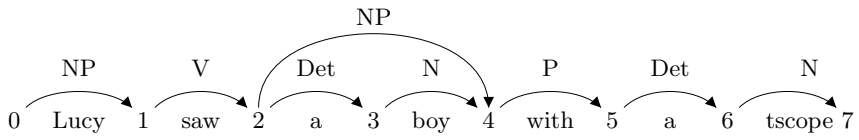


Figure 1.7: Part of a CKY chart

If tabular methods are useful for parsing, it is likely that they would also be useful for generation. In this section, we build our way up from chart parsing to chart generation. We begin by establishing a standard framework and notation for talking about chart-based algorithms (Section 1.3.1). To get a feel for the notation, we present the well-known CKY algorithm (Section 1.3.2) and proceed to dissect it and other algorithms, building up an anatomical guide to chart parsing techniques (Section 1.3.3). Then we attack chart generation proper (Section 1.3.4). As we will see, most of the chart parsing techniques can be transferred in a straightforward manner, the sole exception being indexing. The solution we (and many others) adopt is the use of a flat semantics as introduced by [Kay, 1996]. Kay's algorithm is important enough to go into detail about, so it is what we will wrap this discussion up with (Section 1.3.5).

1.3.1 Chart parsing as deduction

First the terminology and notation. The intermediate storage used by chart parsers is called a chart. It does not matter for our purposes how the chart is implemented. We can simply see it as a set of chart items. We can think of the chart items as corresponding to the edges of a labelled graph (Figure 1.7), so for convenience we may sometimes call the chart items edges.[4] The vertices of this graph are labelled by string positions and its edges by information that we can later use to build a parse tree. What exactly this information is varies from one algorithm to the next. For example, the CKY algorithm uses syntactic categories as labels, but as we will see below, other algorithms will use different kinds of edge label.

Another useful tool for talking about chart parsing is the framework of Parsing as Deduction, or more conveniently, deductive parsing [Shieber *et al.*, 1995]. This framework provides us (i) with a declarative way to talk about parsing algorithms which (ii) eschews needless implementation detail and (iii) makes it easier to see the similarities and differences between various algorithms. A deductive parsing algorithm consists of a set of axioms, goals and inference rules. These rules and axioms are used by some universal parsing engine whose job is to generate the set of possible "facts" that they allow. The axioms tell us facts which are always true, the goals give us facts that have to be true for parsing to succeed, and the inference rules help us get from the axioms to the goals. Inference rules have a premise (a set of facts) and a conclusion (a set of

[4] They are technically not the same thing — some algorithms may produce more than one chart item for the same edge — but the added fluency is worth it.

	mystery deductive parser
Axioms	$[A \to \bullet\beta, l, l]\ A \to \beta$
Goals	$[S\bullet, 0, n]$
Inference rules	$\dfrac{[A \to \beta \bullet w\delta, l, r]}{[A \to \beta w \bullet \delta, l, r+1]}\ w = w_{r+1}$ (Scan)
	$\dfrac{[A \to \beta \bullet C\delta, l, x] \qquad [C \to \eta\bullet, x, r]}{[A \to \beta C \bullet \delta, l, r]}$ (Comp)

Table 1.1: Deductive parsing notation

facts we may derive if the premise is true). So with inference rules, we start from an initial set of facts and produce new facts. If we apply the inference rules repeatedly, i.e. on facts which may themselves be derived from inference rules, then we can hopefully work our way towards the goal. This is precisely what the parsing engine does. It starts from the axiomatic facts and repeatedly applies the inference rules until no more facts can be generated.[5] Then it tells us if any facts correspond to the goal (if there is a parse) and how (what the parses are).

Deductive parsing fits chart parsing very naturally, which is why we are introducing the two notions simultaneously. For one thing, the universal engine that a deductive parser requires could be implemented as a chart parser such that every edge in the chart corresponds to some deductive parsing "fact" (and vice versa). So if chart parsing gives us an implementation of deductive parsing, what does deduction do for charts? The basic answer is that it simplifies the discussion of tree traversal strategies. Tree traversal may well be orthogonal to chart parsing, but looking at chart parsing from an intuitive standpoint, the first thing that comes to mind is a bottom-up strategy. As we saw in Section 1.1, it is more useful to have a mixed traversal strategy which combines top-down and bottom-up information. It is certainly possible to encode these strategies in a chart parser using "active" chart parsing techniques. In a deductive parsing framework, these techniques are quite simple to present; they are just another set of rules and axioms.

That said, let's have a closer look at the notation for deductive (chart) parsing. In Table 1.1, we have an example of a deductive parser. We call this simply "mystery deductive parser", because what is important here is not the algorithm but the notation. Some things to notice about this are that:

1. Every algorithm is presented as its axioms, goals and a set of inference rules.

[5] There is no inherent guarantee that this would terminate. This would have to be proved separately for each algorithm.

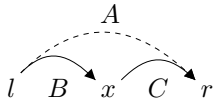
2. The axioms and goals manipulate facts (chart items), which we write in square brackets. An example of a fact that is compatible with this particular algorithm would be $[0, 3, \text{NP} \bullet \text{V PP}]$. What goes inside the square brackets depends on the actual algorithm used, although it is a common practice at least to keep track of string position indices.

3. Inference rules are written in two dimensions, with the premises in the top half and the conclusions in the bottom half. So in the (Comp) rule, the premises $[A \to \beta \bullet C\delta, l, x]$ and $[C \to \eta\bullet, x, r]$ have to be met in order for the conclusion $[A \to \beta C \bullet \delta, l, r]$ to be true.

4. Both axioms and inference rules may be accompanied by preconditions, written on the side. For example, the axiom $[A \to \bullet\beta, l, l]$ $A \to \beta$ has a precondition that there be a grammar rule $A \to \beta$. Likewise, the (Scan) rule has a precondition that the symbol w (in $A \to \beta \bullet w\delta, l, r$) matches the rth word in the input string.

5. Greek and Roman letters are used for different purposes. Roman letters correspond to single nodes, whereas Greek letters stand for a sequence of them. So B literally means the node B, whereas β stands for "some sequence of nodes which we will call β"

6. Rules sometimes include a dot $\bullet$ which serves as a sort of progress marker. The dot is never strictly necessary (although the rules would have to be written a bit differently) but is a useful visual aid. Section 1.3.3.5 will show how dotted rules are used.



CKY (Comp)

And since we will implement the facts as chart edges, we will occasionally accompany the deductive parsing notation with its chart visualisation. In the margin, for example, we see a visualisation of the CKY (Comp) rule in chart form. Solid lines indicate pre-existing chart edges (premises) and dashed lines indicate chart edges to be added (conclusions). Notice that the edges are labelled the same way as in deductive system, except that we have omitted the indices, as these can be inferred by looking at the vertices that the edges connect.

1.3.2 The CKY algorithm

CKY is the ancestor to all chart parsers [Kasami, 1965; Younger, 1967]. It makes for a good example of chart parsing because it is simple and yet covers a broad range of techniques. The algorithm works for context free grammars in Chomsky normal form, to which any CFG can be converted.[6]

Definition 1 (Chomsky normal form)**.** A grammar is in Chomsky normal form if and only if all its productions are of the form:
$A \to \alpha$
$A \to BC$
That is, it consists of unary rules with only a terminal symbol on its RHS, or a binary rule with only non-terminal symbols.

[6]It is also possible to create a variant of CKY that works with arbitrary CFGs, but the standard one is slightly simpler to present and has an $\mathcal{O}(n^3)$ upper bound.

	standard CKY
Axioms	$[A, l, l+1]\ A \to W, W = w_{l+1}$
Goals	$[S, 0, n]$
Inference rules	$\dfrac{[B, l, x] \qquad [C, x, r]}{[A, l, r]}\ A \to BC$ (Comp)

Table 1.2: The CKY algorithm

CKY (Table 1.2) uses a strictly bottom-up traversal of the syntactic tree. The axioms start us off with an edge for every word in the input sentence. The word must correspond to the terminal node of some unary production in the grammar, and the resulting edge connects the two neighbouring vertices l and $l+1$, the left and right positions of the word in the input string. As for the binary productions, these are walked up by the inference rules: we can infer a new edge for every pair of adjacent edges whose categories match the right hand side of some binary production. The new edge spans the two edges from which it was built (its indices are l, the left vertex of the first edge, and r the right vertex of the second edge), and its category is the left hand side of the production. Since the rules are continuously applied, the edges that result from a binary production can themselves lead to other edges being produced by inference rule. If at the end of processing, we have found an edge whose category is S and which connects the vertices 0 and n, we have successfully parsed the input string.

Figure 1.8 illustrates a CKY parse for the sentence "Lucy saw a boy with a telescope", using the grammar below. This sentence has two parses (Figure 1.9), which will both be found.

S → NP VP
VP → V NP | VP PP
NP → Det N | NP PP | Lucy
PP → P NP
V → saw
Det → a
P → with
N → boy | telescope

What makes CKY interesting is that it does not just tabulate intermediate results (this would be the core requirements for a chart parser) but also computes them in a compact manner. It does this with a combination of techniques, subtree sharing and local ambiguity packing. Subtree sharing consists in representing the intermediate results with a compact parse forest instead of a set of parse trees. We see this in the way that the PP edge "with a telescope" is used in both the NP "(a boy) (with a telescope)" and the VP "(saw a boy) (with a telescope)". Local ambiguity packing, a related technique, consists in merging

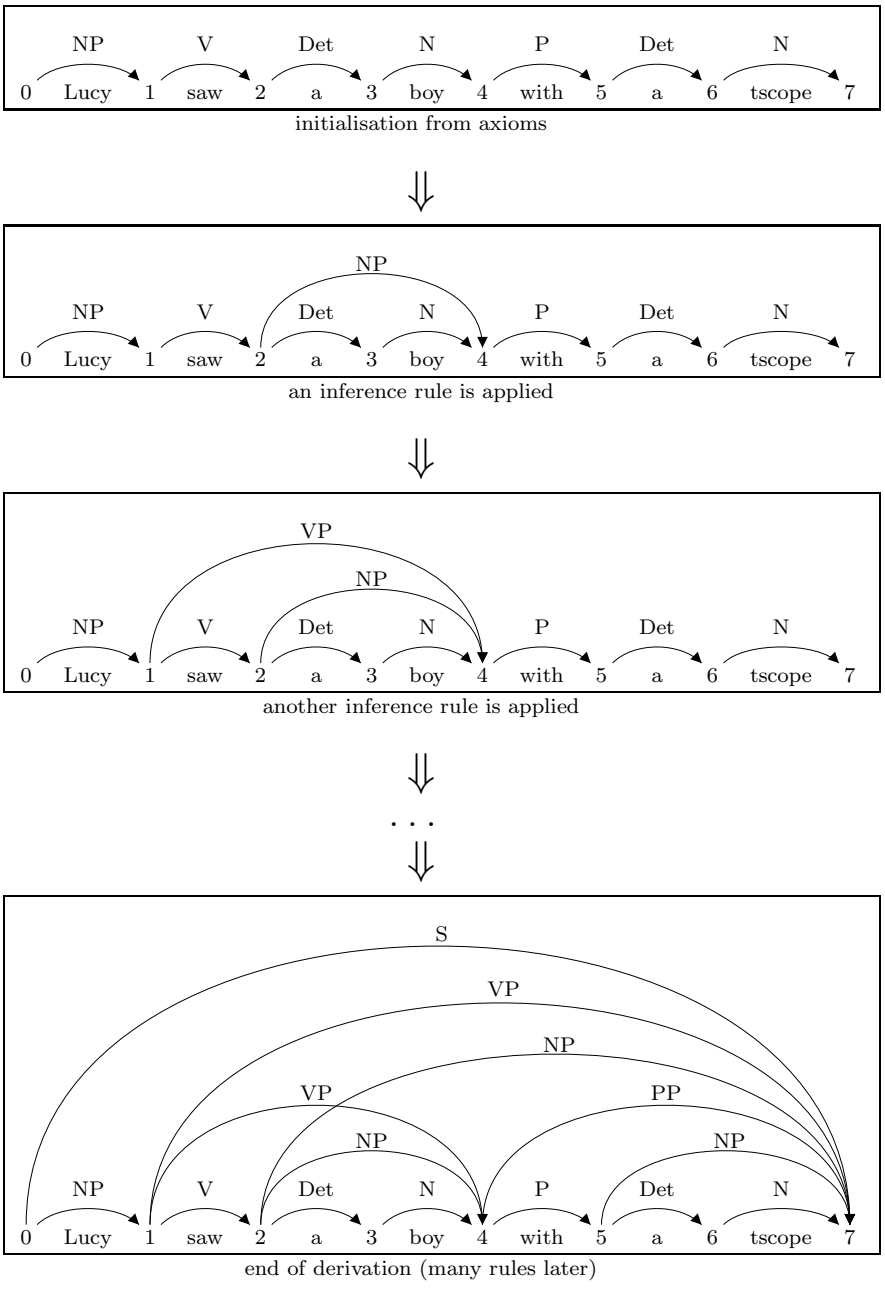


Figure 1.8: The CKY parser in action

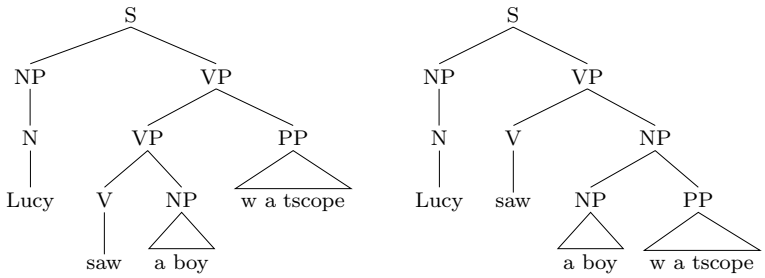


Figure 1.9: Two parses for one telescope

any chart items that can be considered equivalent. For example, the VP node "(saw a boy with a telescope)" has two possible derivations but is represented with a single chart item. Neither of these techniques require any add-ons to the chart parsing engine; they are a simple consequence of the deductive rules defined in Table 1.2.

1.3.3 An anatomy of chart parsing

If we dissect the CKY algorithm we can find some common chart parsing techniques: basic tabulation, indexing, sharing, packing and redundancy checking. In this section we will go over these techniques in greater detail and present two other techniques not implemented in CKY (dotted edges and tabulated prediction). The full catalogue is as follows:

tabulation storing and not recomputing chart items

indexing associating chart items with an index for filtering

sharing pointing to chart items instead of copying them

packing grouping chart items into equivalence classes

partial recognition tracking partial parses of n-ary rules

tabulated prediction storing top-down predictions in the chart

agenda control accompanying the chart by a "to do" list

1.3.3.1 Tabulation

As a bare minimum, chart parsers must tabulate intermediate results, which allows us to avoid the expensive recomputation that comes from backtracking. Tabulation does not necessarily imply sharing. A rudimentary chart parser could still inefficiently copy information from one chart edge to another when building new edges, but the fact that it tabulates these edges in the first place is already better than nothing.

1.3.3.2 Indexing

A standard practice in chart parsing is for chart items to contain a pair of starting and ending string positions. These positions are needed for correctness (two items should only combine if they really are adjacent), but they also help to keep parsing efficient. The indices serve as a quick filter on chart items, allowing us to rapidly determine if two chart items are potentially compatible.

This is an instance of the larger strategy of indexing. Combining edges, or even determining if they can combine, is a potentially costly operation and a very frequently used one since we may have to attempt all combinations of chart items to determine if new ones can be built. Some chart parsers may use additional indices (for example, the CKY-style parser for TAG has an additional pair of indices to keep track of adjunctions) or include "redundant" indices which only help to improve efficiency.

The notion of indexing could also be taken a step further by grouping the chart items into equivalence classes and associating each of these classes with an index. Doing this allows us to filter out entire sets of edges with a single index comparison without having to individually consider each of its members.

1.3.3.3 Subtree sharing

Another common feature of chart parsers is that chart items can be shared. This is not quite the same thing as basic tabulation. Tabulation just tells us that we should store chart items and use them to build new ones, but it does not say how to go about constructing the new edge. A possible default might just be to copy the relevant information wholesale. Subtree sharing improves on this situation by using a pointer to previous items instead of copying them around. This reduces the payload of each chart item to a bare minimum, but adds in a post-processing step if we want to recover parse trees. The trees have to be reconstructed by following the trail of pointers from the chart items to their origins. Subtree sharing does not change the number of chart items produced with respect to just tabulation; however, it makes each individual item cheaper to store (a chart item now has a constant 1 cost instead of the linear $2n - 1$ cost to build a binary tree). This savings is considerable because there can be an exponential number of chart items with respect to the input string.

Figure 1.10 shows the difference between tabulation with copying and tabulation with sharing. In this picture, we have stripped away the chart items which are not relevant to sharing. The item being shared here is `NP(a boy)`, which is used in both `NP(NP(a boy), with a telescope)` and `VP(saw,NP(a boy))`.

For many algorithms, indexing is used as a natural springboard for subtree sharing. For example, in CKY, we can reconstruct the parse trees out of chart items by following the trail of string position indices. But this does *not* mean that an indexing mechanism is necessary for sharing to work. For example, the "core" version of the parsing algorithm in [Neumann, 1994] uses a chart item counter assigning a unique identifier to each edge. This is not quite the same thing as an index because it is not useful for filtering chart items, but is perfectly adequate for subtree sharing.

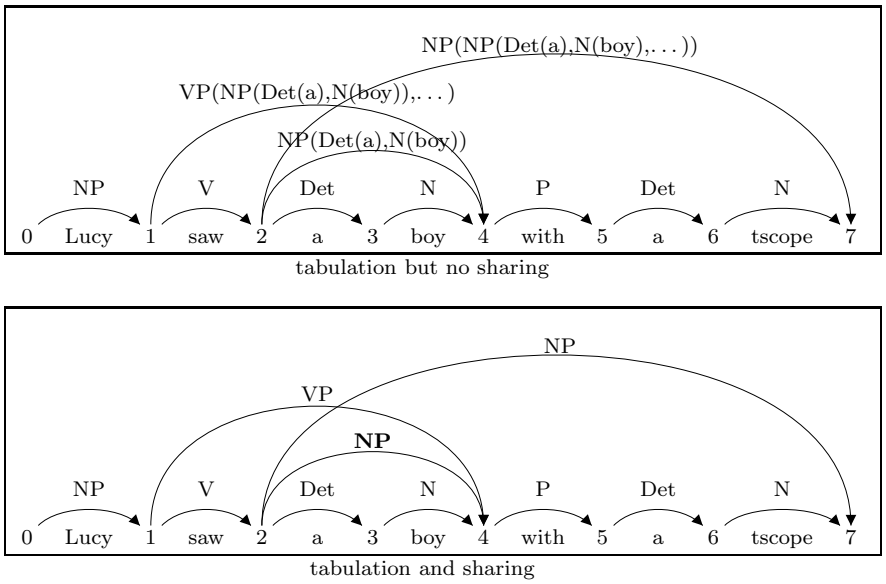


Figure 1.10: With and without subtree sharing

1.3.3.4 Local ambiguity packing

The notion of local ambiguity packing was introduced in [Tomita, 1987]. Packing is subtly different from sharing. In sharing we reuse a chart item in several places, whereas in packing, we merge two different chart items together. Figure 1.11 shows the difference between subtree sharing and packing. Sharing is the recycling of `NP(a boy)` in two different chart items. Packing, on the other hand, consists in treating the two different derivations of `VP(saw a boy with a telescope)` as a single chart item. As opposed to subtree sharing, this can significantly reduce the number of chart items being built. Packing allows for an exponential number of trees to be encoded in polynomial space.

Packing is implemented by the generic chart parsing engine (see Section 1.3.1), but it requires that the individual algorithm supply an equivalence criterion between items — only equivalent items should be packed. The criterion could be the same used for building equivalence classes in an indexing scheme, but neither one necessarily implies the other.

For context free grammars, two items can be considered equivalent if they are equal. For unification grammars, on the other hand, a subsumption check is more appropriate: specific items should be packed into general ones. This requires that the parsing engine support some destructive operations: if a more general item is produced when a more specific one already exists, the more specific one should be replaced with the general one (and any bookkeeping information be merged). Subsumption-based packing can also be inexact because not all of the derivations for a specific chart item may extend to the general one, so a chart parser with subsumption-based packing must also include an unpacking phase at the end of parsing to filter out the invalid unpackings.

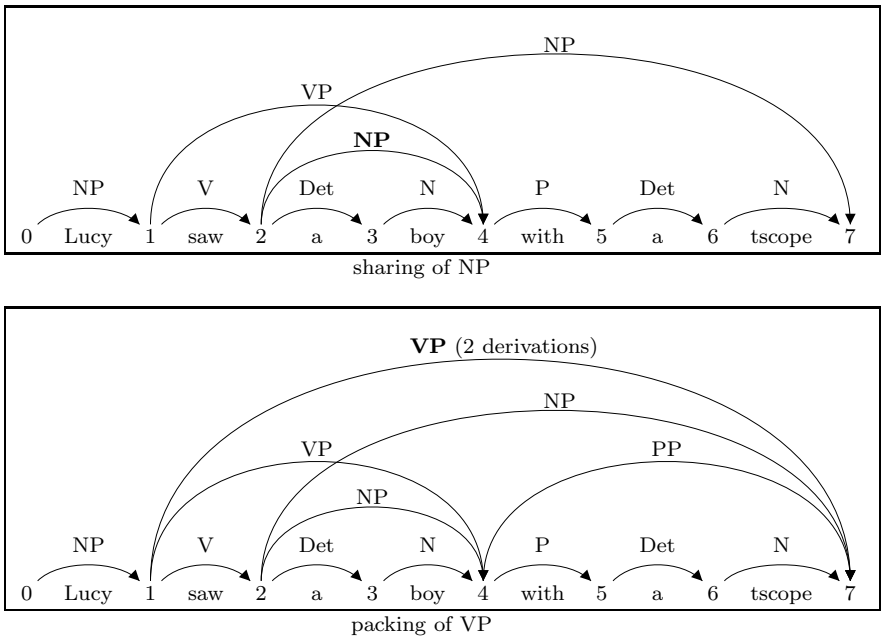


Figure 1.11: Subtree sharing vs. packing

One interesting side-effect of packing is that it can help to avoid the kind of infinite loops that the top-down and head-corner algorithms suffer from. Because we do not visit the same intermediate results twice, we avoid one potential source of infinite loops. This is what allows for context free chart parsing algorithms to use a top-down traversal (for example, Earley's algorithm) without running into the left-recursion problem. As usual, the same guarantees do not extend to unification grammar. For example, we might have chart items with lists in them (e.g. subcategorisation lists). Starting from the top down, we may build a new list out of a pre-existing list by consing a new element onto it. As a minimal example, given the list `[]`, we might produce a new list `X:[]`. This new list is not subsumed by the previous list and so it will not be detected by the packing scheme.[7]

1.3.3.5 Partial recognition

One problem with standard CKY is that the grammars have to be converted to Chomsky normal form (see Definition 1). CKY could be modified straightforwardly to work with n-ary rules, but doing so causes us to lose the $\mathcal{O}(n^3)$ upper bound on the parsing algorithm. How can we make use of n-ary rules without sacrificing efficiency?

The solution consists in introducing a notion of active chart items, in addition to the inactive items that standard CKY uses. The idea is that active

[7] This may be more obvious with an alternative notation. For instance, `cons(X,[])` is not subsumed by `[]`.

	standard CKY	**dotted CKY**
Axioms	$[A, l, l+1]\ A \to W, W = w_{l+1}$	$[A \to \bullet\beta, l, l]\ A \to \beta$
Goals	$[S, 0, n]$	$[S\bullet, 0, n]$
Inference rules		$\dfrac{[A \to \beta \bullet w\delta, l, r]}{[A \to \beta w \bullet \delta, l, r+1]}\ w = w_{r+1}$ (Scan)
	$\dfrac{[B, l, x] \quad [C, x, r]}{[A, l, r]}\ A \to BC$ (Comp)	$\dfrac{[A \to \beta \bullet C\delta, l, x] \quad [C \to \eta\bullet, x, r]}{[A \to \beta C \bullet \delta, l, r]}$ (Comp)

Table 1.3: Dotted CKY

chart items correspond to partial recognition of a rule, on the way to building an eventual inactive item. We distinguish between active and inactive items with the dot notation. Here is a dotted production: $A \to \beta\bullet\gamma$. It indicates that the portion β of the production has already been recognised (as mentioned on Page 5, Greek letters stand for some arbitrary sequence of nodes). The chart items and parsing algorithm are thus modified to keep track of these dotted productions. Table 1.3 presents the rules of this augmented CKY, with the standard CKY alongside for comparison.

What sets standard and dotted CKY apart is their completion rules. Standard CKY completion really does "complete" the traversal of a rule in the sense that given a CKY rule $A \to BC$, if both B and C have both been parsed, then A has also been parsed. Dotted CKY completes only one "step" of the traversal: if for some $A \to \beta C\delta$, we have $\bullet C$ and if we successfully parsed some rule $C \to \eta$, then we note this by moving the dot to the right, yielding $C\bullet$. The big difference here is that we now actually *store* the partial recognition, which we did not do when we were working solely with inactive chart items.

1.3.3.6 Tabulated prediction

So far we have been using the (dotted) CKY algorithm to illustrate the chart parsing techniques. Here, we introduce a technique which is not used in that algorithm. Tabulated prediction is an application of the dotted chart items (active chart items) introduced above. It allows us to transform a purely bottom-up tree traversal strategy into a top-down one (with left-recursive disaster averted by packing).

$$\frac{[A \to \beta \bullet C\delta, l, r]}{[C \to \bullet\eta, r, r]}\ C \to \eta$$

The algorithm is almost identical to dotted CKY (see Table 1.4 for a comparison using the deductive parsing notation and Figure 1.12 using a visual

	dotted CKY	**Earley**
Axioms	$[A \to \bullet\beta, l, l]\ A \to \beta$	$[S \to \bullet\beta, 0, 0]\ S \to \beta$
Goals	$[S\bullet, 0, n]$	
Inference rules	$\dfrac{[A \to \beta \bullet C\delta, l, r]}{[C \to \bullet\eta, r, r]}\ C \to \eta$ (Pred)	
	$\dfrac{[A \to \beta \bullet w\delta, l, r]}{[A \to \beta w \bullet \delta, l, r+1]}\ w = w_{r+1}$ (Scan)	
	$\dfrac{[A \to \beta \bullet C\delta, l, x] \qquad [C \to \eta\bullet, x, r]}{[A \to \beta C \bullet \delta, l, r]}$ (Comp)	

Table 1.4: Dotted CKY vs. Earley deduction

chart parsing notation), save the initialisation axiom and an extra inference rule. The initialisation axioms are tightened up. Whereas dotted CKY will initialise on every rule in the grammar, Earley will only do so for rules with the start symbol on the left hand side. To make up for this, we introduce a new (Pred) rule. This new prediction rule makes all the difference. It does not have the exuberance of dotted CKY, predicting everything in sight. Instead, it only performs predictions on confirmed active chart items. This effectively transforms bottom-up traversal into top-down traversal, because (bottom-up) scans and completions will only be initiated when they have been successfully predicted. The difference is that they will only be predicted if there is a good reason to do so. Note that the theoretical complexity of both algorithms is the same, $\mathcal{O}(n^3)$ in time and $\mathcal{O}(n^2)$ in space, but in practice, the Earley algorithm presents a sizeable reduction of the search space because it prevents the kind of false starts to which a purely bottom-up approach is prone.

1.3.3.7 Agenda based control

Some chart parsing algorithms use a secondary data structure known as an agenda. The agenda acts as a sort of "todo" list of edges. Using it usually involves some variant of the following algorithm schema [Kay, 1996]:

1. Move an item from the agenda to the chart.
2. If it may combine with other items in the chart, add any resulting new edges to the agenda
3. If there are no more chart items to consider, stop, otherwise start over.

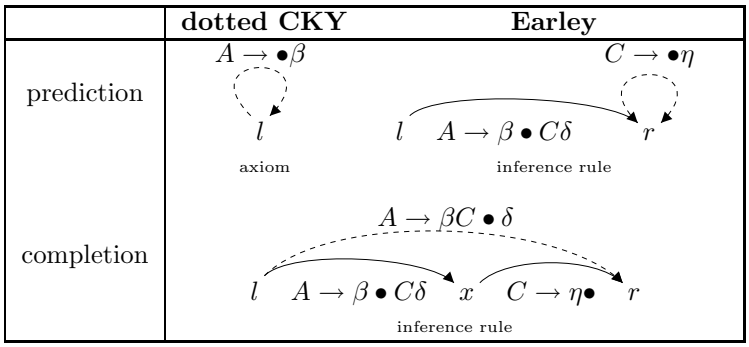


Figure 1.12: CKY/Earley visualisation

We have so far presented chart parsing in terms of a generic deduction-based engine where the tree traversal and search strategy can be supplied as a parameter. Having an agenda lets us parameterise the search strategy as well. For example, using a stack would give us depth-first disambiguation, (using whatever tree traversal and control strategy the rules provide for), or implement it as a queue and get breadth-first search. More sophisticated strategies could be implemented still. [Shieber, 1988]'s uniform architecture, for instance, uses a priority queue for the agenda and can be parameterised for different search strategies by changing the function that assigns priorities to edges. This can model the usual depth-first and breadth-first strategies, but also richer strategies that model psycholinguistic phenomena such as minimal attachment. Clearly, this kind of parameterisability is an improvement over the alternative of hard-coding a strategy into the parser. Agendas and deduction systems do not necessarily imply one another but they fit together in a natural fashion.

On the other hand, note that for a given tree traversal strategy, the same set of chart items will ultimately be generated no matter how we organise the agenda. The only thing that changes is the order in which we produce the chart items. The choice of an agenda is largely irrelevant for chart parsers that exhaustively explore the search space; but it becomes useful when pruning strategies come into play.

1.3.4 Chart generation and indexing

The common ingredients that make a chart parser can also be transferred to surface realisation. Chart generation has largely the same objectives as chart parsing: to construct syntactic trees for a given input. In Table 1.5, we can see the techniques implemented by three early chart generators: the uniform architecture of [Shieber, 1988], the uniform computational model of [Neumann, 1994] and [Kay, 1996]'s chart generation. Some features are standard: core tabulation obviously, structure sharing, partial recognition with active chart items and agenda-based control. [Kay, 1996] does not appear to implement a

	Shieber 1988	Neumann 1994	Kay 1996
core tabulation	yes	yes	yes
structure sharing	yes	yes	yes
packing	yes	yes	no
indexing	no	yes	yes
partial recognition	yes	yes	yes
tabulated prediction	yes	yes	no
agenda based control	yes	yes	yes

Table 1.5: Some chart generation algorithms

packing scheme or use tabulated prediction (his algorithm appears to be akin to dotted CKY), but this is because his main point is to introduce a novel mechanism for indexing.

Indexing was the main issue in chart generation until Kay's proposal. The problem is that the input to generation systems is inherently more complicated[8] than that of parsing. Whereas parsers deal with linear sequences of words (or at worst, automata), generators have to work with semantic trees or graphs as input. If we are only dealing with strings, we can say that chart edges are only allowed to combine with edges that are adjacent to each other in the sequence. This does not work for generation because it is the strings that we are trying to output; we do not know what the sequence is yet, so we cannot tell what items are adjacent to each other. What would be more appropriate for generation would be something based on the input semantics, but what?

Shieber's uniform architecture

The earliest answer to this question was "nothing". The uniform architecture of [Shieber, 1988] essentially does not use an indexing mechanism for generation. Shieber uses a single set of inference rules for parsing and generation (under the deductive parsing framework). For generation, he systematically sets the string positions to 0 so that all substrings are considered to be adjacent to each other, and potentially worth combining. In other words, he deactivates the indexing mechanism when doing generation.

Indexing with essential features

[Neumann, 1994] builds on [Shieber, 1988] by using a single framework for parsing and generation, using an Earley-style algorithm and a unification grammar with a context free backbone. One of the innovations he introduces is an indexing mechanism which can be parameterised to work for either parsing or

[8] We really mean complicated here, and not complex in the sense of computational complexity. On the other hand, it is also true that surface realisation is NP-complete with a Context-Free Grammar, and for that matter, with a Tree Adjoining Grammar [Brew, 1992; Koller and Striegnitz, 2002]

generation. The basic idea is that for parsing, the index of a chart item consists of the string it covers, whereas for generation the index is the semantics. Comparing indices is linear with respect to the size of the input, because it involves at least a string comparison. Neumann reduces the number of index comparisons that need to be made by grouping the chart items into equivalence classes (see Section 1.3.3.2). The general scheme is as follows:

1. Chart items consist of a tuple $[h \leftarrow b_0..b_n, x, in, from]$. $h \leftarrow b_0..b_n$ is an active production. x points to the currently selected node in $b_0..b_n$ (i.e. b_x). It serves the same purpose as the $\bullet$ in active chart parsing, but Neumann uses head-first traversal (as opposed to left-to-right), so the $\bullet$ notation may be a bit misleading. in and $from$ are pointers to equivalence classes; see [Neumann, 1994] for details.

2. Item sets are tuples of the form $\langle AL, PL, Idx \rangle$, where AL is the set of active chart items in the set, PL the set of inactive edges and, Idx the thing they have in common. Only active edges and passive edges in the same class may combine.

3. Idx matches an essential feature of a certain node in each chart item.

4. The essential feature is taken from different parts of the chart item, depending on whether it is an active item or an inactive one.

   a) For inactive items, it is taken from the root node h.

   b) For active items, it is instead taken from the currently selected node b_x (again chosen using head-first, then left-to-right selection).

5. Parameterising the indexing mechanism consists in changing what feature we consider to be essential.

   a) For parsing, the essential feature is PHON, the portion of the input string that the node represents.

   b) For generation, it is SEM, the semantics of the node.

This scheme looks a little surprising at first glance because it seems to imply that two chart items combine if (for example), they share the same string. But this really is equivalent to what traditional chart parsing algorithms like CKY do. In traditional chart parsing, we say that the right index of the active chart item must match the left index of the passive item. In Neumann's algorithm, we say that the essential feature of the xth node in the active edge must match the index of the root node of the passive edge. The equivalence classes are really two-in-one, an equivalence class for active items, and a class for inactive items, the idea being that all the items in the active class are equivalent to the others (in the active class), all the items in the inactive class are equivalent to the others (in the inactive class), and finally that all the items in the active class may combine with all the items in inactive class.

In any case, the point is that [Neumann, 1994] algorithm shows one possible way to get indexing in a chart generator — just use the whole (semantic) feature structure — as an index. There is a shortcoming to this approach, however, which is that it only works in a head-first and top-down tree traversal (e.g. using the Earley algorithm). In order to check if two semantic features are the

same, they must first be instantiated, which is what happens when we use a head-first, top-down traversal. This excludes more lexically-driven, bottom-up approaches (on the other hand, see the remark on Page 13 about the top-down or bottom-up nature of Earley traversal).

Indexing with a flat semantics

A more flexible approach to indexing was introduced by [Kay, 1996], and has been incorporated into many chart generators since [Carroll *et al.*, 1999; Striegnitz, 2000; White, 2004]. The basic approach relies on the use of a flat semantics. We will present flat semantics in greater detail in Chapter 2, but for now 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. The following formula is an example of a flat semantics:

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

The key idea here is that semantic indices, like r or j can also serve as chart indexes in the sense that active and inactive edges should only combine if they articulate around a common semantic index.[9] This is the approach we use in this thesis, so we now delve into a more detailed presentation of the algorithm proposed by Kay.

1.3.5 Kay1996 in detail

The idea of using a flat semantics for generation was first introduced with the SHAKE-AND-BAKE generator [Whitelock, 1992]. SHAKE-AND-BAKE introduces the idea of a generation problem (SHAKE-AND-BAKE generation) which starts from a multiset of richly structured lexical signs instead of a conventional logical form. A rudimentary algorithm for SHAKE-AND-BAKE generation was introduced by [Whitelock, 1992], shift-reduce parsing with an unordered stack, and later improved by [Brew, 1992] with constraint propagation.

Of course, what we now call a flat semantics is no more than a multiset of richly structured lexical signs. Kay's algorithm can thus be seen as a new approach to SHAKE-AND-BAKE generation, using chart parsing techniques. The algorithm works by analogy with chart parsing of free-word-order languages, treating the input semantics as a free-order string and each literal as a word in that string. Following [Kay, 1996], we present the algorithm in two stages, using a core version to show chart generation from a flat semantics, and an extended version to show how the indexing mechanism works.[10]

Semantics, grammar and lexicon

The input consists of a list of literals, as below

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

[9] Depending on your point of view, it is (un)fortunate that we use the same word "index" for both semantic and chart indices.

[10] Regarding the "a bit" in Table 1.6, we can think of the semantic-overlap check called for by the basic version as a sort of indexing mechanism. Kay does not refer to it as such.

	basic	extended
core tabulation	yes	yes
structure sharing	no	no
packing	no	no
indexing	a bit	*yes*
partial recognition	no	*yes*
tabulated prediction	no	no
agenda based control	yes	yes

Table 1.6: Chart generation in [Kay 1996]

GRAMMAR:

$$\begin{aligned} \mathrm{S}(X) &\rightarrow \mathrm{NP}(Y)\ \mathrm{VP}(X,Y) \\ \mathrm{VP}(X,Y) &\rightarrow \mathrm{VP}(X),\ \mathrm{Adv}(X) \end{aligned}$$

LEXICON:

words	**cat**	**semantics**
John	`np(X)`	*name(X,john)*
ran	`vp(X,Y)`	*run(X), arg1(X,Y), past(X)*
fast	`adv(X)`	*fast(X)*
quickly	`adv(X)`	*fast(X)*

Figure 1.13: A small grammar and lexicon

The grammar is assumed to contain only binary rules. The lexicon associates a sequence of words with a category and a semantics. The category and semantics are linked by unification variables. Figure 1.13 shows an example of a grammar and lexicon used.

Basic version

Kay's algorithm uses a bottom-up tree traversal. Since deductive parsing is a useful way to talk about parsing and generation algorithms, we attempt to recast [Kay, 1996] in deductive parsing terms in Table 1.7. Note that this table does not mention unification, which it should by rights, but we have simplified it away because it obscures the basic mechanics we are trying to get at. (See Appendix C for a version with unification taken into account). Also, we have omitted the indices which are associated with each grammar rule when they are not relevant, for example writing $A \rightarrow BC$ where we really mean to say $A(a_1, .., a_n) \rightarrow B(b_1, .., b_n)C(c_1, .., c_n)$.

Axioms Chart items have the form $[words, cat, sem]$. The initial chart items are instantiated from the lexicon by selecting the items whose semantics subsumes the input semantics and performing the appropriate variable substitutions.

Given an input semantics sem and lexicon L:	
Axioms	$[w; A; lexsem(w)] \quad A \rightarrow w,\ w \in L,\ lexsem(w) \subseteq sem$
Goals	$[w_1..w_n; S; sem]$
Inference rules	$\dfrac{[w_1..w_i; B; sem_b] \quad [w_i..w_j; C; sem_c]}{[w_1..w_j; A; sem_b \cup sem_c]}$ $A \rightarrow BC,\ sem_b \cap sem_c = \emptyset$ (Comp)

Table 1.7: The basic Kay 1996, modulo unification

AGENDA

words	cat	semantics
John	`np(j)`	*name(j,john)*
ran	`vp(r,j)`	*run(r), arg1(r,j), past(r)*
fast	`adv(r)`	*fast(r)*
quickly	`adv(r)`	*fast(r)*

CHART

words	cat	semantics

Figure 1.14: Agenda/chart in the beginning of realisation

Goal The realiser has a simple goal, to produce a chart item whose category corresponds to the start symbol of the grammar and whose semantics matches the input semantics. Checking if the semantics matches can be done rapidly by means of a bit vector.

Inference rule The algorithm uses a single (Comp) rule in the standard CKY style. The (Comp) rule has a rudimentary form of indexing (treating the semantics as an index), which says that no two chart items should combine if their semantics overlap. This prevents the realiser from exploring useless and incorrect combinations like "run fast" with "quickly". In any case, if two chart items can be combined, the semantics of the resulting item is the set-union of the semantics of the original chart items. As usual, we can implement these operations efficiently by using a bit vector.

Basic version: an example

Let us work through a small example of the basic algorithm. We use the semantics *run(r), past(r), fast(r), arg1(r,j), name(j,john)* as input and the small grammar and lexicon from Figure 1.13.

Figure 1.14 shows the chart items which are produced from the axioms. We are using agenda-based control, moving items off the agenda one at a time

AGENDA (ITERATION 2)

words	cat	semantics
fast	`adv(r)`	*fast(r)*
quickly	`adv(r)`	*fast(r)*
John ran	`s(r)`	*run(r), arg1(r,j), past(r), name(j,john)*

CHART (ITERATION 2)

words	cat	semantics
John	`np(j)`	*name(j,john)*
ran	`vp(r,j)`	*run(r), arg1(r,j), past(r)*

AGENDA (ITERATION 4)

words	cat	semantics
John ran	`s(r)`	*run(r), arg1(r,j), past(r), name(j,john)*
ran quickly	`vp(r,j)`	*run(r), arg1(r,j), past(r), fast(r)*
ran fast	`vp(r,j)`	*run(r), arg1(r,j), past(r), fast(r)*

CHART (ITERATION 4)

words	cat	semantics
John	`np(j)`	*name(j,john)*
ran	`vp(r,j)`	*run(r), arg1(r,j), past(r)*
fast	`adv(r)`	*fast(r)*
quickly	`adv(r)`	*fast(r)*

Figure 1.15: Agenda/chart after 2 and 4 iterations

and combining them with elements in the chart. In the first iteration, we move the item "John" off the agenda and onto the chart. In the next iteration, we do the same with item "ran", only now, there is something on the chart that can combine with it by the (Comp) rule. We thus create a new chart item "John ran" with the semantics being the union of the previous two items' semantics. Kay seems to use a queue for an agenda (which simulates a breadth-first search), so we add the new item onto the end of the agenda. In the next two iterations, the chart items for "ran" and "fast" combine to form "ran fast" and "ran quickly". Figure 1.15 shows the agenda and chart after these second and fourth iterations. Finally (over the next three iterations) "John ran" does not combine with anything, but "ran fast" combines with "John", as does "ran quickly", yielding two results. This run can be summarised below using the first letter of every word as an abbreviation.

Combinations	Agenda	Chart	Results
	j,r,f,q		
	r,f,q	j	
(j,r) via t1	f,q,jr	j,r	
(r,f) via t2	q,jr,rf	j,r,f	
(r,q) via t2	jr,rf,rq	j,r,f,q	
	rf,rq	j,r,f,q	
(j,rf) via t1	rq	j,r,f,q	jrf
(j,rq) via t1		j,r,f,q	jrf, jrq

Given an input semantics **sem** and lexicon L:	
Axioms	$[w; A\bullet; lexsem(w)] \quad A \rightarrow w,\ w \in L,\ lexsem(w) \subseteq sem$
Goals	$[w_1..w_n; S\bullet; \mathbf{sem}]$
Inference rules	$\dfrac{[w_1..w_i; B\bullet; sem_b]}{[w_1..w_i; A \rightarrow B \bullet C(c\ldots); sem_b]} \quad A \rightarrow BC(c\ldots),$ (Comp1)
	$\dfrac{[w_1..w_i; A \rightarrow B \bullet C(x\ldots); sem_b] \quad [w_i..w_j; C(x\ldots)\bullet; sem_c]}{[w_1..w_j; A\bullet; sem_b \cup sem_c]} \quad A \rightarrow BC(x\ldots),\ sem_b \cap sem_c = \emptyset$ (Comp2)

Table 1.8: Kay 1996 deluxe (with indices)

Extended version (with indexing)

Now that we have the basic algorithm in place, we can improve it with a more aggressive notion of indexing. There are two modifications needed to support this notion.

1. (Comp1) We introduce a notion of dotted edges. For simplicity, we still assume that the grammar has only binary rules; the point of introducing dotted edges is just to have the distinction between active and inactive edges. The introduction of dotted edges results in a new inference rule which promotes inactive edges into active ones.[11]

2. (Comp2) When an active edge with label $A \rightarrow B \bullet C(x_a\ldots)$ and is to be combined with an inactive edge with label $C(x_i\ldots)$, we ensure that the indices x_a and x_i are the same. This check is actually superfluous. We get it for free just by checking if the active edge and inactive chart items may interact (in fact, the only key modification needed is the introduction of dotted edges). But here we make it explicit to show how semantic indices are put to use as chart indices.

Table 1.8 shows the modified version of the algorithm. As usual, we have omitted indices where they are not relevant, so for example, $A \rightarrow BC(x\ldots)$ should really be read as $A(a_1, .., a_i) \rightarrow B(b_1, ..b_j)C(x, c_2, ..c_k)$.

Extended version: an example

Now, for an example. We will be realising from the input semantics below and the grammar and lexicon in Figure 1.16.

[11] Kay does not seem to specify what should happen when there is not enough information to ground all the variables in the resulting active edge. We have constructed his approach with the assumption that they pass through unmodified. Non-ground indices do not appear to be fatal since they will simply be compatible with any other index (by unification). The idea of filtering by semantic indices is nevertheless useful, at least for the cases they do ground.

GRAMMAR

$$\begin{aligned} \mathrm{S}(X) &\rightarrow \mathrm{NP}(Y)\ \mathrm{VP}(X,Y) \\ \mathrm{VP}(X,Y) &\rightarrow \mathrm{V}(X,Y,Z)\ \mathrm{NP}(Z) \\ \mathrm{NP}(X) &\rightarrow \mathrm{Det}(X)\ \mathrm{N}(X) \end{aligned}$$

LEXICON

words	**cat**	**semantics**
cat	`n(X)`	*cat(X)*
dog	`np(X)`	*dog(X)*
saw	`v(X,Y,Z)`	*see(X,Y,Z), past(X)*
the	`det(X)`	*def(X)*
a	`det(X)`	*indef(X)*

Figure 1.16: "The dog saw a cat" grammar and lexicon

dog(d), def(d), see(s,d,c), past(s), cat(c), indef(c)

The derivation of "The dog saw a cat" can be narrated as follows:

0. (Figure 1.17) The initial agenda contains the lexically selected items, all inactive edges with nothing to combine with.

1. (Figure 1.18) In these next five iterations, the initial items are all moved off the agenda. Some of them get promoted into active edges via (Comp1).
. . .

7. (Figure 1.18) An active edge with the string "the" combines with the inactive edge for "dog" to yield an inactive edge "the dog".

8. (Figure 1.19) "saw" has nothing to combine with yet and is moved off the agenda without incident.

9. (Figure 1.19) The "a" active edge combines with "cat" to produce "a cat".

10. The passive edge "the dog" is promoted into an active edge via (Comp1).

11. Both (Comp1) and (Comp2) apply to "a cat", yielding both an active version "a cat", and the combined "saw a cat"

12. The active versions of "the dog" and a cat have nothing to combine with, and are moved off the agenda with no incident. . . .

14. (Figure 1.20) The inactive edge "saw a cat" combines with the active "the dog" via (Comp2), yielding the final result "the dog saw a cat"

Here is the full derivation in compact table form (active edges are marked with an exclamation mark)

Combinations	Agenda	Chart	Results
	t,d,s,a,c		
Comp1(t) via t3	d,s,a,c,t!	t	
	s,a,c,t!	t,d	
Comp1(s) via t2	a,c,t!,s!	t,d,s	
Comp1(a) via t3	c,t!,s!,a!	t,d,s,a	
	t!,s!,a!	t,d,s,a,c	
Comp2(t!,d) via t3	s!,a!,td	t,t!,d,s,a,c	
	a!,td	t,t!,d,s,a,c	
Comp2(a!,c) via t3	td,ac	t,t!,d,s,a,a!,c	
Comp1(td) via t1	ac,td!	t,t!,d,td,s,a,a!,c	
Comp1(ac) via t1	td!,ac!,sac	t,t!,d,td,s,a,a!,c,ac	
Comp2(s!,ac) via t2			
Comp1(ac) via t1	sac	t,t!,d,td,td!,s,a,a!,c,ac,ac!	
Comp2(td!,sac) via t1		t,t!,d,td,td!,s,sac,a,a!,c,ac,ac!	tdsac

1.4 Summary of the main issues

There are three major considerations in building a surface realisation algorithm.

1. Tree traversal is a question of how we uncover the syntactic tree we are trying to build. It can be decomposed into the questions of ascendancy (i.e. top-down vs. bottom-up vs. corner based) and of child selection (i.e. left-to-right vs. head-driven).

2. Search deals with the question of which choices we make when we encounter an ambiguity in the grammar. A large variety of search strategies are available (depth-first, best-first, etc), as are pruning strategies (anywhere from the full pruning to no pruning).

3. Chart parsing consists of a variety of techniques which most algorithms have in common. The techniques can be applied to generation in a relatively straightforward manner and the question of indexing can be addressed with the use of a flat semantics.

These factors tie into the fundamental choices we will see in the next two chapters. The main reason for using a flat semantics is that it helps us cope with the logical-form equivalence problem; but it is also useful for chart generation. Likewise, the main reason for using Tree Adjoining Grammar (TAG) is that it is useful from a linguistic point of view, but it is also nice because, in a sense TAG gives us head-corner traversal for free (the base linguistic units in TAG are trees of arbitrary depth; so we can think a TAG tree as directly encoding the chain-rule processing step of head-corner traversal).

AGENDA

words	cat	semantics
the	`det(d)` •	*def(d)*
dog	`n(d)` •	*n(d)*
saw	`v(s,d,c)` •	*see(s,d,c), past(s)*
a	`det(c)` •	*indef(c)*
cat	`n(c)` •	*n(c)*

CHART

words	cat	semantics

Figure 1.17: "The dog saw a cat" after lexical selection

AGENDA (ITERATION 5)

words	cat	semantics
the	`np(d)` → `det(d)` • `n(d)`	*def(d)*
saw	`vp(s,d)` → `v(s,d,c)` • `np(c)`	*see(s,d,c), past(s)*
a	`np(c)` → `det(c)` • `n(c)`	*indef(c)*

CHART (ITERATION 5)

words	cat	semantics
the	`det(d)` •	*def(d)*
dog	`n(d)` •	*n(d)*
saw	`v(s,d,c)` •	*see(s,d,c), past(s)*
the	`det(c)` •	*indef(c)*
cat	`n(c)` •	*n(c)*

AGENDA (ITERATION 6)

words	cat	semantics
saw	`vp(s,d)` → `v(s,d,c)` • `np(c)`	*see(s,d,c), past(s)*
a	`np(x)` → `det(c)` • `n(c)`	*indef(c)*
the dog	`np(d)` •	*def(d)*

CHART (ITERATION 6)

words	cat	semantics
the	`det(d)` •	*def(d)*
dog	`n(d)` •	*n(d)*
the	`np(d)` → `det(d)` • `n(d)`	*def(d)*
saw	`v(s,d,c)` •	*see(s,d,c), past(s)*
a	`det(c)` •	*indef(c)*
cat	`n(c)` •	*n(c)*

Figure 1.18: Early iterations of "The dog saw a cat"

AGENDA (ITERATION 7)

words	cat	semantics
a	`np(x) → det(c) • n(c)`	*indef(c)*
the dog	`np(d) •`	*def(d), dog(d)*

CHART (ITERATION 7)

words	cat	semantics
the	`det(d) •`	*def(d)*
dog	`n(d) •`	*n(d)*
the	`np(d) → det(d) • n(d)`	*def(d)*
saw	`v(s,d,c) •`	*see(s,d,c), past(s)*
saw	`vp(s,d) → v(s,d,c) • np(c)`	*see(s,d,c), past(s)*
a	`det(c) •`	*indef(c)*
cat	`n(c) •`	*n(c)*

AGENDA (ITERATION 8)

words	cat	semantics
the dog	`np(d) •`	*def(d), dog(d)*
a cat	`np(c) •`	*indef(c), cat(c)*

CHART (ITERATION 8)

words	cat	semantics
the	`det(d) •`	*def(d)*
the	`np(d) → det(d) • n(d)`	*def(d)*
dog	`n(d) •`	*n(d)*
saw	`v(s,d,c) •`	*see(s,d,c), past(s)*
saw	`vp(s,d) → v(s,d,c) • np(c)`	*see(s,d,c), past(s)*
the	`det(c) •`	*indef(c)*
a	`np(x) → det(c) • n(c)`	*indef(c)*
cat	`n(c) •`	*n(c)*

Figure 1.19: More iterations of "The dog saw a cat"

AGENDA (ITERATION 14)

words	cat	semantics
the dog saw ↪ a cat	`s(s) •`	*def(d), dog(d), see(s,d,c), past(s),* ↪ *indef(c), cat(c)*

CHART (ITERATION 14)

words	cat	semantics
the	`det(d) •`	*def(d)*
the	`np(d) → det(d) • n(d)`	*def(d)*
dog	`n(d) •`	*n(d)*
the dog	`np(d) •`	*def(d), dog(d)*
the dog	`s(X) → np(d) • vp(X,d)`	*def(d), dog(d)*
saw	`v(s,d,c) •`	*see(s,d,c), past(s)*
saw	`vp(s,d) → v(s,d,c) • np(c)`	*see(s,d,c), past(s)*
saw a cat	`vp(s,d) •`	*see(s,d,c), past(s), indef(c), cat(c)*
the	`det(c) •`	*indef(c)*
a	`np(c) → det(c) • n(c)`	*indef(c)*
cat	`n(c) •`	*n(c)*
a cat	`np(c) •`	*indef(c), cat(c)*
a cat	`s(X) → np(c) • vp(X,c)`	*def(c), cat(c)*

Figure 1.20: Final iteration of "The dog saw a cat"

Chapter 2

Flat semantics with holes

Surface realisation algorithms can differ in their tree traversal, disambiguation and control strategies. But what about the more fundamental choices? In these next two chapters we will explore two of the core assumptions made by the surface realiser GenI : that our input semantics comes in the form of a flat semantics and that the grammar used is a Feature-Based Tree Adjoining Grammar (FB-LTAG). This chapter deals with the flat semantic assumption. We have briefly introduced flat semantics in the discussion of chart generation. Here, we expand on what exactly a flat semantics is (Section 2.1). We cover the main motivation behind a flat semantics (it is a partial workaround to the logical-form equivalence problem, Sections 2.2 and 2.3). Finally, we shall go over an issue which arises in the context of generation with a flat semantics (intersective modifiers, Section 2.4).

2.1 Flat semantics

Flat semantic representations were not initially used to tackle the logical-form equivalence problem *per se*. The idea of flattening logical formulas was first introduced for theorem-proving [Gabbay, 1996] and later adopted into NLP to enable compact representations of scope ambiguities.

2.1.1 How to iron a formula

Semantic representation languages must handle recursion in one way or another. Natural language expressions can have an arbitrary length; an expression can be built out of another expression. More importantly, their meanings can be built out of the meanings of other expressions, and to an arbitrary depth. For example, consider the sentence "A dog barks." In first-order logic, we might represent this as $\exists d(dog(d) \wedge bark(d))$; but as we see below, this sentence and its semantics can be nested indefinitely:

1. A dog barks.
   $\exists d(dog(d) \wedge bark(d))$
2. A man complains that a dog barks.
   $\exists m(man(m) \wedge complain(m, \exists d(dog(d) \wedge bark(d))))$

3. A neighbour says that a man complains that a dog barks.
   $\exists n(neighbour(n) \wedge say(n, \exists m(man(m) \wedge complain(m, \exists d(dog(d) \wedge bark(d))))))$

4. ...

Flat semantic representations are also recursive, although they express this recursion in a less direct manner. Rather than include other semantic formulas, flat semantics *refer* to them. Any semantic representation language can be flattened. For example, one possible flattening would be (i) identify every non-recursive subformula with a label (ii) replace every embedded labelled formula with its label (iii) recurse. Below is how such a scheme might be applied to the semantics of "Ernest considers buying a dog" in first-order logic:

1. $\exists d(dog(d) \wedge consider(ern, buy(ern, d)))$

2. $\exists d(l1 \wedge consider(ern, l2))$
   l1:dog(d), l2:buy(ern,d)

3. $\exists d(l1 \wedge l3)$
   l1:dog(d), l2:buy(ern,d), l3:consider(ern,l2)

4. $\exists d(l4)$
   l1:dog(d), l2:buy(ern,d), l3:consider(ern,l2), l4:and(l2, l3)

5. *l1:dog(d), l2:buy(ern,d), l3:consider(ern,l2), l4:and(l2, l3), l5:exists(d,l4)*

2.1.2 L_U : an application of hole semantics

The flat semantic language used in this thesis is L_U [Gardent and Kallmeyer, 2003]. It is a reformulation of Predicate Logic Unplugged (PLU) where lambda variables are replaced by unification variables. PLU is an application of hole semantics [Bos, 1995], a general framework for flattening and introducing scope underspecification into semantic representation languages (for example, Bos shows how Discourse Representation Theory can be unplugged). The L_U language somewhat simplifies PLU and adds unification variables for purposes of semantic construction.

There are basically two kinds of L_U formulas, saturated and unsaturated. The difference is that saturated formulas are devoid of unification variables. Surface realisers, parsers and grammars all traffic in unsaturated formulas, but ultimately their goal is to produce a saturated formula.

Definition 2 (L_U formula syntax)**.** Let

- I_{var} be a set individual unification variables and I_{con} be set of individual constants;
- L_{var} be a set of label unification variables and L_{con} be a set of label constants;
- H be a set of "hole" constants and
- R be a set of n-ary relations over $I_{var} \cup I_{con}$.

Given $l \in L_{var} \cup L_{con}$, $h \in H$, $i_1, \ldots, i_n \in I_{var} \cup I_{con} \cup H$ and $R^n \in R$. Then the unifying formulas (UF) of L_U are defined as follows:

1. $l : R^n(i_1, \ldots, i_n)$ is a UF of L_U
2. $h \geq l$ is a UF of L_U
3. If ϕ is a UF of L_U and ψ is a UF of L_U, then ϕ, ψ is a UF of L_U
4. Nothing else is a UF of L_U

Unification and semantic formula construction are not particularly relevant to the question of what makes L_U and flat semantic languages attractive. We will thus ignore unsaturated formulas for now (see Section 3.3).

L_U also supports a notion of underspecification. In L_U, there is a process called plugging, which converts a saturated L_U formula into a set of plugged saturated formulas (or plugged formulas for short). For a full definition of plugged formulas, see [Gardent and Kallmeyer, 2003]. Here, it suffices to say that plugged formulas are saturated formulas which are devoid of hole constants. We will largely ignore the question of underspecification in this thesis, and focus mainly on the plugged formulas.[1]

The set of L_U plugged formulas is equivalent to the set of first-order logic formulas. Converting to first-order logic (FOL) consists of translating some distinguished relations (like `exists` and `forall`) to the FOL equivalents and replacing all labels with the formula that is being labelled.[2] Converting back requires undoing this process. In other words, plugged L_U formulas can be recursively unflattened into FOL ones, and FOL ones can be flattened back ([Copestake *et al.*, 2005] shows the flattening from first-order logic formulas into MRS sans specification of links, which is equivalent to plugged L_U formulas).

2.2 Logical-form equivalence

The problem of logical-form equivalence was first mentioned in [Appelt, 1987] and [Shieber, 1988]. Since then, other researchers have claimed to have solved the problem [Calder *et al.*, 1989; Levine, 1990], prompting Shieber to clarify the issue and to reassert its intractability. There is no solution to the problem, short of solving the AI knowledge-representation problem and addressing the thorny philosophical question of what it means to "mean the same" [Shieber, 1993].

The problem starts off innocently enough. We can see a grammar as expressing a relation between strings and one or more logical form. Ideally, we would simply associate the string with a faithful representation of its meaning, but since it is far from clear exactly what "meanings" are, we use a logical form as an approximation. There are thus three players to deal with: strings, logical forms and meanings. The relationship between the three can be summarised as follows (see also Figure 2.1):

- A string may have more than one meaning

[1] On the other hand, Section 3.3 shows an example of unplugged formulas being used.

[2] [Gardent and Kallmeyer, 2003] do not mention this, but we also adopt the convention from MRS that relations with the same label are considered to be under conjunction.

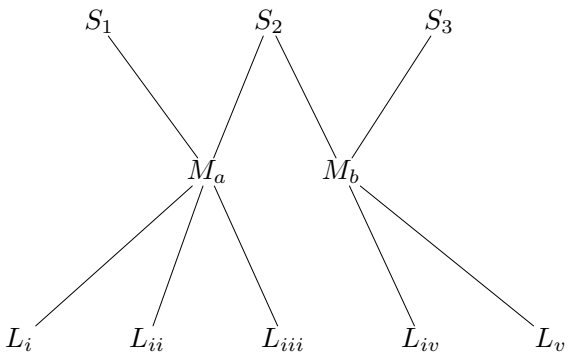


Figure 2.1: Strings, meanings and logical forms

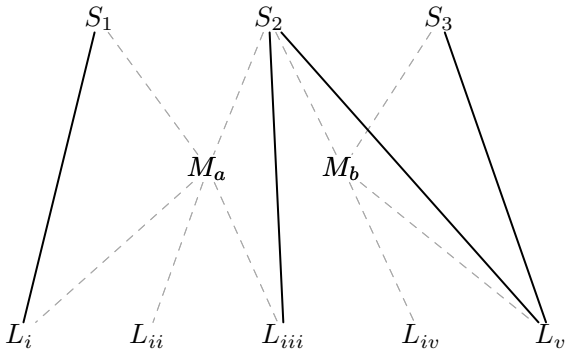


Figure 2.2: Strings, meanings and *canonical* logical forms

- A meaning may be realised by more than one string
- A meaning may be represented by more than one logical form
- On the other hand, a logical form stands for only one meaning.

The crux of the problem is that a single meaning may be associated with more than one logical form. In practice, one usually expects a grammar to associate a string with only one of the logical forms. Following Shieber, we will call this the canonical logical form for that string (under the given interpretation; that is, each distinct meaning gives rise to a distinct logical form). Figure 2.2 shows a possible mapping from strings to their canonical logical form in the grammar. The two things to notice in this figure are that (i) a string is only associated to one logical form for each of its meanings and (ii) two strings with the same meaning can be associated with different logical forms each (for example, "John threw the ball" and "The ball was thrown by John") might get assigned a different logical form).

The job of a parser is then to compute the meaning of each string, which it accomplishes by building its canonical logical form(s). Generation systems have a converse job, deriving a string from a meaning representation. But this is trickier. If the input does not correspond to any of the canonical logical forms recognised by the grammar, it means that we simply get no output. For example, in Figure 2.2, say that the grammar does not associate L_{ii} to any strings. If the generator was given L_{ii} as input, it would not be able to generate anything from it. Let's try this on a more concrete example. Suppose we had the following grammar:

t1	S(S)	$\rightarrow$ NP(X) VP($\lambda X.S$)
t2	VP($\lambda X.S$)	$\rightarrow$ V($\lambda Y \lambda X.V$) NP($\lambda Y \lambda V.S$)
t3	NP($\lambda X \lambda C.S$)	$\rightarrow$ Det($\lambda X \lambda R \lambda C.S$) N($\lambda X.R$)
t4	N($\lambda X.A \wedge N$)	$\rightarrow$ Adj($\lambda X.A$) N($\lambda X.N$)
t5	NP($john$)	$\rightarrow$ John
t6	N($\lambda X.cat(X)$)	$\rightarrow$ cat
t7	Det($\lambda X \lambda R \lambda C.\exists X.(R \wedge C)$)	$\rightarrow$ a
t8	Adj($\lambda X.white(X)$)	$\rightarrow$ white
t9	V($\lambda Y \lambda X.has(X, Y)$)	$\rightarrow$ has

The logical form generated by this grammar for the sentence "John has a white cat" would be $(\exists x.(white(x) \wedge cat(x)) \wedge has(john, x))$. This input could be fed into the grammar and the expected sentence would be realised. But what if we had instead inverted the arguments? If we feed a slightly different input to the grammar, say $\exists x.has(john, x) \wedge (white(x) \wedge cat(x))$, we would not get any output from the grammar. These two logical forms are semantically equivalent, so by rights we should be able to generate from either one (and get the same result); however, because they are not syntactically equivalent, we can only generate from one of them.

The problem of logical form equivalence (LFE) is to be able to do just this, to generate from all the logical forms that have the same meaning (Figure 2.2). The key point is that deciding logical equivalence is undecidable in first-order logic. Plausible sounding algorithms (such as converting to a normal form) do not bypass this fundamental problem. In this section, we visit a series of potential solutions and show why none of them actually solve the problem.

2.2.1 Eliminate canonical logical forms?

The first solution might be to do away with the notion of canonical logical forms. We could simply force the grammar to associate a string's meaning with all the equivalent logical forms. In the first place, this is impossible (consider: ϕ is equivalent to $\phi \vee T$, $\phi \vee T \vee T$, ...). Of course, we could impose some sort of upper bound on the length of the logical form, but this makes grammars much less useful for parsing, because now, if the parser returns more than one logical form (as it inevitably would), we would not actually know if the string was ambiguous or not. We would have to go through the process of canonicalising all the logical forms and then checking how many canonical forms we have left, so back to square one.

2.2.2 Demand canonical logical forms as input?

Another possibility is to simply refuse to solve the problem. To see why this would not work, it helps to see where these logical forms come from in the fist place. Basically, they are a product of the division of labour between strategic generation (deciding what to say) and tactical generation (deciding how to say it). The idea is as follows: a strategic component does higher level reasoning tasks — Shieber calls this a reasoner — and the language-dependent tactical component deals with lower level linguistic tasks like surface realisation. The logical forms are thus to be produced by the reasoner and fed to the tactical generator. So at first glance it seems plausible to simply declare that the tactical generator *requires* a canonical logical form, pushing the burden over to the reasoner; however, as [Shieber, 1993] points out, "[w]hich of the many logical forms representing a meaning is canonical is a grammatical issue, not a semantic one, and reasoners should not have to truck with grammatical issues." Insisting upon a canonical logical form as input would break down that division of labour.

2.2.3 Abolish the division of labour?

If this division of labour is so problematic, one would think that we could simply get rid of it and wash our hands free of the problem. Moreover, some researchers reject the distinction between a strategic and tactical generator (for reasons unrelated to logical-form equivalence) anyway. [Danlos, 1984], for example, shows that conceptual decisions cannot be made independently of linguistic ones. It seems then that to get rid of the strategic/tactical barrier would solve two problems at once.

There are three reasons to reject this proposal. First, from an engineering standpoint, it is easier to develop large systems out of small components with well-defined interfaces, than it is to develop a single monolithic system. Second, even though the tools may be distinct from each other, there is no major reason that they could not be used in an integrated or incremental fashion. Therefore, even if one does not believe in the strategic versus tactical distinction, there is scant advantage to be had in merging the two tools. Finally, a more theoretical objection is that even if the tools were merged, the problem would not really disappear. Shieber argues that the resulting uni-component would still have to do the same reasoning and linguistic tasks anyway, and would thus be as complex as the two components combined, if not more so. So if we are going to be stuck with something as difficult as the original problem anyway, we might as well try to solve the original problem and keep the engineering benefits of separating the two components.

2.2.4 Compute the equivalences?

So what happens if we resign ourselves to solving the logical-form equivalence problem? We can begin by looking more closely at its two practical consequences. The first, as mentioned earlier, is if a reasoner feeds a non-canonical form to the tactical generator, the generator might not produce any output. Say we are using a first-order logic and a grammar which associates each of the following strings with a canonical logical form:

(9) a. John threw a large red ball.
$\exists x.(throw(j,x) \wedge (large(x) \wedge (red(x) \wedge ball(x))))$

b. John threw a red ball that is large.
$\exists x.(throw(j,x) \wedge (red(x) \wedge (ball(x) \wedge large(x))))$

c. John threw a large ball that is red.
$\exists x.(throw(j,x) \wedge (large(x) \wedge (ball(x) \wedge red(x))))$

These sentences have arguably the same meaning, but which sentence is produced ultimately depends on which logical form we put in. This is fine until we come up with a logical form that happens not to be supported by the grammar, say $\exists x.(throw(j,x) \wedge (ball(x) \wedge (large(x) \wedge red(x))))$. To produce any output at all, the generator must be able to convert arbitrary logical forms into a canonical one.

The second consequence is more subtle. The tactical generator must also be able to recognise the equivalence between different canonical logical forms already supported by the grammar. In other words, the tactical generator should "know" the sentences above all have the same meaning. It should not just produce the one string that verbalises its input semantics, but *all* the strings with an equivalent meaning. Why? Because although the sentences may have the same core meaning, they express finer distinctions (like what is being emphasised). If the reasoner has some pragmatic constraint to express, it must be able to do so without having to arrange the logical form to suit the tactical generator. It has to remain language-independent. So to support this possibility, the tactical generator must be able to produce all paraphrases, perhaps being guided by some external constraints expressed outside of the logical form.

These two equivalence tasks are one and the same; solve one and you solve the other. At the very least, one should be able to compare two logical forms and determine if they are equivalent. But even for "relatively inexpressive" logics like first-order logic, the problem of deciding logical equivalence is undecidable. So, gamely computing logical form equivalences is not a solution either.

2.2.5 Use a weaker notion of equivalence?

If logical equivalence is undecidable for FOL, the obvious solution seems then to be that we should switch to a more restricted logic (for example, any formula in propositional logic can be converted to disjunctive normal form in exponential time). This led to the original claims of solving the LFE, to which [Shieber, 1993] is a response. The response is basically that such solutions treat logical equivalence "as an end to itself, rather than a means to an end".

The reason we care about logical form equivalence in the first place is that it is supposed to be an approximation to meaning identity. If two sentences supported by the grammar have the same meaning, their logical forms should be considered equivalent. But is logical equivalence really a *good* approximation of meaning identity? It depends on the notion of equivalence. For example, the notion of equivalence supported by first-order logic can already be considered too weak. It considers too few logical forms as being equivalent.

The problem of fine-grainedness is illustrated in [Shieber, 1993] with an example using a first-order language augmented with some generalised quantifiers and equality:

(10) a. Clapton was the leader of Derek and the Dominos.
$the(x, leader\text{-}of(d, x), c = x)$

b. The leader of Derek and the Dominos was Clapton.
$the(x, leader\text{-}of(d, x), x = c)$

c. Clapton led Derek and the Dominos.
$led(c, d)$

The sentences above are synonymous, Shieber asserts, and it would be reasonable to expect a grammar to associate them with canonical logical forms as above. The problem is that even if we could compute FOL equivalence, it would not tell us if 10b and 10c have the same meaning. We would have to extend equivalence to include some sort of world knowledge, along the lines of

$$\forall x, y.led(x, y) \equiv leader\text{-}of(y, x)$$

There is nothing inherently wrong with doing so, but it only underscores the point that the notion of equivalence supported by FOL is already too weak as it is.

2.2.6 Remark: too weak *and* too strong

It is worth pointing out the problem of finding a good notion of logical equivalence is not simply a problem of finding a notion of equivalence that is powerful enough. We saw above that the equivalences supported by FOL are too few, but as we see here, they are also too many. For example, conjoining a tautology with a logical form gets us something which is equivalent to the initial form:

$$led(x, y) \equiv led(x, y) \wedge (rain \vee \neg rain)$$

The problem here is that if our generator were really to take logical equivalence to heart, it might treat "Clapton led Derek and the Dominos" and "Clapton led Derek and the Dominos and either it is raining or not raining" as being interchangeable. This is clearly not desirable because the two sentences do not mean the same thing.

2.2.7 There is no way out

Any claims to solving the logical form equivalence problem should be treated with great scepticism. The LFE problem is deep. As users, we expect the grammar to associate strings with meanings. But as far as the grammar is concerned, "meanings" do not actually exist; it manipulates strings and logical forms. We only *pretend* that the logical forms are meanings and run into trouble when they are not so. Solving this problem requires that we find a notion of logical equivalence which faithfully reflects the deeper notion of meaning identity, which as Shieber argues, is a manifestation of the AI knowledge representation problem. Since we cannot solve this problem, we opt for a more modest approach: capturing some of the more common equivalences with a flat semantic representation.

2.3 The case for a flat semantics

Computing logical form equivalence is necessary for surface realisation, but undecidable for even first-order logic. As we saw in Section 2.2, there is no feasible way to solve this (AI-hard) problem. Flat semantics can be used as a sort of workaround to the problem. They provide a variant of the non-solution which consists in using a weaker notion of equivalence. Languages like L_U and MRS may have equal expressive power to first-order logic (we can translate formulas from one representation to the other); however, they naturally support a much weaker notion of equivalence which is convenient for generation. Basically, two formulas are considered equivalent if and only if they contain the same exact literals. Let's see how this is useful for dealing with logical-form equivalence.

2.3.1 Equivalence is cheap

The first advantage is that this type of equivalence is extremely cheap to verify. We merely sort the literals and check for syntactic identity.

2.3.2 Equivalence is not too strong

The use of flat semantic representations should only be considered a workaround for some of the more practical instances of the logical-form equivalence problem. With this weakened notion of equivalence, there are many formulas which a flat-semantic-driven tactical generator may fail to recognise as being equivalent. For example, De Morgan's laws ($\neg(p \wedge q) \iff (\neg p) \vee (\neg q)$) are not handled using the L_U notion of equivalence. But perhaps this is more a feature than a bug. As we saw in Section 2.2.6, it is not clear that it is even desirable to compute all the equivalences permitted by the logical formalism. For that matter, flat semantic representations clearly avoid the problem of recognising too many formulas as being equivalent. For example, the formula *l0:led(x,y)* is not equivalent to *l0:led(x,y), l1:or(l2,l3), l2:rain, l3:not(l2).* Of course, the main worry behind the LFE is the inverse: that too *few* formulas are treated as equivalent. Nevertheless, it is beneficial to weaken some aspects of equivalence even further.

2.3.3 Conjunction is commutative and associative

Although this notion of equivalence is weak, it allows us to handle some of the more common equivalences. The main equivalence that can be captured is the commutativity and associativity of conjunction. To see why this is useful, it helps to borrow some examples from [Copestake *et al.*, 2005], which motivate flat semantics for machine translation. The problem in machine translation is to find appropriate transfer rules that can rewrite semantic expressions in one language to expressions in another language.

This can be seen as a variant of the logical-form equivalence problem. For example, the German word "Schimmel" is equivalent to the English "white horse", so we could map directly from one to the other (11a). But then what happens when we try to translate "white English horse" into German?

(11) a. $(white(x) \wedge horse(x)) \leftrightarrow Schimmel(x)$

b. white English horse
$white(x) \wedge (English(x) \wedge horse(x))$

The problem in (11b) is the bracketing of the semantic expression. Without the associativity and commutativity of conjunction, we would not be able to regroup the $white(x) \wedge horse(x)$ and produce the desired $Schimmel(x)$.

Again from [Copestake *et al.*, 2005], consider the English string "fierce black cat" and its natural Spanish equivalent "gato negro y feroz". We can see some plausible logical forms for them in (12a) and (12b) respectively. Now say we wanted to translate from Spanish to English. A naive transfer component would produce the logical form (12c), and this form might not be accepted by the English grammar.

(12) a. $fierce(x) \wedge (black(x) \wedge cat(x))$ (natural English)

b. $gato(x) \wedge (negro(x) \wedge feroz(x))$ (natural Spanish)

c. $cat(x) \wedge (black(x) \wedge fierce(x))$ (Spanglish)

As with the previous example, the only problem lies in the bracketing. Really, the two logical forms (12a) and (12c) should be considered equivalent. The approach proposed in [Copestake *et al.*, 2005] (and adopted for L_U) is to introduce a form of n-ary, commutative, associative conjunction. This is encoded by associating every conjoined literal with the same label. So in MRS and L_U, the above black cat could be rendered as any of the permutations of

(13) a. *l1:fierce(x), l1:black(x), l1:cat(x)*

Revisiting the red ball example from Section 2.2.4 (this time, a purely generation-oriented example), the L_U representations of the following three sentences would be considered equivalent:

(14) a. John throws a large red ball.
l0:∃(x,l1), l1:throw(j,x), l2:large(x), l2:red(x), l2:ball(x)

b. John throws a red ball that is large.
l0:∃(x,l1), l1:throw(j,x), l2:red(x), l2:ball(x), l2:large(x)

c. John throws a large ball that is red.
l0:∃(x,l1), l1:throw(j,x), l2:large(x), l2:ball(x), l2:red(x)

2.3.4 Scope can be compactly represented

It is worth noting that it is possible for a flat semantic representation to be *too* flat in the sense that scopal distinctions are lost. For example, in (15), overzealous flattening has caused us to lose an important distinction between two sentences:

(15) a. $every(x) \wedge horse(x) \wedge old(x) \wedge white(x)$

b. Every old horse is white

c. Every white horse is old

This is avoided by semantic representations where each literal is associated with a label. In L_U, the semantics above would be rendered as either one of the following:

(16) a. *l0:every(x,l1,l2), l1:horse(x), l1:old(x), l2:white(x)*
Every old horse is white

b. *l0:every(x,l2,l1), l1:horse(x), l2:old(x), l1:white(x)*
Every white horse is old

As an added bonus, L_U and other such formalisms also provide facilities for underspecifying the scope if need be. This results in a single compact representation of semantics with possible scope ambiguities. See [Copestake *et al.*, 2005] and [Gardent and Kallmeyer, 2003] for details.

2.3.5 Indexing is cheap

We had mentioned one of the practical advantages of flat semantic representations in Chapter 1, that indexing in chart generation is straightforward to implement. There are two kinds of indexing that flat semantic representations easily support. The first kind is checking that the semantics of two chart items do not overlap. This consists in taking the intersection of their two semantics and can be implemented by some very cheap bit vector operations. The second kind (Section 1.3.5) consists in making sure that the outgoing semantic index of an active chart item matches the incoming semantic index of the passive chart item it is being combined with. This too is a very cheap operation.

On the other hand, nothing in chart generation really hinges on the use of a flat semantics. For example, [Neumann, 1994] implements indexing with recursive feature structures. The only apparent difference is that indexing is $\mathcal{O}(1)$ with a flat semantics [3] and $\mathcal{O}(n)$ with a recursive one (with respect to the string length of the semantic input).[4] That said, indexing check is performed for each pair of chart items being potentially combined, so it is worthwhile for the check to be as efficient as possible.

2.4 Intersective modifiers

As we saw above, flat semantics can treat conjunction as commutative and associative at little cost. Unfortunately, the commutativity and associativity comes at a price: we have now introduced a source of word order ambiguity. That ambiguity is not specific to flat semantic representations. It occurs anywhere that conjunction is commutative and associative. Confusing the issue, on the other hand, is that viewing a flat semantics as just a free-word-order string, chart generation is $\mathcal{O}(2^n)$. But as we saw in Section 2.1, flat semantics are not really free-word-order strings. Order is imposed by the indices which semantic literals share amongst themselves... except for the case of intersective

[3] Checking if two semantics overlap is $\mathcal{O}(n)$, where n is the number of literals in the input semantics. But it is a very cheap $\mathcal{O}(n)$ (bit vectors). The second kind of indexing is $\mathcal{O}(1)$

[4] It appears to be $\mathcal{O}(n)$ anyway if we assume that the semantic features of the chart items being compared are ground. The comparison in Neumann's case is to check if the semantics match, so this is akin to a string comparison.

modifiers. With intersective modifiers, the commutativity and associativity of conjunction become problematic. If a string has a word with k modifiers, the realiser will produce 2^k versions of that string, one for each subset of modifiers. For example, here are the $2^3 = 8$ possible subsets of modifiers in "fierce little black cat":

(17) cat,
fierce cat,
little cat,
black cat,
fierce little cat,
fierce black cat,
little black cat,
fierce little black cat

The 2^k possible subsets is already bad in itself, but it is also compounded by a number of factors:

1. We have to realise not just the word, but also a full sentence using that word and anything in between. If the sentence being realised is "the fierce little black cat runs", we might produce the strings "cat", "the cat" and "the cat runs"... with the 8 variants for each string.

2. More than one word in the string may have intersective modifiers. Consider "the fierce little black cat jumps over the lazy brown dog".

This problem is also *unrelated* to the issue of word order or lexical ambiguity, but these can only make matters worse:

3. Allowing the modifiers to combine in any order (for example, "fierce little cat" and "little fierce cat") means that each one of the 2^k subsets of modifiers can be permuted in a factorial number of ways.

4. Lexical ambiguity (meaning that a literal may be realised by more than one lexical entry, e.g. "quickly" vs. "fast") means that there are $\prod_i a_i^n$ choices of one lexical item per literal, where a_i is the ambiguity of the literal i.

To date, four solutions have been proposed for dealing with the intersective modifiers problem.

2.4.1 Sealed indices

The first proposal was made in [Kay, 1996]. The idea is that we can distinguish between internal and external indices. External indices are those which appear on the left hand side of the rule. For example, in the grammar fragment $\mathrm{VP}(X, Y) \rightarrow \mathrm{V}(X, Y, Z)\ \mathrm{NP}(Z)$, the variables `X` and `Y` would both correspond to external indices. The variable Z, on the other hand, corresponds to an internal index; in a sense, it is hidden by the grammar rule.

To see how this distinction would be used, consider the semantics

def(a), young(a), tall(a), polish(a), athlete(a), run(e), arg1(e,a)

Using the grammar fragment below, we should be able to generate the sentence "The tall young Polish athlete runs."

$$\begin{array}{ll}
\mathrm{S}(E) & \rightarrow \mathrm{NP}(S)\ \mathrm{VP}(E,S) \\
\mathrm{NP}(X) & \rightarrow \mathrm{Det}(X)\ \mathrm{N}(X) \\
\mathrm{N}(X) & \rightarrow \mathrm{Adj}(X)\ \mathrm{N}(X) \\
\mathrm{Det}(X) & \rightarrow \text{the} \\
\mathrm{Adj}(X) & \rightarrow \text{tall} \\
\mathrm{Adj}(X) & \rightarrow \text{young} \\
\mathrm{Adj}(X) & \rightarrow \text{polish} \\
\mathrm{N}(X) & \rightarrow \text{athlete} \\
\mathrm{VP}(E,S) & \rightarrow \text{runs}
\end{array}$$

Assuming that the grammar only allows for one word order, we now have $2^3 = 8$ possible subsets of modifiers to build, when all but one are ultimately useless. The solution that Kay proposes is basically to enforce that all modifiers are inserted into a word before trying to do anything else with it. Specifically:

> [A]s a matter of principle, no edge should be constructed if the result of doing so would be to make internal an index occurring in part of the input semantics that the new phrases does not subsume. [Kay, 1996]

For example, we would not be allowed to build up the sentence "the athlete runs", because doing so would require that (i) we invoke the grammar rule $\mathrm{S}(r) \rightarrow \mathrm{NP}(a)\ \mathrm{VP}(r,a)$[5] which (ii) makes the index `a` internal, although (iii) the literals *tall(a), polish(a), young(a)* are yet to be subsumed. On the other hand, this mechanism would allow for "the tall polish athlete runs" to be built because every literal which uses the index `a` is now subsumed, so we do not care if the index is made internal or not. In other words, the key behind this mechanism is that we do not allow for an edge to "seal off" semantic indices that we may still need.

This does not solve the exponential nature of the problem (we still build the 2^k subsets of modifiers), but it does mitigate the two compounding factors. First, when a word is part of a larger string, the surrounding string no longer uses all of the 2^k subsets. We still build them, but only use the fully instantiated one. So whilst we would still get "tall athlete", "tall young athlete", etc.; we at least avoid building "the tall athlete", "the tall Polish athlete", "the tall Polish athlete runs". The strings would require the full "tall young Polish athlete" lest they seal off its semantic index prematurely.

Similarly, having more than one modified word is less of an issue because partial subsets of modifiers no longer interact. For example, in "the nosy neighbour says the tall young Polish athlete runs", we only combine the full "nosy neighbour" with the full "tall young Polish athlete", and not with anything in between. If we have a word with n modifiers and another one with m modifiers, we now only have to deal with $2^n + 2^m$ combinations, and not the whole $2^{(n+m)}$.

2.4.2 Delayed modifier insertion

The sealed index mechanism has some limitations in practice. As observed in [Carroll *et al.*, 1999], the problem is that we have to make indices external in

[5] We have instantiated the unification variables for expository reasons.

sealed indices	**delayed modifier**
Polish athlete,	the athlete,
young athlete,	the athlete runs,
tall athlete,	
young Polish athlete,	the Polish athlete runs,
tall Polish athlete,	the young athlete runs,
tall young athlete,	the tall athlete runs,
tall young Polish athlete,	the young Polish athlete runs,
	the tall Polish athlete runs,
the tall young Polish athlete,	the tall young athlete runs,
the tall young Polish athlete runs	the tall young Polish athlete runs

Table 2.1: Sealed indices vs. delayed modification

order to propagate them correctly throughout the grammar. Doing so means that we do not seal any indices or prevent the spurious combinations that result. Extending Carroll *et al.*'s example, consider the sentence "How did the newspapers say John ran to the store yesterday in the heat?" The problem here is that the index for the running event has to be propagated all the way up the tree, so that it can be modified by "How" and "quickly". But this means that we never seal off the index. Consequently, we do not prevent such useless combinations as:

(18) How did the newspapers say John ran?
How did the newspapers say John ran to the store?
How did the newspapers say John ran yesterday?
How did the newspapers say John ran to the store yesterday?
...

[Carroll *et al.*, 1999] propose an "inverse" solution to the problem. Instead of building up all the modifiers in a word and then inserting it into a larger structure, Carroll *et al.* build the surrounding structure first and inserts the modifiers later. We can compare the two approaches by the edges they end up building on an example, "the tall young Polish athlete runs" (Table 2.1). As we can see in the table, each strategy ends up building a different subset of possible intermediate results.

The delayed modifier approach is less sensitive to the particularities of the grammar rules than the sealed index approach. Furthermore, it has the advantage of delaying the proliferation of edges. In other words, we begin by generating a relatively small number of intermediary structures to get a syntactically complete sentence which we can then modify at our leisure. Explicitly separating "syntactically necessary" operations from "semantically necessary" ones (insertion of intersective modifiers) might be useful, because it lends itself more straightforwardly to alternate strategies for inserting these modifiers. For example, as Carroll *et al.* suggests, if the order of modifiers is fixed, we can avoid building the 2^k subsets and just apply them in sequence. Or more exotically, we could also use modifier insertion to satisfy pragmatic constraints,

that is, building a complete sentence and then adding in enough modifiers to uniquely refer to some object in a discourse context [Striegnitz, 2001].

On the other hand, the delayed modifiers approach requires some rather fundamental changes to the surface realiser. First, realisation now occurs in two distinct phases, one which builds up the core syntactic structures and one which inserts the modifiers. Second, we need to change the tree traversal strategy to support a notion of adjoining structures into an already built tree. As we will see in the next chapter, the TAG formalism already supports adjunction natively (which makes TAG potentially attractive for surface realisation). Delayed modifier insertion is the approach which is used in this thesis (Chapter 4) for dealing with intersective modifiers. For completeness, we now discuss two other approaches to the same problem.

2.4.3 Logical form chunking

[White, 2004] proposes a more flexible variant of the sealed index technique. It requires that the grammar write a small set of rules to chunk the input semantics into sub-problems that are solved independently and then combined. For example, the input

$$def(a), young(a), tall(a), polish(a), athlete(a),$$
$$slow(e), eat(e), arg1(e, j), arg2(e, d),$$
$$indef(d), tasty(d), steamed(d), dumpling(d)$$

might be chunked into three distinct problems, allowing us to separately build up "the tall Polish athlete", "a tasty steamed dumpling" and "...slowly eats...". (This is following the default rule, which chunks sub-trees of the input logical form).

As White points out, this approach is less automatic than Kay's original proposal. It "require[s] the insight of the grammar author" to write chunking rules. On the other hand, it is a lot more flexible and can, depending on the chunking rules, account for more cases than the original proposal.

White has also considered the delayed modifier approach, but reports that it is not entirely clear how such an approach would fit into surface realisers with an anytime search strategy (such as OpenCCG). The idea behind anytime search is to combine a best-first disambiguation strategy[6] with the possibility of timing out. If the timeout is reached before the realiser finishes, it returns the best result it has. Presumably, the problem with delayed modification is that the realiser is forced to compute all the syntactically complete structures before starting to add modifiers, which might cause it not to return a semantically complete result before the timeout.

2.4.4 Index accessibility filtering

Index accessibility filtering is another variant of the sealed index approach [Carroll and Oepen, 2005]. It is more flexible than the original approach, fully automated (cf. logical form chunking) and more efficient than delayed modifier insertion in practice.

[6] "Best" being defined by an n-gram language model

```
1:  function COMBINE(i1, i2)
2:      x.fs ← combine-fs(i1, i2)
3:      x.sem ← i1.sem ∪ i2.sem
4:      x.accessible ← i1.accessible ∪ i2.accessible
5:      if is-active(x) then
6:          add-to-agenda(x)
7:      else
8:          old-accessible ← x.accessible
9:          new-accessible ← collect-semantic-vars(x.fs)
10:         inaccessible ← old-accessible \ new-accessible
11:         missing-sem ← input-semantics \ x.sem
12:         missing-indices ← get-indices(missing-sem)
13:         if missing-indices ∩ inaccessible = ∅ then
14:             x.accessible ← new-accessible
15:             add-to-agenda(x)
16:         else
17:             discard(x)
18:         end if
19:     end if
20: end function
```

Figure 2.3: Index accessibility filtering

The approach is a slight generalisation of the original sealed indices. It assumes that there is an operation `collect-semantic-vars` which traverses a feature structure and returns all the available semantic variables.[7] Each chart item stores a set of "accessible" indices, which are manipulated as follows (see Figure 2.3) for more details:[8]

1. When two chart items are combined, the accessible indices of the resulting item is the union of that of the original items.

2. When the resulting chart item is *inactive*:

   a) The accessible sets are recalculated via `collect-semantic-vars`. This purges the accessible set of any indices which became unavailable (see footnote 7) as a result of the chart item becoming inactive. We say that the purged indices are "newly inaccessible".

   b) If there are any literals in the input semantics which are not yet covered by the chart item, and whose semantics are newly inaccessible, the chart item is discarded.

Index accessibility filtering can be seen an incremental improvement to the three other methods that have been proposed so far as a means of dealing

[7] In HPSG, according to John Carroll (personal communication), this can be defined as any variable that is not in the DTRS feature.

[8] Our presentation incorporates some simplifications made by Carroll and Oepen after the publication of [Carroll and Oepen, 2005].

with intersective modifiers. Unfortunately, the more general problem originally pointed out in [Carroll *et al.*, 1999] remains unsolved, i.e. that semantic indices need to be available and thus accessible at all times. The useless combinations shown in Section 2.4.2 would still be produced under index accessibility filtering. But at the very least, intersective modifiers and a few more general cases are taken into account (for example, negation [Carroll and Oepen, 2005]).

The main advantage of this technique is that it requires no manual intervention on the part of the grammar writer. Delayed modifier insertion requires that intersective modifiers are explicitly defined in the grammar. Logical form chunking requires the use of manually written chunking rules. Index accessibility filtering needs no such thing.

2.5 Summary of flat semantics

Flat semantics implement recursion through indirection, by pointing to other semantic expressions instead of including them. Using a flat semantics allow us to live albeit uneasily with the logical-form equivalence problem. We can use them to cheaply compute the logical equivalences which are problematic in practice, and otherwise leave the problem unsolved. Allowing the commutativity and associativity of conjunction introduces a new problem, intersective modifiers, for which several solutions have been proposed. This thesis adopts the delayed modifier insertion approach. The semantic representation language used in this thesis is L_U. As we shall see in the next chapter, this formalism can be used in conjunction with a Feature-Based Tree Adjoining Grammar.

Chapter 3

Tree Adjoining Grammar

It is now widely accepted that natural languages cannot be described by context free grammar, and that something more powerful is needed instead. How powerful exactly is not as clear, although from a computational standpoint, it would be attractive for a grammar formalism to only be as powerful as needed. For example, an unrestricted formalism that can model a Turing machine, could certainly characterise natural languages, but then parsing with such formalisms may be undecidable.

Tree-Adjoining Grammar (TAG) is a formalism that generates the "mildly context sensitive" class of languages. Briefly, it means that it can generate certain languages that context free grammars cannot, although it cannot generate every context-sensitive language. This means, for example, that TAG can describe the language $a^n b^n c^n$ (the set of strings consisting of some as followed by the same number of bs and then of cs), but it cannot describe the language $a^n b^n c^n d^n e^e$. TAG, in other words, is more powerful than CFG, but it is not *too* powerful (it can still be parsed in polynomial time, $\mathcal{O}(n^6)$, to be precise).[1]

We will be using FB-LTAG, a variant which retains the formal properties of TAG. In this chapter, we present the formalism as it is used by our surface realiser. We will start in the next section with the core formalism and add in the two extensions which give us FB-LTAG. In Section 3.2, we present the notion of TAG derivation, which is central to the formalism and which will be useful for discussing generation systems that use TAG. Another pertinent aspect (Section 3.3) is how we link TAG, a syntactic formalism, with L_U, the semantic formalism we presented in the previous chapter. Finally, at the end of this chapter, we will wrap up by highlighting the key properties of TAG that make it particularly relevant to generation, especially in light of the issues we saw in Chapters 1 and 2.

3.1 From TAG to FB-LTAG

The purpose of this chapter is not to cover the exact formal properties of TAG, as these are described elsewhere (for example, [Joshi and Schabes, 1997]) and not especially relevant to this thesis. What we hope to achieve is instead a

[1] On the other hand, whether it is powerful *enough* is another question. There are cases where TAG can provide unsatisfactory analyses, and extensions like MC-TAG have been developed to address these, but these are outside the scope of this thesis.

description of the formalism which is minimal yet adequate for understanding the surface realiser on which my research is built. We begin with TAG and work our way up through FB-TAG and LTAG. FB-LTAG is simply a straightforward combination of the latter two extensions.

3.1.1 TAG - The core formalism

As defined in [Joshi and Schabes, 1997], 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 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.

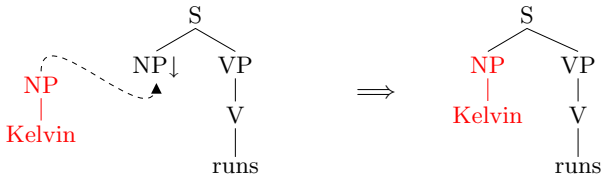


Figure 3.1: TAG substitution

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 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

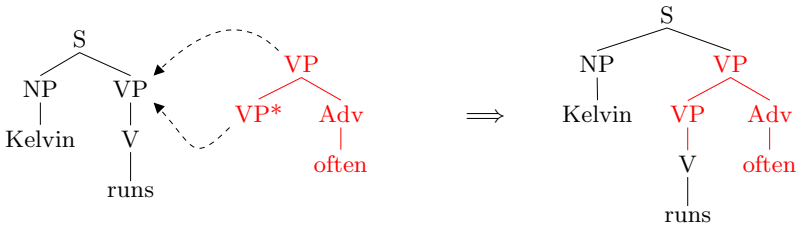


Figure 3.2: TAG adjunction

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.

3.1.2 Adjunction constraints

As we saw above, the adjunction operation has certain restrictions: we cannot adjoin onto substitution nodes, nor can we adjoin into a node whose category does not match (the category of the root/foot node of) the auxiliary tree being adjoined. Sometimes, it is convenient for linguistic description to refine these restrictions with a set of constraints on adjunction. Each tree node in a TAG $\langle \Sigma, NT, I, A, S \rangle$ may be associated with one of these three constraints:

Obligatory Adjunction $OA(T)$ Adjunction on the node is mandatory and the auxiliary tree being adjoined must be a member of the set $T \subset A$.

Selective Adjunction $SA(T)$ Adjunction on the node is optional, but the auxiliary tree being adjoined must be a member of the set $T \subset A$.

Null Adjunction NA Adjunction on the node is forbidden. This is a very commonly used shorthand for $SA(\emptyset)$.

These constraints are an extension to the core formalism, and for that matter, so is substitution. Strip away adjunction constraints and substitution nodes and we will have the original TAG defined in [Joshi *et al.*, 1975]. The version of TAG we use, FB-LTAG has two additional extensions, (non-recursive) feature structures and lexicalisation.

3.1.3 FB-TAG - Feature Based TAG

Feature structures are useful for expressing constraints such as person and number agreement. We use an extension to TAG which adds feature structures and unification to the formalism [Vijay-Shanker and Joshi, 1988].

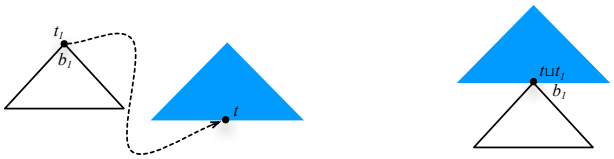


Figure 3.3: Substitution with features

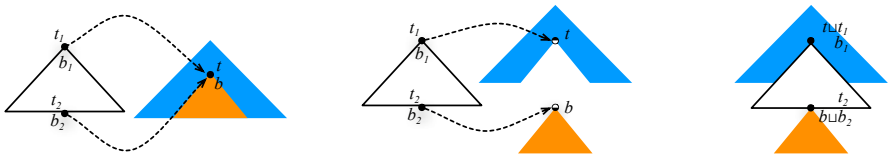


Figure 3.4: Adjunction with features

Each node is associated with two feature structures, *top* and *bottom*[2]. The substitution and adjunction operations are modified to unify feature structures if possible, and to fail otherwise:

substitution The *top* feature structure of the substitution site is unified with the *top* feature structure of the root node of the tree being substituted.

adjunction The *top* feature structure of the adjunction site is unified with the *top* feature structure of the auxiliary tree's root node. The *bottom* feature structure of the adjunction site is unified with the *bottom* feature structure of the auxiliary tree foot node.

The intuition is that the *top* feature can be used to relate a node with its parent. Likewise, the *bottom* feature can relate a node with its children. This can be used to create a sort of chain, percolating features up from *bottom* to *top* of a node to *bottom* of the node above and so forth. When we adjoin something into this chain, we peel apart the *bottom* and *top* features of a node, and insert the auxiliary tree in between.

Three details about unification

It is worth noting that the feature structures used in FB-TAG are non-recursive, that is none of the feature values may themselves be feature structures. This means that the feature structures are bounded by some constant size and we do not exceed the generative capacity of standard TAG.

Also, the scope of the unification variables is over the entire elementary tree, so if the value X appears in different nodes of the same tree, it refers to the same variable (by convention, any value that begins with ? is a variable)

[2]Except for terminal nodes or substitution nodes, which only get a *top* feature

Finally, the construction of a FB-TAG derived tree now occurs in two phases. The first phase is the construction proper of the derived tree, that is the substitution and adjunction of elementary trees into a single structure. The second phase consists of a validation step, where the *top* and *bottom* features of each node are unified. If unification fails, the derived tree is invalid.

Feature structures and adjunction constraints

Note that feature structures can also be used in place of adjunction constraints. We can enforce obligatory adjunction by deliberately inserting conflicting top and bottom features into a node. If an auxiliary tree is adjoined into that node, the conflicting features are pulled apart and thus rendered inert. But if we neglect to adjoin something we will have the expected unification failure. Enforcing selective adjunction is somewhat simpler: we insert features that will only unify with the desired auxiliary tree(s). For null adjunction, we could insert features that no auxiliary trees possess, although in practice we prefer just to use a null adjunction constraint instead.

3.1.4 LTAG - Lexicalised TAG

Lexicalised grammars are those for which each elementary structure is associated with a lexical anchor. Following [Schabes *et al.*, 1988], a grammar formalism is lexicalised if it consists of (i) a finite set of structures to be associated with lexical items, usually the heads of those structures (ii) one or more operations for composing these structures.

Such grammars are known to have two major advantages with respect to non-lexicalised ones. The first is linguistic: syntactic preferences sometimes depend on the lexical items. For example, some verbs take one argument ("sleeps"), and others take more. Associating these lexical items with syntactic structure allows us to express these preferences in a natural manner. The second is computational: lexicalised grammars simplify parsing. They make it possible to break the task down into two distinct phases

1. where we select all the elementary structures (i.e. lexical items) that correspond to the input and

2. where we combine the selected items together.

Every structure corresponds to a piece of the input, and so it can only be used once. In other words, parsing with a lexicalised grammar is decidable; there are no problems with recursion and non-termination, because once we "consume" a piece of the input, we consume the elementary structure that goes with it. This can also be true for surface realisation. As we saw in the previous chapter, recursion and non-termination are perennial problems, but if we can somehow "lexicalise" our grammars from a semantic point of view, such that each elementary structure is associated with a "piece" of the semantics we can have the same guarantees of decidability.

It turns out that TAG lends itself rather well to lexicalisation. A Lexicalised Tree Adjoining Grammar (LTAG) is a TAG for which at least one node in an elementary tree is a terminal node. For example, in Figure 3.5, the elementary trees on the left are LTAG trees (they have an anchor or more each),

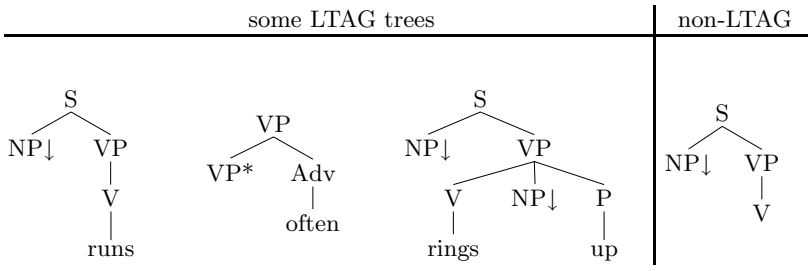


Figure 3.5: LTAG and non-LTAG trees

whereas the ones on the right are not. (L)TAG[3] is a particularly attractive choice because (i) it has what is known as an extended domain of locality and (ii) it allows one to cleanly separate recursion from the core structures of the grammar. This has useful consequences from a linguistic point of view, which include a straightforward treatment of long distance dependencies, semi-frozen expressions, compound words and exceptions [Kroch and Joshi, 1985; Abeillé, 1990]. These linguistic applications of TAG are outside the scope of this thesis. Setting aside these applications, the same basic properties which make TAG attractive from a linguistic standpoint also make it particularly relevant for generation. We will discuss them in greater detail in Section 3.4.

In this thesis, we will be using FB-LTAG, a straightforward combination of FB-TAG feature structures with LTAG lexicalisation.

3.2 TAG Derivations

A derived tree is the result of combining a set of elementary trees together by substitution or adjunction. Sometimes, a derived tree by itself does not give us enough information. For example, in Figures 3.6 and 3.7, we see a small grammar and the derived tree for the sentence "Yesterday, John kicked the bucket". This sentence could have either been built from the tree α_{kicked}, or from the idiomatic $\alpha_{kickedbucket}$. For a given derived tree, it would be useful to know (i) what elementary trees it is made of (ii) and how they were put together. This is represented with a TAG derivation tree, a sort of blueprint for derived trees. Paraphrasing [Joshi and Schabes, 1997]:

1. The root of a derivation tree is labelled by an S-type initial tree (i.e. a tree with root S, for sentences)
2. Every other node is labelled either by an auxiliary tree or an initial tree. It is also associated with a tree address.

Each node label represents an elementary tree used to construct our derived tree. The relationship between these elementary trees is described by

[3]We will henceforth using TAG to refer to TAG-based formalisms like LTAG or FB-TAG in general

the relationship between the corresponding nodes in the derivation tree. That is, if a node *Dtr* is a daughter of a node *Par* in the derivation tree, this tells us that the elementary tree corresponding to *Dtr* was either substituted or adjoined into the one for *Par*. Also, the tree address on node *Dtr* tells us at which location of *Par* the substitution/adjunction took place. One derivation tree corresponds to exactly one derived tree, but as we can see in Figure 3.8, a single derived tree may have more than one derivation possible.

3.2.1 Multiple vs. embedded adjunction

Now that we have a notion of TAG derivation, it is worth pointing out that multiple adjunctions on to a single node are *not* allowed. Borrowing an example from [Scheffler, 2003], the grammar in Figure 3.9 has two modifiers, "spicy" and "red", which could potentially adjoin onto a noun, "pepper". In TAG, there is technically no way to adjoin both of these modifiers on to the noun. We could *embed* the adjunctions so that "spicy" is adjoined on to "red", and the resulting derived tree onto "pepper".

If we had allowed for multiple adjunctions (Figure 3.10), we could have adjoined them both on directly onto the noun. The results look very similar; they both yield the derived tree `NP(N(spicy,N(red,pepper)))`. But their derivations are fundamentally different. Allowing multiple adjunctions onto the same node seems to make more sense from an intuitive standpoint — it is not the redness that is spicy, but the pepper — and perhaps it is useful to be able to distinguish between a "spicy red" pepper and a spicy "red pepper". After all, that is exactly what we do for the literal "John kicks the bucket" and its idiomatic counterpart. Even if we do not care about funny-looking derivation trees (for example, SemFrag does not use the derivation tree for semantic construction, so why worry?), perhaps it is worth considering the syntactic ramifications behind these semantic distinctions (for example, "John kicks the blue bucket" makes no sense as far as the idiomatic flavour of bucket-kicking goes).

We could easily allow for multiple adjunctions,[4] but we choose not to do so because (i) it would not help us to produce any new derived trees and (ii) it could induce a combinatorial explosion when intersective modifiers come into play.[5] Multiple adjunctions remain forbidden for now, but should be looked into as part of future work.

3.3 FB-LTAG augmented with L_U flat semantics

One of our goals is to use an FB-LTAG grammar in a reversible grammar, for both parsing and generation; however, this would not be possible without first introducing a compositional semantics into the formalism. In this section, we

[4] It is not in the standard definition of derivation because it would introduce an ambiguity in the interpretation of derivation trees. But as [Schabes and Shieber, 1994] show, as long as the derivation trees contain a partial order — if there are two adjunctions to the same node, one is specified to occur before the other — allowing for multiple adjunctions is equivalent to the standard TAG.

[5] The problem is that we cannot forbid embedded adjunctions, so for each intersective modifier, we would have a choice of either multiple or embedded adjunction as a means of inserting that modifier.

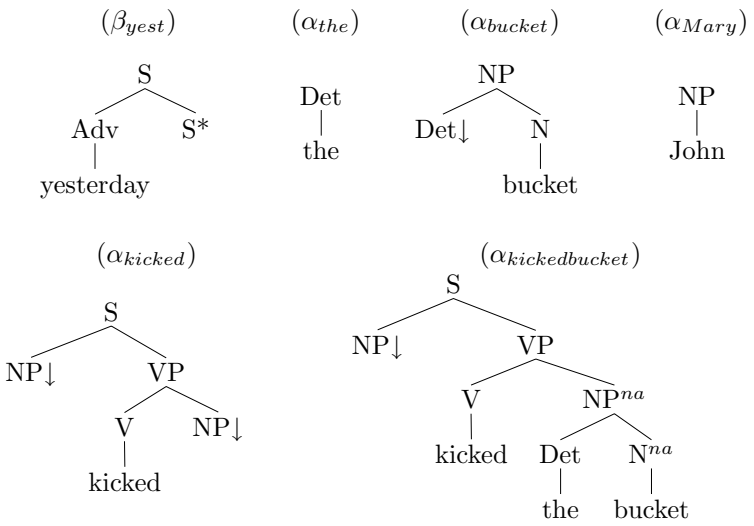


Figure 3.6: Pieces of "Yesterday, John kicked the bucket"

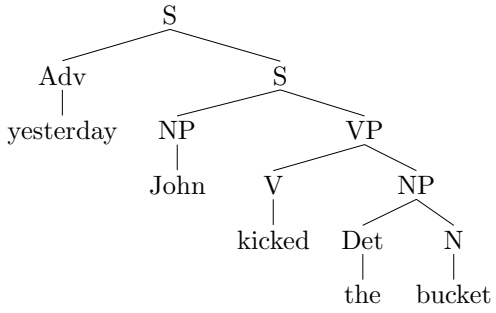


Figure 3.7: "Yesterday, John kicked the bucket"

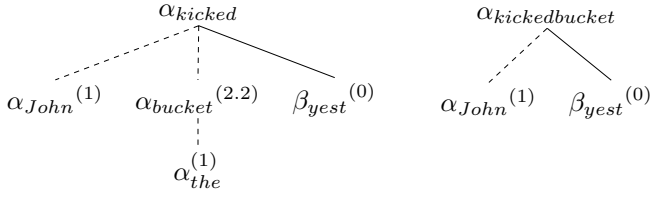


Figure 3.8: Two derivations for one bucket

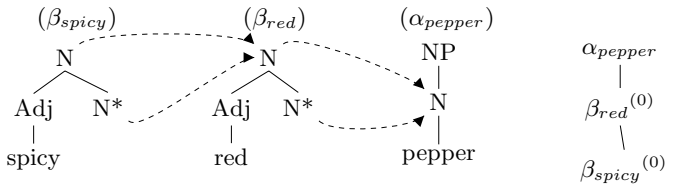


Figure 3.9: "spicy red pepper" with embedded adjunctions

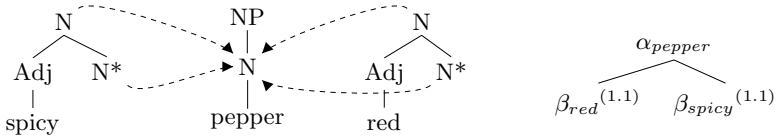


Figure 3.10: "spicy red pepper" with multiple adjunctions

will see that the FB-LTAG can be augmented [Gardent and Kallmeyer, 2003] with an L_U flat semantics (L_U stands for underspecified logic, see Chapter 2). The modification incurs a slight increase in the generative capacity of the formalism and also requires the introduction of set union into the derivational machinery of FB-LTAG. The basic idea is that

1. Every elementary tree is associated with an L_U formula, i.e., its semantics. To recap, an L_U formula is a set of literals, where each literal $R(i_1, \ldots, i_n)$ consists of a relation R over some number of unification variables (or constants) i_1 to i_n.[6]

2. Some nodes of every elementary tree are decorated with unification variables from its L_U formula. These are typically the root node, foot node, any substitution nodes, and any nodes that can be adjoined to. As for the decorations, these consist of an *index* feature (abbreviated *idx*) with a unification variable or a constant for its value. Some nodes are also associated with a similar *label* feature, the purposes of which will be illustrated below.

3. The semantics of a derived tree is understood to be the union of the semantics of all the elementary trees combined modulo the unification performed during derivation. The union operation does not affect FB-LTAG derivation in any way, and can be seen as a trivial post-processing step.

Below, we will see that the semantics of each tree can be encoded as a non-recursive feature structure; however, for purposes of presentation, we will write the semantics of each elementary tree separately, as with the example grammar in Figure 3.11.

[6]The first variable is the label, so we write $R(i_1, i_2 \ldots, i_n)$ as $i_1 : R(i_2, \ldots, i_n)$

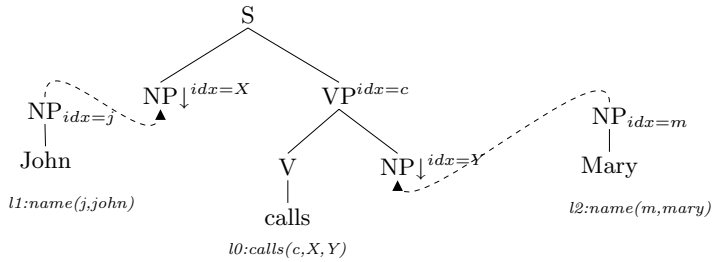


Figure 3.11: Pieces of "John calls Mary"

In other words, the construction of a semantic formula consists in identifying L_U variables with constants (via standard FB-LTAG unification, #2 above), and taking the union of the elementary tree semantics (#3). For example, when we substitute (the trees for) "John" into "calls", the corresponding NP node has the feature-value pair $idx = X$ in its *top* feature and $idx = j$ in its *bottom* feature. In FB-LTAG, the *top* and *bottom* features of every node must be unified at the end of derivation, so in the end, the unification variable X will be replaced by the constant j, and likewise Y with m. Taking the union of the unified semantic formula gives us the semantics of the derived tree (Figure 3.12), *l1:name(j,john), l0:calls(c,j,m), l2:name(m,mary).*

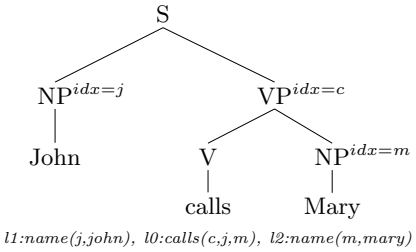


Figure 3.12: Completed "John calls Mary"

It is worth noting that the decoration of nodes with unification variables from the L_U formula, is vital to the correct propagation of semantic indices. For example, if the unification variables in the NP substitution nodes had been swapped, the semantics would instead be *l1:name(j,john), l0:calls(c,m,j), l2:name(m,mary)*, i.e., that of the converse sentence, "Mary calls John". So, for parsing, the unification variables associated with nodes ensure that the correct semantic formula is built; syntax controls the semantics.

For generation, unification variables are used in reverse; semantics controls syntax. The difference is that in generation, the elementary trees are associated with a known semantics, so the L_U unification variables have all been *pre-instantiated* with constants. Figure 3.13 shows a set of elementary trees that might be used for generation. It looks almost identical to Figure 3.11, the only difference being that the variables X and Y are replaced by j and m,

respectively. We would never be able to substitute "Mary" into the left the substitution node, because a unification failure would ensue. In other words, we would never produce "Mary loves John" when we really mean to say "John loves Mary".

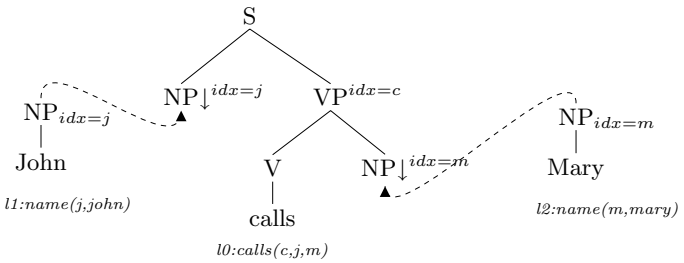


Figure 3.13: Pieces of "John calls Mary" in generation

3.3.1 Derived trees and underspecification

The syntax/semantics interface relies on unification variables shared between the elementary trees and the semantic formulas. Now let us see this interface at work with a more complicated example, this time using L_U underspecification. In Chapter 2, we had briefly alluded to the fact that an L_U formula can leave quantifier scope underspecified if need be. For example, the sentence "Every dog chases a cat" could be associated with the semantics

$$
\begin{array}{ll}
\multicolumn{2}{c}{l0{:}chases(d,c),} \\
l1{:}dog(d), & l3{:}cat(c), \\
l5{:}\forall(d,h1,h2), & l6{:}\exists(c,h3,h4), \\
h1 \geq l1, & h3 \geq l3, \\
h2 \geq l0, & h4 \geq l0
\end{array}
$$

The idea here behind these formulas is that the literals $h1 \geq l1$ represent scope constraints. The formulas can be "plugged", replacing the hole constants $h1$ and $h2$ by labels. The two different pluggings for the above formula and their readings are (19a) and (19b).

(19) a. Every dog chases a cat (each)
l0:chases(d,c), l1:dog(d), l3:cat(c), l5:∀(d,l1,l6), l6:∃(c,l3,l0)

b. Every dog chases a cat (a specific cat)
l0:chases(d,c), l1:dog(d), l3:cat(c), l5:∀(d,l1,l0), l6:∃(c,l3,l5)

Here, we are not interested in the plugging mechanism (see [Gardent and Kallmeyer, 2003] for details). We are more interested in the syntax/semantics interface, building up the underspecified formula itself with an FB-LTAG grammar. As far as the grammar is concerned, there is nothing exceptional about scope constraint literals like $h1 \geq l1$. Here, we write them using infix notation, but this is just a convenience. Otherwise, they are just normal, run of the mill literals.

The mechanism here has been developed for TAG grammars that treat quantifiers as adjuncts. In the French grammar of [Abeillé, 2002], particularly, a noun is added to a verb by substitution, and a quantifying determiner is added to the noun by adjunction, as in Figure 3.14.

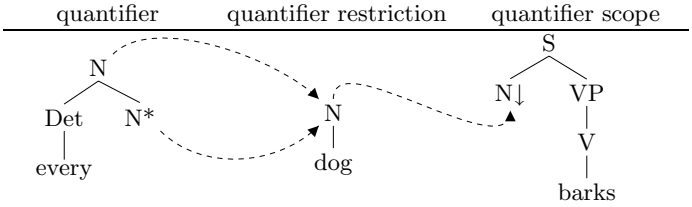


Figure 3.14: "Every dog barks"

Here we would associate the quantifying determiner "every" with the formula

$$l0{:}\forall(X,h1,h2),\ h1{\geq}R,\ h2{\geq}S$$

which captures the relation expressed between the denotation of its nominal argument (R, the quantifier restriction) and that of some external verbal argument (S, the quantifier scope). Our intention here is to ensure that the quantifier scope and restriction are associated with the correct labels. The two ideas here are that

1. We keep track of a *label* feature in our TAG elementary trees in addition to the *index*. We will abbreviate this by omitting the feature names, that is writing simply x, l to stand for $index = x, label = l$.

2. The label variables are divided between the *top* feature of the root node (scope) and the *bottom* feature of the foot node (restriction). We see this for "every" in Figure 3.15, with the S label in the root and the R label in the foot. Likewise, we place the noun label (*l1*) in the *bottom* of its root node, and the verb label (*l2*) in the *top* of the relevant substitution node. This strategic separation of the labels ensures that the quantifier scope argument is unified with the verb label (S with *l2*), and the quantifier restriction with the noun label (R with *l1*).

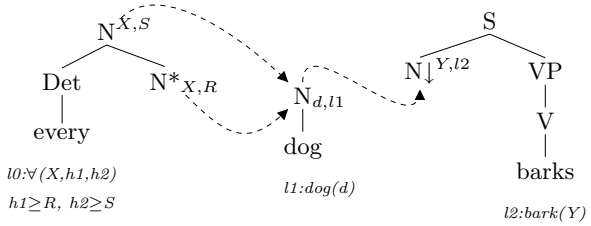


Figure 3.15: "Every dog barks" with semantics

By using a *label* feature and by carefully dividing them into *top* and *bottom* features, we can model the "has-scope-over" ($\geq$) relations that make quantifiers

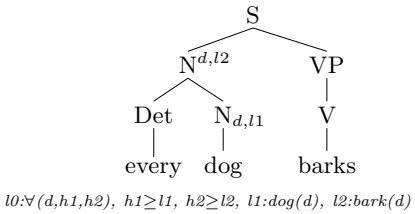


Figure 3.16: "Every dog barks" assembled

work in an L_U semantics. Now what about scope ambiguities? These are a straightforward consequence of this mechanism. In a sentence like "Every dog chases a cat", each determiner is independently adjoined into its respective noun, "every" into "dog" and "a" into "cat". The quantifier scope arguments of both determiners will both have scope over the labels of their nouns, and their quantifier restriction arguments will have scope over the label of "chases", yielding the underspecified formula we saw at the beginning of this section (*l1:dog(d), l3:cat(c), l5:∀(d,h1,h2), l6:∃(c,h3,h4), h1 ≥ l1, h3 ≥ l3, h2 ≥ l0, h4 ≥ l0*).

Note that in this example, the approach has a side-effect of enforcing an obligatory adjunction into "dog"; otherwise, there would be a mismatch on the *label* feature during *top* and *bottom* unification. In this particular situation, the constraint is acceptable, because we would want to forbid the lack of a determiner ("dog barks").

3.3.2 Semantics as feature structures

There are two ways that one could associate an L_U semantics with TAG elementary trees.

- We could treat the elementary tree and semantics as separate structures, i.e. manipulating tree/formula pairs instead of just trees.
- Alternately, we could encode the semantics into FB-LTAG feature structures.

The second way is conceptually more elegant; it avoids the extra requirements of the first way, either (a) extending TAG substitution and adjunction to ensure that the appropriate variable substitutions take place in the semantics during unification or of (b) adding a post-processing step to a parser, which "replays" the appropriate unifications during semantic construction.

The feature structure encoding is a variant of the one proposed in [Parmentier, 2007].[7] Basically, we add a set of features to the parent node of the LTAG anchor. The purpose behind this placement is to ensure that the features are never unified with anything else. The features added would be from a set of

[7] Slightly modified to emphasise that we do not need to introduce recursive feature structures to perform the encoding.

distinguished semantic features, that is, for each literal, *pred* for the relation and a feature *label* for its label, and arg_n for each of its other arguments. So as we can see in Figure 3.17, the semantics *l0:calls(c,j,m)* would be encoded as the feature structure:

$$[label = l0, pred = calls, arg_1 = c, arg_2 = j, arg_3 = m]$$

This scheme can also be made to work with elementary trees that have more than one literal in its semantic formula. We would simply use the features $pred^x$, arg_0^x and arg_n^x for each of its literals. For example, one could imagine a variant of the semantics *l0:calls(c,j,m)*, where we unbundled the thematic roles into *l0:calls(c), l0:agent(c,j), l0:patient(c,m)*. This longer semantic formula could be expressed as the feature structure:

$$\begin{bmatrix} label^1 = l0, & pred^1 = calls, & arg_1^1 = c, & \\ label^2 = l0, & pred^2 = agent, & arg_1^2 = c, & arg_2^2 = j, \\ label^3 = l0, & pred^3 = patient, & arg_1^3 = c, & arg_2^3 = m \end{bmatrix}$$

For parsing, exploiting this semantics would require a post-processing step that trivially reinterprets the *pred* and *arg* features as L_U semantic formulas, and takes the union of all semantic formulas found in the derived tree. For generation, it would require a *pre*-processing step that performs the encoding. Neither of these steps affects the actual derivation.

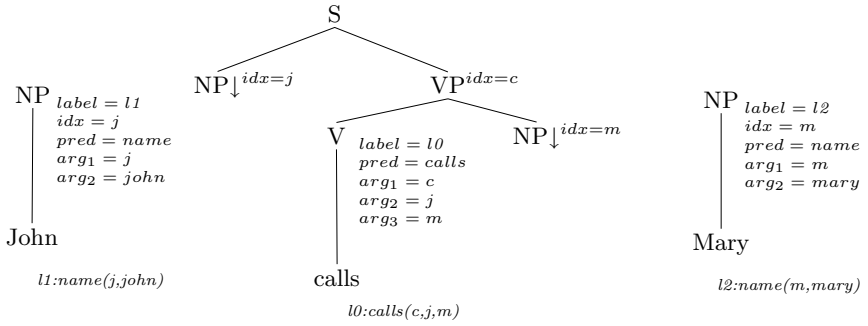


Figure 3.17: Encoding semantics as feature structures

L_U and generative capacity [Kallmeyer and Romero, 2004] has remarked that the L_U approach to TAG semantics changes the generative capacity of the formalism (on the grounds that feature structures are no longer finite as a result). The culprit is the use of semantic labels and variables as possible feature values. The consequences of this increased generative capacity seem to be limited for parsers and surface realisers. Consider how the grammar is to be used by an FB-LTAG parser (or surface realiser): we select an elementary tree for each part of the input string, instantiating its semantics as we go along, and then we combine the trees we have selected. At this point in the process, the number of feature structures we can have is now finite, because here there are only a limited number of semantic variables to go around! Each word is going to be associated with some maximum n semantic variables, and so

the number of feature structures is related to $n \times w$, w being the size of the input. (Note however, that the size of feature structures is now proportional to the size of the input, and no longer constant). Given these considerations, we do not know how the increased generative capacity actually plays out for putting L_U-augmented FB-LTAG grammars to use. Alas, such considerations do not change the fact that the set of languages that could be recognised by L_U-augmented FB-LTAG is not the same as for its unaugmented cousin.

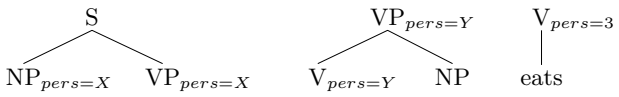
3.4 Generation with TAG

An (L_U-augmented) TAG grammar is a *reversible* resource, a pool of linguistic knowledge that can be used for either parsing or generation. Though the lion's share of TAG research has gone into parsing, researchers have long observed that the formalism is particularly suitable for generation in a way that CFG-based formalisms are not [Joshi, 1987; McDonald and Pustejovsky, 1985]. In this section, we will see why this is the case. Also, in the subsequent chapters of this thesis, we will discuss some TAG (and other) generation systems, especially comparing them with GenI, the surface realiser that we use. In the meantime, let us have a closer look at the theoretical relevance of TAG to generation.

3.4.1 Extended domain of locality

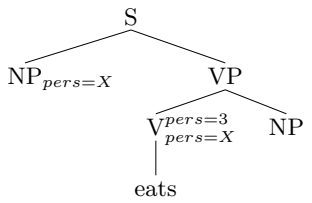
Informally speaking, a domain of locality in a grammar is where things in the grammar "go together"; it is where one can specify constituency, constraints, word-order and unifications [Joshi, 1987]. All grammar formalisms have a domain of locality. Context free grammar (and CFG-derived formalisms like HPSG and LFG) have a limited domain of locality because their trees[8] only have a depth of 1. In contrast, TAG elementary trees can have a depth greater than 1, and so the formalism can be said to have an extended domain of locality.

This has benefits both practical and theoretical. From a practical standpoint, it means that we can avoid littering grammar rules with features that serve only to propagate constraints or to simulate locality in other ways. For example, consider this small context free grammar augmented with feature structures. To handle subject-verb agreement, we are forced to thread the *pers* feature from one rule to the next:



Enforcing these constraints is more straightforward with a TAG. We define just a single elementary tree that expresses exactly the desired constraints:

[8] We can think of context-free grammar rules as trees, the left hand side being the root and the right hand side being the leaves. So the rule `S → NP VP` can be seen as a tree `S(NP VP)`



For generation in particular, the extended domain of locality can be helpful in dealing with words that have an empty semantics, for example, the complementiser "that" or infinitival "to". These words are to surface realisation what gaps (or empty categories) are to parsing. In a naive approach, they require that all trees with an empty semantics be considered as potential constituent candidate at each combining step. In terms of efficiency, this roughly means increasing the size of the input by n (just like postulating gaps at all position in an input string increases the size of that string). To avoid this shortcoming, a common practice [Carroll *et al.*, 1999] consists in specifying a set of rules which select empty semantic items on the basis of the input literals. However these rules fail to reflect the fact that empty semantic items are usually functional words and hence governed by syntactic rather than semantic constraints. With TAG, we simply invoke the extended domain of locality, treating the empty semantic words as co-anchors.

3.4.2 Factoring recursion from domain of dependencies

An extended domain of locality can make for more convenient feature percolation and more efficient handling of empty semantic items. So then, what are the theoretical benefits? To begin with, it permits us to adopt the linguistic convention that each TAG elementary tree corresponds to a predicate and its arguments. On the previous page, we saw an example of an LTAG tree, `S(NP↓,VP(V(ring), NP↓,P(up)))`. This tree acts as a single, self-contained linguistic unit, the verb "to ring up" as well as its arguments, a subject and direct object. It is not possible to express it in the same way with a CFG because those trees cannot be made deep enough. This is not just a matter of practical grammar engineering; it is important because it means that TAG trees can be linguistically *meaningful* units.

Then again, tidy linguistic units are of little use if we could not modify them in any way. A context free equivalent to the "ring up" tree would presumably be a set of trees/rules. It may not be as elegant or linguistically motivated, but at least it allows for adjuncts. For example, we could imagine a hypothetical CFG rule like `VP` → `VP frequently` which lets us build sentences like "John rings Mary up frequently". In order for the tight linguistic packages to be useful, TAG needs to provide a mechanism for inserting modifiers into strings. This is where adjunction comes in. In TAG, modifiers are represented as auxiliary trees, and these auxiliary trees can be spliced into other trees. Adjunction means that we truly exploit the extended domain of locality (to have self-contained linguistically motivated units), whilst still accounting for modifiers. Furthermore, when using adjunction, we still preserve the constraints and the predicate-argument structure originally expressed by the trees we adjoin into. Put another way, TAG adjunction allows us to factor out recursion "from the

domain over which dependencies are initially stated" [Joshi and Schabes, 1997]. So it is not just that it allows for modifiers, but does so in a way that preserves the original advantages of the extended domain of locality.

Coming back to generation, the factoring out of recursion has the consequence that when we have closed all the substitution nodes of an elementary tree, the resulting derived tree is syntactically complete and can stand alone. This has two benefits for TAG surface realisers. First, it means that we can flexibly interleave planning with realisation. Since auxiliary trees are syntactically optional, the sentence planner could flexibly decide either to include an auxiliary tree in one sentence, or perhaps save it for the next [Joshi, 1987]. Indeed, as we will see in the next chapter, some generation systems, G-TAG, SPUD and INDIGEN exploit this capability in one way or another [Danlos, 1998; Stone and Doran, 1997; Striegnitz, 2004]. Second, it allows us to treat substitution and adjunction as separate phases of derivation. This helps us to deal with the intersective modifiers problem (Section 2.4) as we will see in the next chapter.

Chapter 4

GenI and SemFraG

GenI is a surface realiser, originally developed by Carlos Areces and Claire Gardent [Areces, 2003]. It uses an FB-LTAG grammar with an L_U semantics and a bottom-up chart generation algorithm. In this particular chapter, we present the basic algorithm, developed with the INRIA ARC GENI.[1] SemFraG is an FB-LTAG for French with an L_U semantics [Gardent, 2006]. It has been used for both parsing with semantic construction [Parmentier, 2007] and for surface realisation with GenI. Most of the research into the realiser has revolved around the development of this grammar.

Here we discuss the original version[2]. This chapter has three basic sections; we present the surface realiser in Section 4.1, the grammar in Section 4.2 and some related TAG generation work in Section 4.3. This chapter also concludes the first part of this thesis. In the following chapters, we shall present a series of extensions to the realiser. Here, we focus on the basic algorithm.

4.1 GenI

GenI uses a lexicalised grammar. To take advantage of this, it uses two broad phases: lexical selection, which returns a set of elementary trees, and tree assembly, which combines the selected trees in reasonable ways. We will discuss these two phases in greater detail here, but first here is an overview of the algorithm according to the three facets of realisation algorithms presented in Chapter 1.

Traversal In its traversal of the derived tree, GenI is head-driven, but this is merely a consequence of using TAG. The TAG formalism separates the notion of a derived tree from a derivation tree. One actually can think of tree traversal strategies in terms of the derivation tree. This is perhaps a more useful way to look at the algorithm. In its traversal of the *derivation* tree, GenI is bottom-up; it starts out with a set of lexically selected items and finds out how to combine them into successively larger structures. Figure 4.1 shows in what sense tree

[1] GENI is an *Action de Recherche Concertée* (ARC), a joint research project bringing together several laboratories together on a single topic, in this case, Generation and Inference. See `http://www.loria.fr/projets/geni`

[2] More or less the original version; the 2003 version has a design bug, namely the lack of an auxiliary agenda.

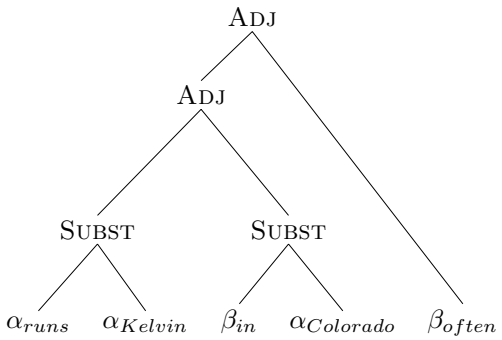


Figure 4.1: GENI does bottom-up traversal of the derivation tree

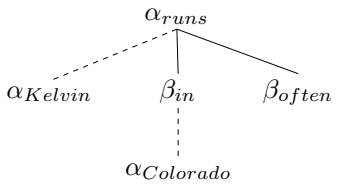


Figure 4.2: Traditional representation of the derivation tree in 4.1

traversal is bottom-up for GENI (Figure 4.2 shows the same derivation tree using the traditional notation). The key here is that we start from the lexical items and build up.

Search GENI performs an exhaustive search. In other words, it attempts to return all results and does not prune away any part of the search space that would cause a dead end (no commitment, in the terminology of Section 1.2.3). Incidentally, it uses a depth-first search because our agenda works as a stack. This choice should be considered an implementation detail — consing onto the agenda is cheaper than concatenating — and does not make a difference anyway because our search is exhaustive.

Tabulation GENI implements a rudimentary form of chart generation with neither subtree sharing[3] nor packing (this is not for any theoretical reason, just the lack of time). Table 4.1 has the complete list of chart generation features so far.

[3] One might conceivably argue that GENI does effectively have sharing, because it "copies" subtrees by just pointing to them, but the validity of this argument will need to be checked more carefully.

core tabulation	yes
structure sharing	no
packing	no
indexing	maybe
partial recognition	yes
tabulated prediction	no
agenda based control	yes

Table 4.1: Chart generation in GenI is rudimentary

We can say that it does some form of indexing, in that it verifies that the semantics of two chart items do not overlap before attempting to combine them.

It has some notion of partial recognition (dotted edges), in that it performs its TAG operations in an (arbitrarily) fixed order. In other words, it visits the substitution nodes of a tree in a strict left-to-right order, not attempting to plug anything into a node unless the ones on its left are already closed. Imposing an order on these nodes is sensible because it eliminates spurious permutations of filling one node before the other. On the other hand, we cannot really say that it performs tabulated prediction à la Earley. Say for example, that we have the following lexically selected items to combine:

- τ_{says} - $\mathtt{S}_e\mathtt{(NP}_s\downarrow\mathtt{, says, S}_{e1}\downarrow\mathtt{)}$
- τ_{runs} - $\mathtt{S}_e\mathtt{1(NP}_k\downarrow\mathtt{, runs)}$
- τ_{Serena} - $\mathtt{NP}_s\mathtt{(Serena)}$
- τ_{Kelvin} - $\mathtt{NP}_k\mathtt{(Kelvin)}$

The sentence that results from their combination would be "Serena says that Kelvin runs". We can say that GenI uses partial recognition in the sense that it would not attempt to substitute τ_{runs} into τ_{says} until its NP_s node is also plugged with τ_{Serena}. This results in fewer chart items. Nevertheless, we cannot say that it does tabulated prediction because it attempts to combine τ_{Kelvin} into τ_{runs} without knowing if it would be useful to do so. That said, it is not entirely clear if tabulated prediction is all that useful in the context of exhaustive search.

4.1.1 Lexical selection

Now, on to the two phases of GenI. For a parser, lexical selection might consist of collecting the elementary trees that correspond to the words of the input sentence. This is similar for realisation except that we collect the elementary trees that correspond to literals of the input semantics, more precisely those trees whose semantics subsumes part of the input semantics.

The input to lexical selection is thus (i) a flat input semantics and (ii) an FB-LTAG grammar. The grammar consists of a set of lexical items consisting of an elementary tree and a lexical semantics. More precisely:

Given an input semantics **sem** and lexicon L:		
Axioms	$[\tau; s; rn; (n_1, \ldots, n_x)]$	$\langle \tau, s \rangle \in lexselection(\mathbf{sem}, L)$, rn is the root node of τ, $(n_1, \ldots, n_x)$ are the subst nodes of τ
Goals	$[\tau; s; rn; ()]$	
Inference rules	$\dfrac{[\tau_p; s_p; rn_p; (n_{p1}, \ldots, n_{px})] \quad [\tau_c; s_c; rn_c; (n_{c1}, \ldots, n_{cy})]}{[\tau_{pc}; s_p \cup s_c; rn_p; (n_{p2}, \ldots, n_{px}, n_{c1}, \ldots, n_{cy})]}$	$rn_c = n_{p1}$, $s_b \cap s_c = \emptyset$, $\tau_{pc} = subst(\tau_c, \tau_p)$ (Sub)

Table 4.2: GENI substitution phase (modulo unification)

Definition 3 (lexical item)**.** A lexical item consists of a pair $\langle T, S \rangle$, where T is an LTAG elementary tree (Section 3.1.4) and S is an L_U formula (Section 2.1). Unification variables in the lexical item are understood to have scope over the whole item, that is T and S.

Lexical selection is described in greater detail in Appendix D. We basically retrieve any lexical item whose semantics is non-empty and can be unified with some part of the input semantics. As we saw in Section 3.3, the unification variables in the lexical semantics are shared with the associated elementary tree. Via the unification of the (subsumed part of the) input and lexical semantics, semantic indices from the input are propagated into the feature structures of the elementary tree. Because the input semantics is saturated, the instantiated lexical item also has a saturated semantics.

4.1.2 Tree assembly (chart generation)

After the lexical selection phase, we move into tree assembly. The input to this phase is the semantic formula we wish to realise, and the set of lexically selected elementary trees. Our objective is to find every derived tree that can be built from these parts and which is (i) syntactically complete, meaning it has no empty substitution sites and (ii) semantically complete, meaning that its semantics exactly matches the input semantics. Ultimately, the goal of realisation is to produce a set of strings, but this basically consists of concatenating the leaves of each derived tree.

To cope with the problem of intersective modifiers (Section 2.4), the algorithm uses the delayed modification strategy of [Carroll *et al.*, 1999]. Realisation occurs in two chart generation phases, a substitution phase (where only TAG substitutions are performed), and an adjunction phase (likewise, TAG adjunctions only). Separating forces the realiser to only insert modifiers into syntactically complete structures. It also happens to be a very natural strategy to use with TAG because adjunction and auxiliary trees are part and parcel of the formalism. This could be seen as a potential advantage for TAG as a generation formalism [Gardent and Kow, 2005].

Given an input semantics **sem**, and goal edge items GS from the previous phase	
Axioms	$[\tau; s; fn; (n_1, \ldots, n_x)]$ $\langle \tau, s \rangle \in GS$, $(n_1, \ldots, n_x)$ are adjoinable nodes of τ fn is the foot node of τ (or $-$ if initial)
Goals	$[\tau; \mathbf{sem}; -; (n_1, \ldots, n_x)]$
Inference rules	$\dfrac{[\tau_p; s_p; -; (n_{p1}, \ldots, n_{px})] \quad [\tau_c; s_c; fn_c; (n_{c1}, \ldots, n_{cy})]}{[\tau_{pc}; s_p \cup s_c; rn_p; (n_{p2}, \ldots, n_{px}, n_{c1}, \ldots, n_{cy})]}$ $fn_c = n_{p1}$, $s_b \cap s_c = \emptyset$ $\tau_{pc} = adj(\tau_c, \tau_p)$ (Adj)

Table 4.3: GENI adjunction phase (modulo unification)

We can think of the two phases as two successive chart generation problems fed to a generic deductive parser (see Tables 4.2 and 4.3, as well as Appendix C for the version with unification taken into account, and Appendix D for pseudocode). For clarity, here is a more procedural description of the algorithm. We are using the usual agenda based control (Section 1.3.3.7), where the agenda is a data structure for storing unprocessed intermediate results. For convenience, we introduce a new data structure, called an auxiliary agenda. The auxiliary agenda is used to set aside any syntactically complete auxiliary trees found during the first phase of realisation. It is not strictly necessary, but it saves the trouble of filtering the chart afterwards. Here are the two phases in detail:

Substitution phase

1. Store the lexically selected trees in the agenda, except for auxiliary trees which are devoid of substitution nodes (put these in the auxiliary agenda).

2. Loop: Retrieve a tree from the agenda, add it to the chart and try to combine it by substitution with trees present in the chart. Add any resulting derived tree to the agenda. Stop when the agenda is empty.

Adjunction phase

3. Move the chart trees to the agenda and the auxiliary agenda trees on to the chart. Discard all trees which are not syntactically complete (i.e. that still have open substitution nodes), as they will never become complete in this phase.

4. Loop: 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.

The inner workings of this algorithm might be better illustrated by an example or two:

Kelvin runs often

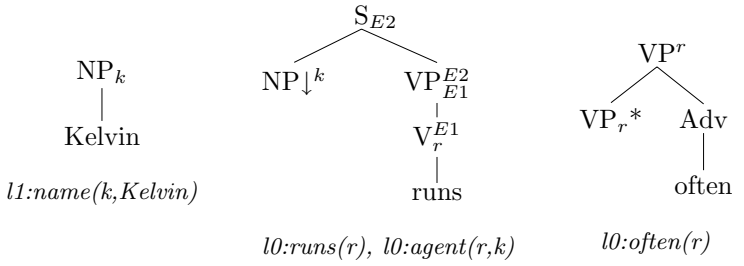


Figure 4.3: Kelvin runs often

Let us start with a simple one. Suppose that the input semantics is *l0:run(r), l0:agent(r,k), l0:often(r), l1:name(k,Kelvin).* Lexical selection gives us a set of elementary trees lexicalised with the words "Kelvin", "often", "runs" (Figure 4.3). The trees for "Kelvin" and "runs" are placed on the agenda, the one for "often" is placed on the auxiliary agenda.

We begin with the substitution phase, which consists of systematically exploring the possibility of combining two trees by substitution. Note that in the table below, the letters 'k', 'r' and 'o' stand for the elementary trees that correspond to "Kelvin", "runs" and "often", respectively. When the trees combine, we write, for example, 'kr' to mean a derived tree for "Kelvin runs". We see that here, the tree for "Kelvin" is substituted into the one for "runs", and the resulting derived tree for "Kelvin runs" is placed on the agenda. Trees on the agenda are processed one by one in this fashion, although in this simple example, there is only one substitution to be done. When the agenda is empty, indicating that all combinations have been tried, we prepare for the next phase.

Combination	Agenda	Chart	AgendaA
	k,r		o
	r,	k	o
↓(r,k)	kr	r,k	o
		r,k,kr	o

All items containing an open substitution node are erased from the chart (here, the tree anchored by "runs") as there is no hope that they will be made complete in this phase. 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 "Kelvin runs often".

Combination	Agenda	Chart	Results
	k,kr	o	
	kr	o	
⋆(kr,o)	kro	o	
		o	kro

Note that the chart never changes during the adjunction phase. This is because we do not store new items onto the chart. As each tree is removed from the agenda, either we notice that it is semantically complete and save it as a result, or we perform all adjunction operations involving (i) the tree and (ii) a tree from the chart, and put any resulting derived trees back onto the agenda. The tree itself is no longer of any use and may be discarded. We will see more implications of this below, when we have more than one auxiliary tree that can adjoin into the same node.

Kelvin runs often in Colorado

Now, a slightly more complicated example. Here, we shall see auxiliary tree with substitution nodes, as well as a lexical selection which leads to more than one result. Our new input semantics is *l1:name(k,Kelvin), l0:often(r), l0:run(r,k), l0:in(r,i), l0:place(i,Colorado)*, basically the same as before with two new literals. The lexically selected trees (Figure 4.4) are the same as before plus trees for "in" and "Colorado". The trees for "in", "Colorado", "Kelvin" and "runs" are placed on the agenda, whereas the tree for "often" is placed on the auxiliary agenda.

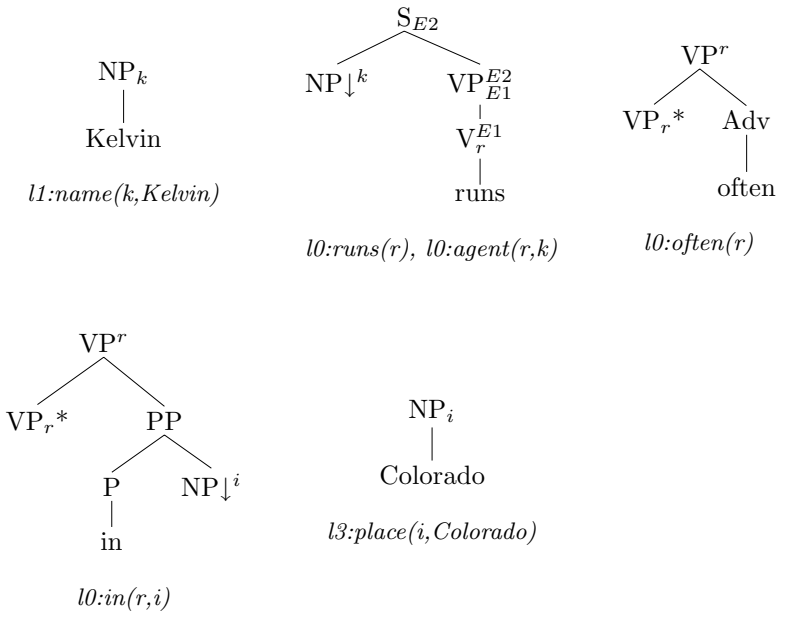


Figure 4.4: Kelvin runs often in Colorado

Again, the substitution phase explores all possible substitutions, of which there are two: "Kelvin" into "runs" to get "Kelvin runs", and "Colorado" into "in" to get "in Colorado". The case of "in Colorado" is particularly interesting because it is a syntactically complete auxiliary tree `V(V*,PP(P(in),NP(Colorado)))`, in the sense that it has no open substitution sites. When we produce such trees, we transfer them to the auxiliary agenda, so that they can be used during the adjunction phase.

Combination	Agenda	Chart	AgendaA
	i,c,k,r		o
	c,k,r	i	o
↓(i,c)	ic,k,r	i,c	o
	k,r	i,c	o,ic
	r	i,c,k	o,ic
↓(r,k)	kr	i,c,k,r	o,ic
		i,c,k,r,kr	o,ic

As before, all items containing an empty substitution node are erased from the chart (here, the trees anchored by "runs" and by "in"). The agenda is then reinitialised to the content of the chart and the chart to the content of the auxiliary agenda (here "often" and "in Colorado"). The **adjunction phase** proceeds, performing all possible adjunctions. 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 sentences "Kelvin runs often in Colorado" and "Kelvin runs in Colorado often".

Combination	Agenda	Chart	Results
	c,k,kr	o, ic	
	k,kr	o, ic	
	kr	o, ic	
⋆(kr,o), ⋆(kr,ic)	kro,kric	o, ic	
⋆(kro,ic)	kroic,kric	o, ic	
	kric	o, ic	kroic
⋆(kric,o)	krico	o, ic	kroic
		o, ic	kroic, krico

Here, we get more than one result because trees can combine in different ways (we could also get more than one result if we had an ambiguous lexical selection, but this is not the case here). Note, by the way, that the combinations here are a result of embedded adjunctions and not multiple adjunctions. The fact that GenI supports the former but not the latter is a technical detail, and affects the number of outputs it produces (allowing for multiple adjunctions introduces more output since we cannot forbid embedded adjunctions), as well as the derivation trees for the output.

4.2 SemFraG

SemFraG is a core grammar for French. It combines an FB-LTAG syntax with an L_U semantics. The aim of SemFraG is to serve as a paraphrastic grammar, one that associates distinct grammatical realisations with the same essential meaning to a single logical form. For example, the sentences "Jean aime Marie" and "Marie est aimée par Jean" ("John loves Mary" and "Mary is loved by John") would have the same semantic representation (*l1:aimer(e), l1:agent(e,j), l1:patient(e,m)*). To get an idea of the paraphrastic power, we

built a test suite with over 80 cases. It produced a total of 1582 sentences with an average (mean) of 18 paraphrases per case.

4.2.1 Factorised representation

Note that in theory, we did not distinguish between a lexicon and a grammar; the FB-LTAG grammar and lexicon are one and the same. In practice, we do make such a distinction, which helps us to avoid the redundancy that would ensue otherwise. For example, the current SemFRaG lexicon has 150 verbs, typically with 5 conjugations each (median and mean; 41 maximum). To simplify, we say that they can be used in 170 different syntactic contexts.[4] This multiplies out $150 \times 5 \times 170 = 127\ 500$ possible verb trees. We use the three standard factorisations from the XTAG system [XTAG Research Group, 2001].

Morphological lexicon First, we separate the morphological information from the rest of the grammar. The morphological lexicon associates the inflected form of a word with its lemma and morphological features, whereas the syntactic lexicon associates a lemma with its elementary trees. In the example above, this factorisation alone would leave us with $150 \times 170 = 25\ 500$ trees.

Tree schemata The syntactic lexicon can be further separated into a lexicon proper and a set of tree schemata. Thus an entry in the lexicon consists of a lexical semantics, a lemma, and the name of a tree schema (or as we will see in the next paragraph, a set of tree schemata). A tree schema is almost an LTAG elementary tree, the sole difference being that (one of) its terminal node has been chopped off. The parent of the erstwhile terminal node is called the anchor and tells us where the terminal node was meant to be. To reassemble an FB-LTAG lexical item, we take an entry from the syntactic lexicon and its tree schema, and plug the lemma into the anchor node. This first factorisation saves us from repeatedly describing the elementary trees. This does not change the number of our entries in the lexicon (we would still have 25 500), but makes each entry considerably smaller.

Tree families This is where the third factorisation comes in. We group the schemata into a set of tree families, typically on the basis of predicate argument structure (see Appendix A for the full list of families). This way, entries in the syntactic lexicon need only point to the tree family and be associated to all the tree macros within. Revisiting our example yet again, we find ourselves with just 150 entries in the syntactic lexicon.

4.2.2 Metagrammar

Continuing the idea of factorisation further, it is worth noting that the tree schemata are themselves built from a factorised representation, called a metagrammar. This avoids redundancy in building and maintaining the grammar, as there can be many tree schemata (6000 in the current grammar[5]) with a

[4] SemFRaG has 172 trees in the `n0Vn1` family; clearly, not all verbs are in that family, but it is fairly typical.

[5] Earlier we said that each lemma is associated with 170 schemata, but we didn't say that it was the same 170 schemata each time!

fair amount of redundancy between them. This aspect of the (meta)grammar becomes relevant in Chapters 6 and 7, so we will discuss it in greater detail over there.

4.3 Related NLG systems for TAG

There is a large number of generation systems that use TAG. Some of these we will discuss in the second part of this thesis, when the context is appropriate. Here, we will provide a brief overview of generation systems which use TAG.

4.3.1 Surface realisers

Generation as dependency parsing

[Koller and Striegnitz, 2002] also perform TAG surface realisation from a flat semantics. Their approach consists in translating the surface realisation problem into a constraint satisfaction one (actually, as a dependency parsing problem, which can be solved using constraint based approaches). We will discuss their approach further in Section 5.6.3 as it is related to our contributions there.

SmartKom realiser

The SmartKom realiser (henceforth SKGen) [Scheffler, 2003] also uses FB-LTAG and a flat semantics (MRS). Like [Koller and Striegnitz, 2002], SKGen translates surface realisation into a constraint satisfaction problem. The slight difference is that Scheffler provides a direct translation (doing away with the indirection of TDG parsing). Curiously, the realiser itself was not implemented with a generic constraint solver, but by hand. The implementation uses top-down traversal (of the derivation tree), depth-first search (plus backtracking) and no tabulation.[6]

FERGUS: statistical supertagging

Fergus is a TAG surface realiser of the "overgenerate-and-select" variety [Bangalore and Rambow, 2000b]. It uses statistical methods to perform two tasks, supertagging and the ordering of adjuncts. Unlike GenI, it uses full commitment, taking the one best lexical selection and the one best ordering (respectively) for each stage. The use of statistical methods in the second phase is relevant to GenI because it could supplement our current tactic for dealing with intersective modifiers. We postpone the insertion of modifiers until the last minute, but we still have to insert them anyway at some point. If we could impose an order on the modifiers, there would not be an intersective modifiers problem. Ideally, the ordering would come from linguistic knowledge, but identifying this order might be difficult (linguistically subtle) and outside the scope of a core grammar like SemFraG. It seems that for this particular case, statistical methods could be used without causing any meaningful loss in paraphrastic power.

[6] Scheffler writes that SKGen traverses the semantic graph depth-first, which sounds like a top-down traversal of the derivation tree, but we could be mistaken.

ISOFT: Backing Systemic Grammar with a TAG

ISOFT is one of two TAG realisers which combines a Systemic Functional Grammar (SFG) with TAG [Yang, 1992]. SFG is not so much a grammar formalism as an approach to linguistics (Systemic Functional Linguistics), one which focuses not on phrase structure, but on the function (the purpose) of linguistic choices [Halliday, 1978]. SFG are well-suited for generation because they provide a mapping from higher level abstract input (the generator's goals) to linguistic realisation; however, the implementation of the linguistic back-end has traditionally left much to be desired. For example, in KPML (a popular surface realiser and SFG implementation), the linguistic back end consists of simple string concatenation and ordering rules, without any notion of splicing one string into another. ISOFT attempts to make up for this by using a TAG as the linguistic back-end of the SFG. This system is relevant to our work on paraphrase selection and will be discussed in greater detail there (Section 6.5, Page 142).

4.3.2 Beyond surface realisation

TAG is also used in generation systems that perform surface realisation along with other tasks. The key difference between GenI and these systems is that the former focuses exclusively on the realisation task.

MUMBLE

MUMBLE is one of the earlier, if not *the* earliest generation systems for TAG [McDonald and Pustejovsky, 1985]. Like ISOFT, it incorporates a Systemic Functional Grammar. Unlike ISOFT, it treats generation as involving "three temporally intermingled activities": determining the goals of the utterance, text planning and realisation.

G-TAG: Text planning and multi-sentence generation

G-TAG is an extension to LTAG developed primarily as a generation formalism [Danlos, 1998]. One key feature of G-TAG is that it accounts for multi-sentential texts, which it achieves by (i) explicitly allowing for elementary trees that combine sentences into "texts" and (ii) introducing a semantic-conceptual layer which sits atop the syntax-semantic layer that is TAG.

The main job of G-TAG is text and sentence planning, tasks which are typically performed "before" surface realisation. G-TAG also includes a surface realisation module, but it is not the main focus of the system. The realiser converts a g-derivation tree produced by the system into a g-derived tree. It then linearises the g-derived tree and performs morphological generation. The g-derived tree can stand for a set of surface variants, in the sense that the linearisation module can optionally use a set of rewriting rules to convert from one variant to another. Alternatively, the linearisation module could choose to leave the tree intact, and just read the leaves, in which case, the canonical surface form is produced.

SPUD: Sentence Planning Using Description

SPUD is a sentence planner with a surface realisation component [Stone *et al.*, 2003]. The input to SPUD is a set of (modal first-order logic) formulas, called updates. The output is a communicative intent, basically a TAG derivation tree accompanied by semantic and pragmatic information. This communicative intent can be fed into an external "surface realiser," which converts the TAG tree into a string and performs morphological generation.

What makes SPUD particularly interesting is that various sentence planning tasks and surface realisation are performed together. Starting from a communicative goal, SPUD simultaneously discovers the semantics of the expression it is trying to build whilst building up the actual syntactic structure. This approach addresses an issue with generation pipelines, where decisions near the beginning of the pipeline fail to take into account information near the end of the line. Bad decisions early on in the pipeline could result in a lot of needless backtracking. Systems like G-TAG and SPUD may be more efficient in the grand scheme of things, because they allow constraints on several levels to be taken into account at the same time.

InDiGen: SPUD with charts

InDiGen can be thought of as an improvement to SPUD. In terms of search strategy, SPUD uses a greedy search with full commitment. Greedy search means that SPUD always makes the locally optimal choice, which may or may not lead to the globally optimal result. Full commitment means that once it makes its local decisions, it discards all alternatives, which eliminates any possibility of backtracking. InDiGen replaces the full-commitment strategy with a no-commitment one. To avoid the backtracking that would ensue, it uses chart-generation to cache the intermediate results [Striegnitz, 2004].

InDiGen is the most direct ancestor to GenI. The ideas that GenI borrows are (i) returning all possible paraphrases, taking advantage of the chart and (ii) the two-phase strategy described in Section 4.1.2. That said, InDiGen does have a different take on realisation than GenI. The input consists of a discourse context (world knowledge and conversational record), and a communicative goal of the form *describe X (communicating I)*. The X is semantic entity, an index like *e1*, and the I is akin to the GenI input semantics. The key difference is that InDiGen does not try to match the input semantics exactly, but to produce a sentence whose semantics may be a superset of the input. More precisely, the adjunction phase has the job of adding in modifiers, until pragmatic constraints associated with the output are satisfied by the discourse context.

Part II

Contributions

Chapter 5

Polarity filtering

There are two sources of complexity in the surface realisation task, lexical ambiguity and the lack of word-order constraints. In the first part of this thesis, we discussed some techniques for dealing with this complexity in general, as well as techniques specifically for managing the lack of word-order constraints (e.g. delayed modification). In this chapter, we focus on lexical ambiguity, the other source of complexity. This chapter proposes a way to constrain the effects of lexical ambiguity. The idea is inspired from "electrostatic tagging" [Perrier, 2003; Bonfante *et al.*, 2003], which consists essentially of pruning the initial search space by applying a global filter on the initial possible combination of lexical items.

In this chapter, we adapt the technique to surface realisation. We begin by discussing the intuitions behind this filter (Section 5.1). Next, we build up the actual algorithm from a barebones version to one which addresses some major shortcomings in the approach (Section 5.2). We follow up on this with a brief aside showing how polarity filtering can be integrated with chart generation (Section 5.3), and with a presentation of some extensions to the algorithm (Section 5.4), most of which have already been implemented. Finally, we present an evaluation of this technique (Section 5.5) and wrap up with a discussion of related work (Section 5.6).

5.1 Polarised intuitions

Polarity filtering is a straightforward adaptation of electrostatic tagging, a symbolic approach to the supertagging task in parsing. Parsing and surface realisation both have versions of the supertagging problem. For parsing, the problem consists in associating each word of the input string with an elementary tree; for realisation, it is literals that have to be associated, not words. Both variants of the task are about dealing with lexical ambiguity. So to begin with, we will now discuss lexical ambiguity and its impact in surface realisation, and having done so, introduce the basic intuition behind polarity filtering.

5.1.1 Lexical ambiguity

In surface realisation, lexical ambiguity comes into play after the lexical selection step. Here, we have retrieved the set of lexical items whose semantics

subsumes (part of) the input semantics. For example, given the semantics:

l0:picture(p), l1:cost(c,p,h), l2:high(h).

A surface realiser and a hypothetical grammar/lexicon might select the following items (see Figure 5.1 on Page 94):

- $\tau_{painting}$ or $\tau_{picture}$ for *l0:picture(p)*;
- τ_{cost} (the noun) or τ_{costs} for *l1:cost(c,p,h)*;
- τ_{high} or $\tau_{a\ lot}$ for *l2:high(h)*.

This selection here is ambiguous in the sense that there is more than one lexical item chosen for any given part of the input semantics. For example, each of the literals[1] in this input semantics are represented by two lexical items each.

Of course, we cannot use *all* of these items to produce a single sentence. To form a sentence, we must first identify some combination of exactly one lexical item for each literal in the input semantics.[2] On the one hand, there is an *a priori* exponential number of such combinations; it is product of the ambiguity for each literal. Given an input semantics with n literals,[3] where each literal is identified by an index i and where there are a_i lexical items for that literal, the number of possible lexical combinations is the product

$$\prod_{1 \leq i \leq n} a_i$$

The reason that lexical ambiguity is a problem is not just that the search space is large, but that in practice, many of the lexical combinations within are useless. They contain some syntactic incompatibility between the items so that there is no possible way of assembling them into a valid result. In the example above, τ_{cost} may be combined with τ_{high} to get "the cost of X is high". It might also be combined with $\tau_{a\ lot}$, but then the resulting "the cost of the a lot" is useless. Likewise, τ_{costs} can combine with $\tau_{a\ lot}$ ("costs a lot"), but not with τ_{high}. Note that the problem of compatible lexical combinations is not necessarily tied to synonymy in the lexicon. In TAG, a lexical item may be anchored to a variety of elementary trees, each tree representing a syntactic variation for that item.

This is the usual cause of ambiguity in the current version of SemFrag (See Appendix A for a list of families and their ambiguities). To get an idea of the effects of this ambiguity, consider the French sentence "Jean avertit Paul que l'ingénieur a une idée intelligente" (Jean warns Paul that the engineer has an intelligent idea), which is produced by one of our test cases. The semantics for this sentence is shown in Table 5.1, along with the ambiguity (that is, the number of lexical items selected) associated to each literal.[4] The ambiguities here come from syntactic variation alone and would be worse if we allowed for synonyms in our lexicon. Multiplying them out gives us $8 \times 10 \times 89 \times 229 =$

[1] For now, let us assume that by "part of the input semantics" refers to a single literal, e.g., *l1:cost(c,p,h)*. We will generalise out of this assumption in Section 5.2.4.

[2] For now, we ignore the issue of lexical items with a multi-literal or empty semantics.

[3] This is only valid if the semantics of each lexical item has only one literal, otherwise, computing the ambiguity is more complicated.

[4] Again, it is slightly simplified to get rid of multi-literal semantics. See Section 5.2.4.

literal	word	ambiguity
l1:jean(j)	Jean	1
l2:paul(p)	Paul	1
l4:le(x)	le	1
l3:ingénieur(x)	ingénieur	8
l7:un(i)	un	1
l5:idée(i)	idée	1
l6:intelligent(i)	intelligent	10
l7:avoir(e1, x, i) i	avoir	89
l8:avertir(e2, j, p, e1)	avertir	229

Table 5.1: An ambiguous lexical selection

1630480 possible lexical combinations, the large majority of which (99.9%) are syntactically incompatible. The task at hand is thus to find the remaining 1728 syntactically plausible combinations.

5.1.2 Approximating validity with polarities

Electrostatic parsing is based on the principle of associating each lexical item with a set of polarities. The polarities act as constraints on how the lexical items may combine. For example, a lexical item with two `-np` polarities can be seen as requiring two NP nodes; whatever it combines with must ultimately counterbalance that negative polarity (for example, two items with a `+np` polarity might work). The polarities can then be used to guide the parser: if a combination of lexical items has a non-neutral polarity, it is necessarily invalid (the inverse is not true; a combination could have a neutral polarity and still be invalid). Crucially, counting the polarities of lexical combinations can be done quickly and cheaply.

In surface realisation, polarity filtering is an intermediary step that sits between lexical selection and surface realisation proper. First we choose the lexical items whose semantics subsumes the input (lexical selection), filter out the lexical combinations with non-neutral polarities, and perform realisation on the remaining items. Making the filter work for surface realisation mostly consists of using flat semantic literals where Perrier *et al.* would use words from the input string. The only caveats are that we have to deal specially with lexical items that have an empty semantics or one with more than one literal in it, which we will ignore until Sections 5.2.4 and 5.2.5.

The filter has two parts. First, we need to assign polarities to lexical items. We can do this offline by augmenting the grammar with polarity annotations (this is following [Bonfante *et al.*, 2004], who adapt electrostatic parsing to LTAG from Interaction Grammar). Basically, for every category *cat*, we award every elementary tree with a $-cat$ polarity for every substitution node of that category and a $+cat$ polarity if its root node is of that category (except for auxiliary trees, for which the root node is ignored). For instance, a tree like `S(NP↓,VP(V(hates),NP↓))` would have the polarities `-np -np +s`. Since we

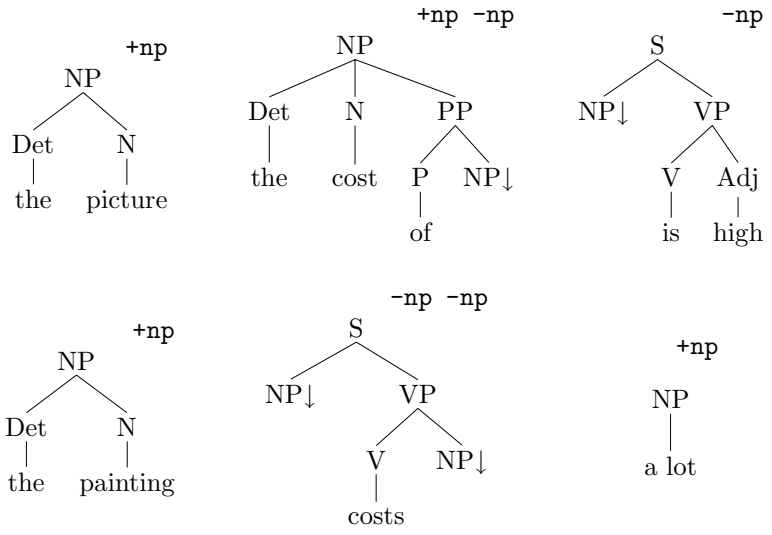


Figure 5.1: Computing the polarity of lexically selected trees

are only developing the intuitions for now, we can simplify matters by focusing on one category only, say `np`. This leaves us with a polarity of `-np -np` for that tree, which we will abbreviate as `-2np` (similarly, `-np +np` can be written `0np`).

Let us see now how polarities would be assigned to a set of lexically selected items. Figure 5.1 shows the selection for the "cost of the painting" example in Section 5.1.1. Their polarities are summed up in the table below. Each column of the table contains a single literal from the input semantics, the lexical items that realise this literal, and their associated polarities.

l0:picture(p)	*l1:cost(c,p,h)*	*l2:high(h)*
$\tau_{picture}$ `+np`	τ_{cost} `0np`	τ_{high} `-np`
$\tau_{painting}$ `+np`	τ_{costs} `-2np`	$\tau_{a\ lot}$ `+np`

The problem at hand is to detect the lexical combinations (i) which cover the entire semantics, (ii) which associate each literal with exactly one lexical item, momentarily ignoring the problem of empty and multi-literal semantics, and (iii) whose aggregate polarity (charge) is zero. The lexical combinations we find in the process may not necessarily be valid, but this is for the surface realiser proper to find out. What polarity filtering helps us to do is to quickly rule out the combinations which are sure to fail. If a lexical combination has a negative charge, it is invalid because there are not enough trees to close off all its substitution nodes. Likewise, if the combination has a positive charge, it has initial trees which remain unused (since each tree corresponds to a piece of the input semantics, we *have* to use all trees, so the combination is invalid).

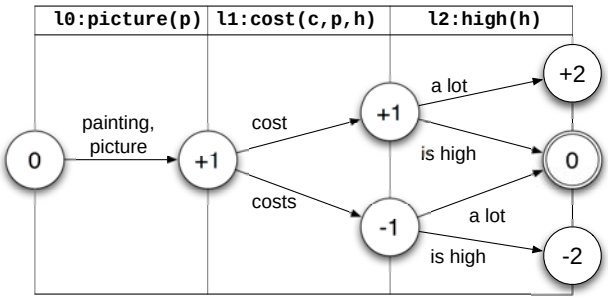


Figure 5.2: A polarity automaton

In Section 5.1.1, we saw that the lexical items τ_{cost} and $\tau_{a\ lot}$ are not a useful combination — we would get "the cost of a lot" which is then syntactically incompatible with "the painting". We can test this using the polarities. If we sum up the polarities of $\tau_{painting}$, τ_{cost} and $\tau_{a\ lot}$, we have a `+2np` charge and reject the combination. In contrast, the combination $\tau_{painting}$, τ_{costs}, $\tau_{a\ lot}$ has a charge of `0np`, which means it would be worthwhile to try using it for surface realisation. This is summed up in the table below, which shows what polarities would get assigned with each combination of lexical items.

`0np`	the cost of the painting/picture is high $\tau_{painting}$, $\tau_{cost\ of}$, $\tau_{is\ high}$ $\tau_{picture}$, $\tau_{cost\ of}$, $\tau_{is\ high}$
`0np`	the painting/picture costs a lot $\tau_{painting}$, τ_{costs}, $\tau_{a\ lot}$ $\tau_{picture}$, τ_{costs}, $\tau_{a\ lot}$
`+2np`	(*) the cost of the painting/picture a lot $\tau_{painting}$, $\tau_{cost\ of}$, $\tau_{a\ lot}$ $\tau_{picture}$, $\tau_{cost\ of}$, $\tau_{a\ lot}$
`-2np`	(*) the painting/picture costs is high $\tau_{painting}$, τ_{costs}, $\tau_{is\ high}$ $\tau_{picture}$, τ_{costs}, $\tau_{is\ high}$

5.1.3 Polarity automaton

A naive approach to computing these polarities would be to do so individually for each lexical combination. Clearly, this is a bad idea because there are exponential number of possible combinations. Since many lexical combinations have lexical items in common we can avoid a great deal of redundant computation. [Perrier, 2003; Bonfante *et al.*, 2003] achieve this by building an automaton that compactly encodes the polarities of all lexical combinations. We can build a polarity automaton for realisation in much the same way.

First let's have a look at the structure of a polarity automaton (Figure 5.2). The automaton is organised in columns which are labelled by some literal. Each

column represents not one literal, but the set of literals by which it and the preceding columns are labelled. Every state in this automaton sits on the right border of some column, except for the initial state which is on the left of the first column. Each state corresponds to a possible net charge for some set of literals; more precisely (i) the literals on the left of the state, i.e., the literals represented by the columns to its left) and (ii) the charge of lexical combinations whose semantics matches that of the columns. Note that there can be more than one state on the right of a column because each combination of literals may result in a different net charge; however, some combinations may converge to the same charge, in which case they are represented with the same state.

Building this automaton is a simple, incremental process, which we have illustrated in Figure 5.3 on the next page. We start out by putting an initial state (charge `0np`) on the left of the first column. We build the automaton left-to-right, column-by-column (literal-by-literal). At each iteration, we look at the states on the left border of the current column and build new transitions to new states on the right border. Each transition that we add corresponds to the lexical choice in the current column. For example, transitioning from the second column, *l1:cost(c,p,h)*, to the third, *l2:high(h)*, we have a choice between the noun "cost" and the verb "costs". The noun has a charge of zero and so it represents a transition from the `+1np` charge in the previous column to another `+1np` in the current one. On the other hand, the verb has `-2np` charge and it causes us to transition from `+1np` to `0np`. If we continue this process until we cover all the literals we will have a compact representation of all lexical combinations and their associated net charges. We declare the final state to be that which covers all literals of the input semantics (i.e. a state in the last column) with a `0np` charge. To get the combination of lexical items without any known syntactic impossibilities, we return the set of paths that go from the starting state to the final one.

5.2 Building polarity automata

We now have a rough sketch of the algorithm for building polarity automata. In the following sections, we provide a more detailed presentation of the algorithm and the various complications we will encounter. First, we provide some more details on how exactly we associate every elementary tree with a set of polarities (Section 5.2.1). We then revisit the simplest version of the algorithm (Section 5.2.2), essentially filling in the details of that we saw in the overview. In the next few sections, we do away with the simplifications we have assumed up to now, adding in multiple polarity keys (Section 5.2.3); lexical items with a multi-literal semantics (Section 5.2.4); and lexical items with a null semantics (Section 5.2.5).

5.2.1 Computing the polarity of each lexical item

The construction of polarity automata requires that each lexical item be associated with a charge. As we mentioned in Section 5.1, it is possible to use more than one label for filtering, for example, `s`, `vp` and `np` instead of just the latter. It is also possible to extend this scheme to attributes other than the

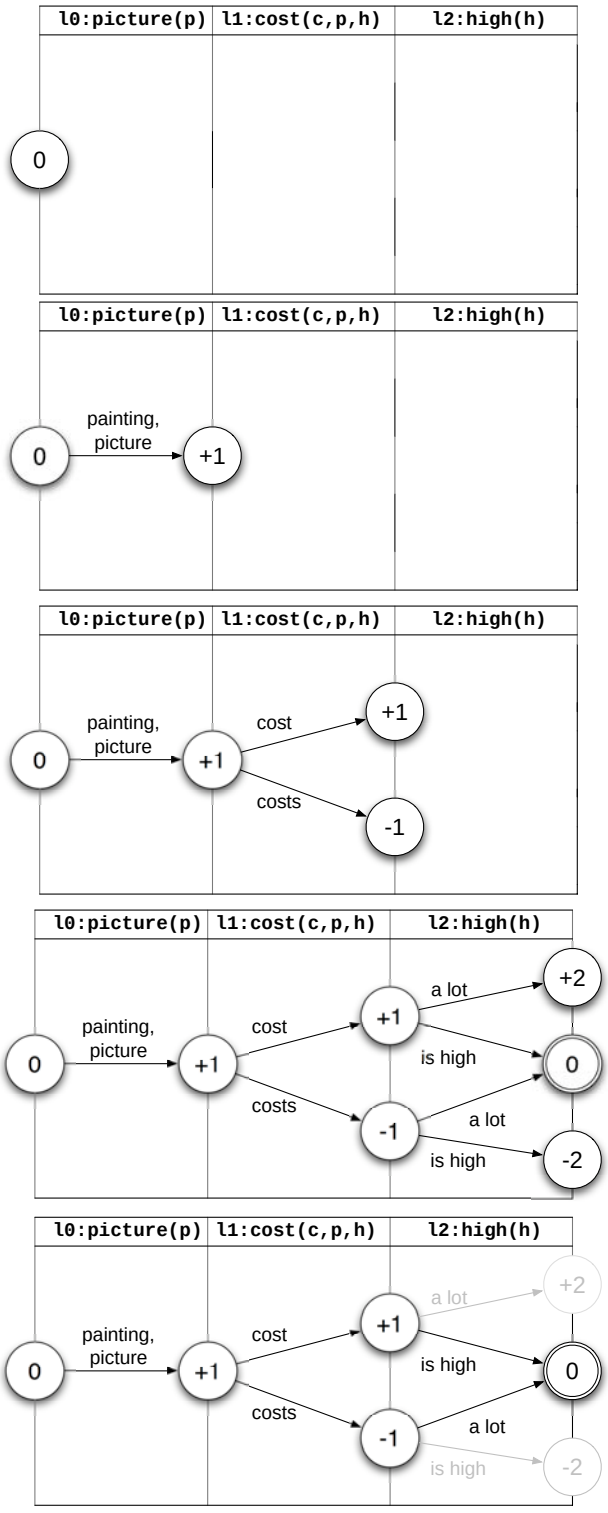


Figure 5.3: Building a polarity automaton

node category. We can thus write the labels as $a{:}k$, where a is some attribute (e.g. `cat`) and k a possible value for that attribute. For convenience, we call these labels polarity keys.

Computing the polarities of a lexical selection happens in two phases; first we compute the set of possible polarity keys and second we compute the charges which are associated with them.

Possible polarities keys

Given a set of lexically selected elementary trees, the first thing we do is to calculate the set of possible polarity keys. This is the set of keys $a{:}k$ where for the top feature of every substitution nodes of every tree, and of the root node of every initial tree, the attribute a is associated with a constant value and k is one of those values.[5] Note that in the SemFRaG grammar, we are currently limited to polarity keys of the form `cat:`C. We had attempted to use semantic indices as well, but we have determined that it is not always linguistically valid for the semantic indices of an elementary tree's root node to be instantiated. Extending the set of polarity keys actually used is a topic for future research, the infrastructure for which is provided by this chapter.

The charge of an elementary tree

Ignoring syntactic structure, we can think of an elementary tree as a set of nodes N. Given a tree N and a polarity key $a:k$, the charge of that tree $pol(N, a{:}k)$ is the sum of charges for each of its nodes.

$$pol(N, a{:}k) = \sum_{n \in N} pol'(n, a{:}k)$$

The rough idea is to assign negative charges to substitution nodes (and foot nodes) and positive charges to root nodes. We only assign such a charge if the node matches the given polarity key. A node matches a polarity key if its top[6] feature structure contains an attribute value pair (a_n, k_n) that matches the key, in other words, if $a_n = a$ and $k_n = k$.

$$pol'(n, a{:}k) = \begin{cases} 1 & \text{if } n \text{ matches } a{:}k \text{ and is the root node of an initial tree} \\ -1 & \text{if } n \text{ matches } a{:}k \text{ and is a substitution node} \\ 0 & \text{otherwise} \end{cases}$$

Note that we ignore any attributes with a non-constant value. This would be incorrect were it not for the fact that we only consider as a possible polarity key any $a{:}k$ where the value associated with a is constant in every substitution or root node of every tree[7]. This certainly limits the number of polarity keys we can use (for example, we cannot use the `idx` attribute in the current state of the SemFRaG grammar), but non-constant values do not have a meaningful

[5] We consider that TAG trees have a `cat:`C feature in both the top and bottom features of every node, where C is the category of that node.

[6] Actually, some attributes tend to be present in the bottom feature of a node; so for the sake of robustness, we have elected to use the *bottom* feature for root nodes (substitution nodes only have a top feature).

[7] Every root node of every initial tree

"polarity", which means that the restriction is a necessary evil. We will present a way to work around this restriction in Section 5.4.2 (due to [Bonfante *et al.*, 2004]), but the part which would be relevant to using indices as polarity keys has not yet been implemented.

5.2.2 Polarity automaton for one key

Now that we have a more precise definition of polarity keys, we can start to build polarity automata. In this section, we provide a definition of a single-key polarity automaton, which lets us make use of a single polarity key $a:k$. Each path in this automaton (that leads to a final state) represents a lexical combination with a neutral polarity for that key. To make use of *multiple* keys, we can build a separate automaton for each key. Each automaton represents a set of syntactically plausible lexical combinations according to a given polarity key. To find the lexical combinations which are syntactically plausible for all the keys, we can take the intersection of the automata using a standard algorithm for the intersection of finite state automata (FSA) [Hopcroft and Ullman, 1979].

Definition 4 (Single-key polarity automaton)**.** A single-key polarity automaton is an acyclic deterministic finite state automaton. It is a tuple $\langle \Sigma, S, s_0, \delta, F \rangle$ where

- Σ is the input alphabet, the set of FB-LTAG elementary trees (with flat semantics).
- S is the set of states, where each state is of the form $\langle l, c \rangle$, with $l \in \mathbb{N}$ and $c \in \mathbb{Z}$. The intended interpretation is that l a literal counter pointing to the lth literal of the input semantics, c is the charge of that state.
- s_0 is the initial state, an element of S
- δ is the state transition function ($\delta : S \times \Sigma \rightarrow S$), is the set of transitions from one literal to its immediate neighbour on the right.
- F is the set of final states.

Given an input semantics L and a polarity key $a:k$, a single-key polarity automaton can be built in the following manner:

1. Σ is the set of lexically selected items. For convenience, we also define the function $\sigma : \mathbb{N} \rightarrow \Sigma$, which given an index l, returns the set of lexically selected items whose semantics consists of the l-th literal in L.
2. S is built in the following way
   a) $\{s_0\}$ is an element of S (see below)
   b) If a state $\langle l, c \rangle$ is an element of S and if $l < |L|$,
      then for every elementary tree $t \in \sigma(l)$,
      the state $\langle l+1, c' \rangle$, with $c' = c + pol(t, a{:}k)$, is also in S
3. s_0 is the tuple $\langle 0, z \rangle$, where z is 0, $a{:}k$ is required to appear in the root node (e.g., $a{:}k$ is `cat:s`), in which case z is -1.

4. δ is defined for all $l < |L|$ and $t \in \sigma(l)$ as:
   $\delta(\langle l, c\rangle, t) = \langle l+1, c'\rangle$ with $c' = c + pol(t, a{:}k)$.

5. F is the singleton set $\{\langle |L|, 0\rangle\}$. If this set is not reachable, the automaton is empty.

Note: since we are only interested in the paths that lead to a final state, polarity automata are always minimised in practice, that is, with any states and transitions which are not on a path to the final state removed. This can be done either with a standard FSA minimisation algorithm [Hopcroft and Ullman, 1979], or with a customised version that works specifically on polarity automata.[8] From now on, we assume that all polarity automata are minimised.

5.2.3 Polarity automaton for n keys

Using multiple polarity keys can be an expensive affair because of the potential cost of computing their intersection. In theory, the intersection of a automata with n states each requires $\mathcal{O}(n^a)$ time/space, because we have to compute the Cartesian product of their states. For polarity automata in particular, computing the intersection is somewhat cheaper, because we need only compute the Cartesian products of all the states $\langle l, c\rangle$ that have a literal index l in common [Le Roux, 2007b]. Nevertheless, the cost is exponential with respect to the number of automata.

To improve efficiency of this in practice, we perform the intersection incrementally, building each successive automaton off the previous one as a sort of "skeleton". It is hoped that, in practice, the intermediary automata are smaller as a result of this incremental process, with each successive automaton encoding fewer and fewer lexical combinations.[9]The approach is as follows: we define an n-key polarity automaton, which effectively encodes the intersection of n polarity automaton. Given a set of x polarity keys, we attempt to build an x-key polarity automaton. We start from a 0-key automaton which represents the lexical selection. Then we iterate through the polarity keys; given a polarity key $a{:}k$ and an n-key automaton, we construct an $(n+1)$-key automaton. In other words, from the 0-key automaton, we build a 1-key automaton, and from that, we build a 2-key automaton and so forth until we have built up an N-key automaton covering all the polarity keys. This is equivalent to building N separate polarity automata and taking their intersection.

Definition 5 (n-key polarity automaton)**.** An n-key polarity automaton is an acyclic deterministic finite state automaton $\langle \Sigma, S, s_0, \delta, F\rangle$, where

- $\Sigma, s_0, \delta,$ and F all have the expected meaning (see Definition 4)
- S is the set of states. Each state is of the form $\langle l, c, s\rangle$, with $l \in \mathbb{N}$ and $c \in \mathbb{Z}$, and $s \in \mathbb{N}$. The intended interpretation is that l is the index of

[8]The minimisation algorithm works "backwards" and recursively: any states $\langle |L|, c\rangle \notin F$ are removed; any state $\langle l, c'\rangle$ which transitions exclusively to removed states (i.e., of the form $\langle l+1, c''\rangle$) is also removed.

[9]In a personal communication, Le Roux points out that taking the intersection of n automata all at once is more time-efficient than taking the intersection of successive pairs of automata, but that it imposes a very large cost in space, as we would have to consider the Cartesian product of n sets of states.

some literal in the input semantics and c is the charge of that state. s is a state counter pointing to a state in an $(n-1)$-key polarity automaton. For convenience, we will treat states and state counters interchangeably.

n-key polarity automata can be built inductively. We start out by building a seed automaton, that is, a 0-key automaton:

1. Σ is the set of lexically selected items.
2. S is built in the following way
   a) $\{s_0\}$ is an element of S (see below)
   b) If a state $\langle l, 0, 0\rangle$ is an element of S and if $l < |L|$, then $\langle l+1, 0, 0\rangle$ is also in S
3. s_0 is the tuple $\langle 0, 0, 0\rangle$
4. δ is defined for all $l < |L|$ and $t \in \Sigma$ as $\delta(\langle l, 0, 0\rangle, t) = \langle l+1, 0, 0\rangle$
5. F is the singleton set $\{\langle |L|, 0, 0\rangle\}$.

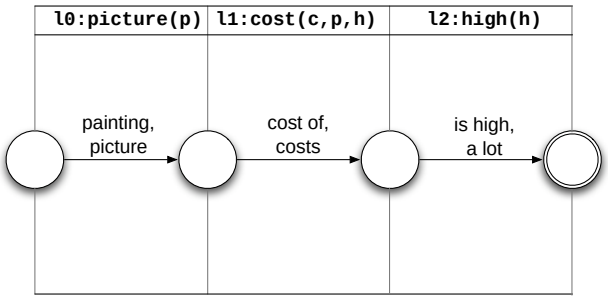


Figure 5.4: A seed automaton

Given an n-key polarity automaton, and a polarity key $a':k'$, we can build an n'-key automaton $\langle \Sigma, S', s'_0, \delta', F'\rangle$ with $n' = n+1$.

We first define a helper function, $next : S' \times \Sigma \rightarrow S'$, to handle the essential bureaucracy of building up states in the new automaton: $next(\langle l, c, s\rangle, t) = \langle l+1, c_2, s_2\rangle$ where $c_2 = c + pol(t, a':k')$ and s_2 is the state $\delta(s, t)$. With this helper out of the way, we can continue our induction step:

6. Σ is the same as before.
7. S' is built in the following way
   a) $\{s'_0\}$ is an element of S (see below)
   b) If a state $s' = \langle l, c', s\rangle$ is in S', and if $l < |L|$: for every t such that $\delta(s, t)$ is defined, $next(s', t)$ is in S'.
8. s'_0 is the tuple $\langle 0, z, s_0\rangle$, where z' is 0, unless $a':k'$ is required to appear in the root node (e.g., $a':k'$ is `cat:s`), in which case z' is -1.
9. δ' is defined for all $l < |L|$. Given a state $s' = \langle l, c', s\rangle$; for every t such that $\delta(s, t)$ is defined, $\delta'(s', t) = next(s', t)$.

10. F' is the set of states such that for every $f \in F$ (i.e. every final state in the previous automaton), $\langle l, 0, f \rangle$ is in F'.

See Figures 5.4 and 5.5 for an example of a seed automaton, an 1-key automaton built off it with `cat:np` as a polarity key.

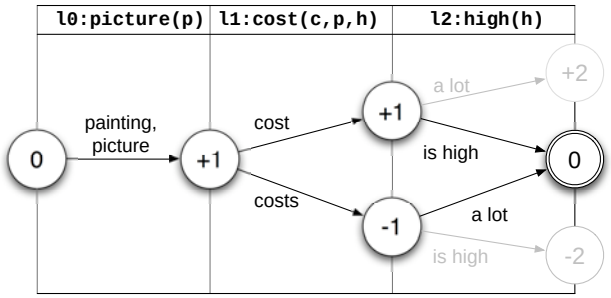


Figure 5.5: A 1-key automaton

5.2.4 Multi-literals semantics

Up to now we have assumed that every lexical item selected corresponds to exactly one literal in the semantics. But not all lexicons are designed in this fashion. To begin with, one might choose to make the thematic roles into separate literals in the semantic representation, thus using a representation more like *l0:love(l), l0:agent(l,j), l0:patient(l,m)* for the word "loves" instead of *l0:love(l,j,m)*.[10] Furthermore, some lexicons may have entries whose semantics contain more than one literal.

We could, for instance, extend the running example ("The cost of the painting is high") by introducing a single lexical entry $\tau_{is\ expensive}$ with the semantics *l1:cost(c,p,h) high(h)*. The lexical selection and its polarities (for a single key `cat:np`) would thus be

l0:picture(p)	*l1:cost(c,p,h)*	*l2:high(h)*
$\tau_{picture}$ `+cat:np`	τ_{cost} `+cat:np -cat:np`	τ_{high} `-cat:np`
$\tau_{painting}$ `+cat:np`	τ_{costs} `-cat:np -cat:np`	$\tau_{a\ lot}$ `+cat:np`
		$\tau_{is\ expensive}$ `-cat:np`

Intuitively, we would expect polarity filtering to detect the following polarities for the various possible lexical combinations, which is the same as in our running example, except with "is expensive" being taken into account:

Expected polarities

`0 cat:np`	the cost of the painting/picture is high
`+2 cat:np`	(*) the cost of the painting/picture a lot
`0 cat:np`	the painting/picture costs a lot
`-2 cat:np`	(*) the painting/picture costs is high
`0 cat:np`	the painting/picture is expensive

[10] Incidentally, this is how the semantics of SemFrag entries are encoded.

However, if we fed such a lexical selection to a single-key or an n-key polarity automaton, it would not give the right results. The first problem is that it would treat "expensive" as two separate lexical entries (one for each literal). As a result, it would find the lexical combinations in the table below, with the wrong polarities:

MISCOUNTED POLARITIES

0 cat:np	the cost of the painting/picture is high
+2 cat:np	(*) the cost of the painting/picture a lot
0 cat:np	(*) the cost of the painting/picture is expensive
-2 cat:np	(*) the painting/picture costs is high
0 cat:np	the painting/picture costs a lot
-2 cat:np	(*) the painting/picture costs is expensive
-1 cat:np	(*) the painting/picture is expensive is high
+1 cat:np	(*) the painting/picture is expensive a lot
-1 cat:np	(*) the painting/picture is expensive is expensive

Let's begin by making it clear what is and is not wrong about this miscount. First, $\tau_{is\ expensive}$ appears twice in the combination "the painting/picture is expensive is expensive", whereas we only need one. But this is also a non-issue because we could always treat the lexical combinations as sets, ignoring any duplicates. Second, incorrect paths like "the cost of the painting/picture is expensive" are being needlessly visited because their polarity is 0. But these are also a relative non-issue because the surface realiser can always detect non-sensical combinations for itself. It is always acceptable (if somewhat unfortunate) for us to underfilter. What is not acceptable, on the other hand, is to *over*filter. For example, the combination "the painting/picture is expensive" is filtered out (this appears in the table as "the painting/picture is expensive is expensive") because its polarity was incorrectly found to be -1.

To prevent this sort of overfiltering from happening (and the underfiltering while we're at it), we need to prevent the polarity filter from double-counting a lexical item. To do so, we need a mechanism for determining if we have already seen the lexical item (at a given state) or not. If we have already seen the lexical item, we do not include it a second time nor do we count its polarity again. This requires an extension to our polarity automata, which consists of adding an extra element E to the definition of the automaton state. As we shall see below, this extra element will be used to determine if we have already seen a certain semantic literal (and consequently infer which lexical items we are allowed to explore). We focus here on the n-key automata.

Definition 6 (multi-literal n-key polarity automaton)**.** An multi-literal n-key polarity automaton is an acyclic *non-deterministic* finite state automaton (the non-determinism is limited to allowing the empty transition). It is a tuple $\langle \Sigma, S, s_0, \delta, F \rangle$, where

- $\Sigma, s_0, \delta,$ and F all have the expected meaning (see Definition 4, Page 99)
- S is the set of states. Each state is of the form $\langle l, c, s, E \rangle$, with $l \in \mathbb{N}$, $c \in \mathbb{Z}$ and E is a set of natural numbers ($E \in \{e | e \subseteq \mathbb{N}\}$). The intended interpretation is that l is the index of some literal in the input semantics, c is the charge of that state, s is a state counter referring to a state in the

an $(n-1)$-key automaton and that E is the set of "extra" literal indices that we have already seen.

The main modification to automaton construction happens in the multi-literal seed automaton, where we must introduce a mechanism for ignoring already-seen literals and their lexical entries.

1. Σ is the set of lexically selected items.
2. S is built in the following way
   a) $\{s_0\}$ is an element of S (see below)
   b) If a state $\langle l, 0, 0, E\rangle$ is an element of S and if $l < |L|$, then $\langle l+1, 0, 0, E'\rangle$ is also in S, with
   $$E' = \begin{cases} E \setminus \{l\} & l \in E \\ E & \text{otherwise} \end{cases}$$
3. s_0 is the tuple $\langle 0, 0, 0, \emptyset\rangle$
4. δ is defined for all $l < |L|$ and $t \in \Sigma$ as follows
   a) Given $l \in \mathbb{N}$, a literal index and E a set of literal indices,
   b) Let X be the set of literal indices x such that $x > l$ and the semantics of t includes the x-th literal of L. These are the "extra" literals.
   c) As before, let
   $$E' = \begin{cases} E \setminus \{l\} & l \in E \\ E & \text{otherwise} \end{cases}$$
   d) $\delta(\langle l, 0, 0, E\rangle, t) = \langle l+1, 0, 0, E' \cup X\rangle$
5. F is the singleton set $\{\langle |L|, 0, 0, \emptyset\rangle\}$.

Figure 5.6 shows what the seed automaton would look like in the running example. Notice that a state is introduced to keep track of the semantics *l2:high(h)*, which $\tau_{is\ expensive}$ has in addition to its first literal *l1:cost(c,p,h)*.[11]

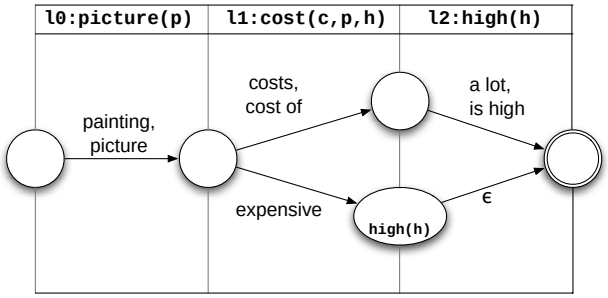


Figure 5.6: A multi-literal seed automaton

The inductive step of building a multi-literal $(n+1)$-key polarity automaton from an n-key one is a straightforward adaptation of the inductive step for

[11] It is worth bearing in mind that these literals are just sorted in some arbitrary order.

regular n-key automata (Section 5.2.3). All the work of dealing with the "seen" literals has already been done by the seed automaton, so the only thing we have to do with them is to copy them from one automaton to the next. Otherwise, there are no differences. Given a multi-literal n-key polarity automaton, and a polarity key $a':k'$, we can build an n'-key automaton $\langle \Sigma, S', s'_0, \delta', F' \rangle$ with $n' = n + 1$.

First, we define a helper function, $next : S' \times \Sigma \to S'$, as follows: $next(\langle l, c, s, E \rangle, t) = \langle l + 1, c_2, s_2, E_2 \rangle$, where $c_2 = c + pol(t, a' : k')$ and $s_2 = \langle l + 1, x, y, E_2 \rangle = \delta(s, t)$. And now the inductive step:

6. Σ is the same as before.
7. S' is built in the following way
   a) $\{s'_0\}$ is an element of S (see below)
   b) If a state $s' = \langle l, c', s, E \rangle$ is in S', and if $l < |L|$: for every t such that $\delta(s, t)$ is defined, $next(s', t)$ is in S'.
8. s'_0 is defined as follows:
   a) Let $\langle 0, z, x, E \rangle$ be the state s_0 from the previous automaton.
   b) Let z' be 0, unless $a':k'$ is required to appear in the root node (e.g., $a':k'$ is `cat:s`), in which case z' is -1.
   c) s'_0 is $\langle 0, z', s_0, E \rangle$
9. δ' is defined for all $l < |L|$. Given a state $s' = \langle l, c', s, E \rangle$, for every t such that $\delta(s, t)$ is defined, $\delta'(s', t) = next(s', t)$.
10. F' is the set of states such that for every $f = \langle l, 0, x, E \rangle \in F$ (i.e. every final state in the previous automaton), $\langle l, 0, f, E \rangle$ is in F'.

Figure 5.7 shows the multi-literal 1-key polarity automaton that would be build for the key `cat:np`.

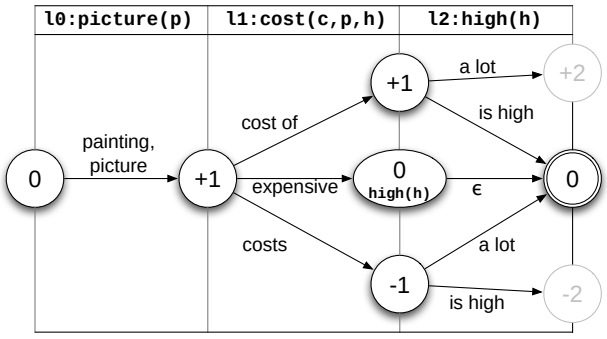


Figure 5.7: A multi-literal 1-key automaton

An inadvisable simplification

One final remark about multi-literal semantics. The mechanism for tracking the "extra" semantic literals appears to be much too complicated. Instead of all this adding and removing of literals to states, why not just make a direct transition *across* the span of multiple literals? For example, in Figure 5.7, would it not work just as well for "expensive" to transition directly from the +1 state to the final 0 state and bypass the empty transition? Here, yes but in general, *No.* The problem is that we must visit all semantic literals and do so in a fixed order; however, the semantics of multi-literal lexical items will necessarily be adjacent to each other in the order we sort them. Furthermore, no amount of sorting can make them adjacent in the general case because we encounter situations where making one set of literals contiguous comes at the price of breaking up another group. Borrowing an example from [Shemtov, 1996], the idea of "quickly moved into" can also be expressed as "rushed into" or "quickly entered". Given the ordering below, if we build a direct transition for $\tau_{entered}$ over *l0:move(m,x), l2:into(x,y)*, we would do so by erroneously skipping over the literal *ll:quick(x)*.[12]

lexical item	**semantics**		
"moved"	*l0:move(m,x)*		
"rushed"	*l0:move(m,x)*	*l1:quick(m)*	
"entered"	*l0:move(m,x)*		*l2:into(x,y)*

Multi-literal n-key polarity automata in GenI

To give an idea how these automata are used in practice, we have included in Figure 5.8 an actual run of GenI, building up a multi-literal 3-key automaton for the semantics *l1:jean(j), l2:aimer(a), l2:agent(a, j), l2:patient(a, f), l2:le(f), l3:femme(f), l4:partir(p),l4:agent(p, f).* Using the SemFRaG grammar we got 36 outputs from this, the simplest of which was "Jean aime la femme qui part" (John loves the woman who leaves).

5.2.5 Null semantic lexical items

Just as we need to account for lexical items whose semantics has more than one literal, we also need to consider the possibility that a lexical item may have an empty semantics. Lexical items with a null semantics typically correspond to function words: complementisers ("John likes **to** read"), subcategorised prepositions ("Mary accuses John **of** cheating"). Such items will be completely ignored by the polarity automata, because the algorithm for building them only triggers the insertion of items when encountering a semantic literal to which they are associated. In theory, null semantic items are a problem not just for polarity filtering but surface realisation proper. The surface realisation algorithm uses the semantics to control how often a lexical item is used (i.e. once), but since these items do not have a semantics, they could cause non-termination of the realiser. Null semantic items are an open invitation to infinite recursion. To

[12] In this example, we could still work around the semantics by sorting the semantics so that *l0:move(m)* is in the centre. But this does not generalise well. Consider the semantics *foo,bar,baz,quux* and a set of lexical items with the semantics *foo, bar*; *foo, baz* and *foo, quux*. We cannot sort ourselves out of that one.

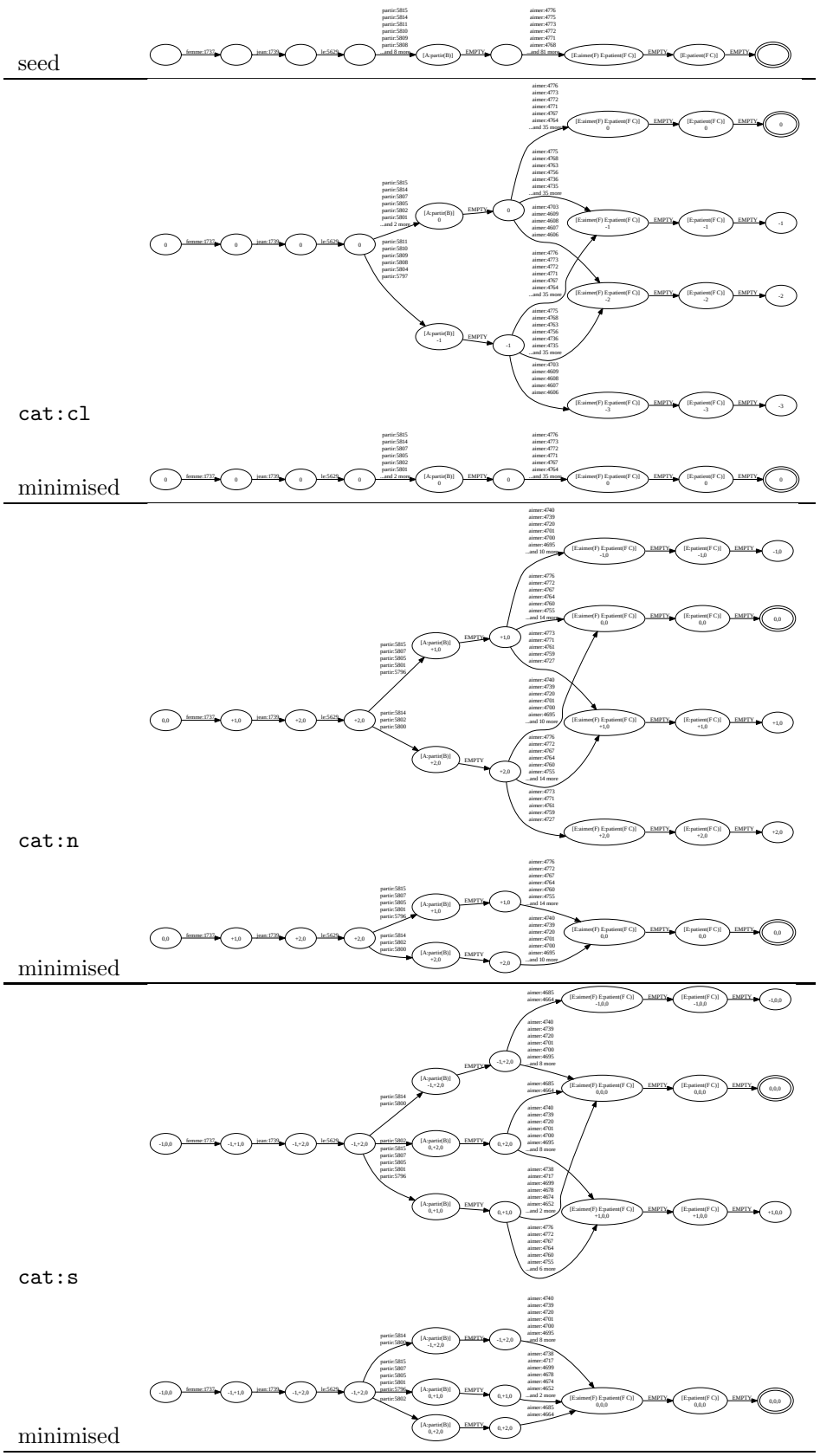


Figure 5.8: Building a multi-literal 3-key polarity automaton in GENI

deal with such lexical items, we could either take precautions to prevent any mishaps (for example, by restricting their use or their syntactic properties) or outright forbid them in the lexicon.

We chose to forgo the use of null semantic items. The reason we can afford to do this is that TAG's extended domain of locality makes it practical to treat these items as part of the elementary tree, that is, as co-anchors to some primary lexical item. The English infinitival "to", for example, can appear in the tree $\tau_{to\ take}$ as `S(Comp(to),V(take),NP↓)`. For the moment, there appear to be no plausible lexical items aside from such co-anchors that would have a genuinely null semantics.

On the other hand, it is conceivable that certain lexical items, namely pronouns, have a hidden semantics. For example, it may be reasonable to expect a grammar to associate the semantics *l1:bill(b), l2:kick(b,b)* with "Bill kicks himself". In this example the lexical item for "himself" is not ostensibly associated with any literal. That looks deceptively like a null semantics, but we would argue that "himself" does indeed have a semantics, namely a variable X that can be instantiated to an index like *b*. Using such lexical items requires that we discover the semantics hidden behind such inputs, something which we will attempt in Section 5.4.1. In the meantime, we can work around the problem by eschewing the use of hidden semantic items. We will handle pronouns, for example, by using an explicit literal so that in order to realise "Bill kicks himself", we might input something like *l1:bill(b), l2:kick(b,b), l3:himself(b).*

5.3 Chart generation with polarity automata

Polarity filtering sits between lexical selection and surface realisation proper. It gives us a compact representation of those combinations of lexical items that potentially lead to a successful realisation. The question that remains is how exactly we can go about using this information.

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. This at least prevents us from trying to combine lexical items that do not belong together. However, this simple approach is inefficient because it does not account for the fact that different paths may often have lexical items in common.

The second, more sophisticated approach does not perform a separate chart generation step for each path. Instead we generate using all the paths at once, but 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. Notice that this approach can be thought of as an extension to whatever chart indexing scheme (Section 1.3.4) that is in use.

5.4 Extensions

Adapting the polarity filtering technique to surface realisation has required that we (i) build polarity automata on the basis of literals instead of lemmas

HIDDEN SEMANTICS	
extra columns in automata	yes
virtual literals	yes
auto-detection of semantic "polarities"	no
INTERVALS	
atomic disjunction	yes
variables	no
ADJUNCTION	
adjunction polarities	no
with intervals	no
SMALLER AUTOMATA	
sorting the keys	no
sorting the literals	yes
bundling transitions	yes

Table 5.2: Status of extensions in GenI

(ii) extend the algorithm for multi-literal semantics and (iii) account for items with a null semantics. That was just the bare minimum for getting polarity filtering to work in generation. There isn't any reason to stop there. In this section, we present a grab bag of techniques for making better use of polarity automata, or more specifically

1. using lexical items that have a hidden semantics,
2. dealing with variables and atomic disjunction whilst computing the polarity of lexical items [Bonfante *et al.*, 2004],
3. filtering for TAG adjunction,
4. building smaller automata

Not all of these extensions have been implemented in GenI yet; see Table 5.2 for details.[13]

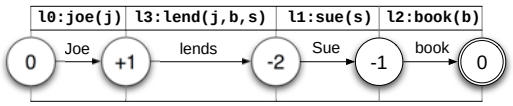
5.4.1 Hidden semantics revisited

We have so far avoided the issue of hidden semantics by simply legislating them out of existence. What if we did actually want to use such items? The use we have in mind is for pronouns, whose semantics may arguably be a variable index.[14] For example, the pronoun "her" in (20b) has semantics *L:X*, instantiated to *l1:s*, and in (21), "he" has the semantics *L:Y*, instantiated to *l4:j*.

[13] Variables should be fairly easy to add in, but one will need to test it and determine if it is better if in practice it would be more useful to accept variable values, or to limit ourselves to constants and warn the user when there are any variables.

[14] We retain the practice of associating each literal with a label so that we can distinguish between multiple uses of the same index.

(20a)



(20b)

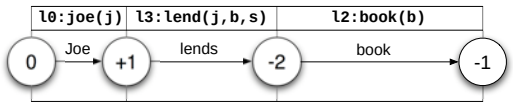


Figure 5.9: Difficulty with hidden semantics.

(20) a. *l0:joe(j), l1:sue(s), l2:book(b), l3:lend(j,b,s), l4:boring(b)*
"Joe lends Sue a boring book."

b. *l0:joe(j), l2:book(b), l3:lend(j,b,s), l4:boring(b)*
"Joe lends her a boring book."

(21) *l0:joe(j), l1:sue(s), l2:leave(j), l3:promise(p,j,s,l2)*
"Joe promises Sue to leave."
or "Joe promises Sue that he would leave."

In Figure 5.9, we can see how polarity automata for (20a) and (20b) differ. Building an automaton for (20b) fails because τ_{sue} is not available to cancel the negative polarities for τ_{lends}; instead, a pronoun must be used to take its place. The problem is that the selection of a lexical item is only triggered when the construction algorithm visits one of its semantic literals. Since pronoun semantics have zero literals, they are *never* selected. Making pronouns visible to the construction algorithm would require us to count the indices from the input semantics. Each index refers to an entity. This entity must be "consumed" by a syntactic functor (e.g. a verb) and "provided" by a syntactic argument (e.g. a noun).

We make this explicit by annotating the semantics of the lexical input (that is, the set of lexical items selected on the basis of the input semantics) with a form of polarities. Roughly, nouns provide indices[15] (+), modifiers leave them unaffected, and verbs consume them (−). Predicting pronouns is then a matter of counting the indices. If the positive and negative indices cancel each other out, no pronouns are required. If there are more negative indices than positive ones, then as many pronouns are required as there are negative excess indices. In the table below, we show how the example semantics above may be annotated and how many negative excess indices result:

semantics	b	j	s
l0:joe(+j) l1:sue(+s) l2:book(+b) l3:lend(-j,-b,-s) l4:boring(b)	*0*	*0*	*0*
l0:joe(+j) l2:book(+b) l3:lend(-j,-b,-s) l4:boring(b)	*0*	*0*	*1*
l0:joe(+j) l1:sue(+s) l2:leave(-j,-s) l3:promise(p, j,-s,l2)	*0*	*0*	*0*
l0:joe(+j) l1:sue(+s) l2:leave(-j,-s) l3:promise(p,-j,-s,l2)	*0*	*1*	*0*

[15] except for predicative nouns, which like verbs, are semantic functors

Counting surplus indices allows us to establish the number of pronouns used and thus gives us the information needed to build polarity automata. We implement this by introducing a virtual literal for negative excess index, and having that literal be realised by pronouns. Building the polarity automaton as normal yields lexical combinations with the required number of pronouns, as in Figure 5.10.

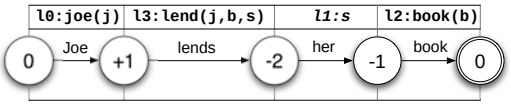


Figure 5.10: Constructing a polarity automaton with hidden semantics.

This becomes more complicated when the lexical input contains lexical items with different annotations for the same semantics. For instance, the control verb "promise" has two forms: one which solicits an infinitive as in "promise to leave", and one which solicits a declarative clause as in "promise that he would leave". This means two different counts of subject index *l4:j* in (21): zero for the form that subcategorises for the infinitive, or one for the declarative. But to build a single automaton, these counts must be reconciled, i.e., how many virtual literals do we introduce for *l4:j*, zero or one?

The answer is to introduce enough virtual literals to support the largest count (in this case one), and to balance them by adding the virtual literals to the lexical semantics of the smaller counts. To handle example (21), we introduce one virtual literal for *l4:j* in order to select the pronoun "he" in "promise that he would leave". This extra pronoun is not selected for the infinitive form "promises to leave", because it is accounted for in the semantics of lexical item $\tau_{promise_to}$, which now consists of *l3:promise(p,j,s,l2)* and the virtual literal *l4:j*.

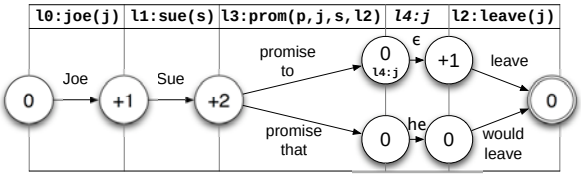


Figure 5.11: Hidden and virtual semantics

Note that this technique will still need to be further explored before it can be put to practical use. An open question that remains is how we would go about annotating each and every lexical item's semantics with the requisite polarities. Our prototypes have manual annotations, but this is clearly not a scalable solution. In the meantime, we have simply resigned ourselves to semantics by decree, using explicit literals like *l0:himself(x)* for pronouns instead of the hidden semantic mechanism.

5.4.2 Variables and atomic disjunction

To simplify the presentation of polarity automata, we have so far required that any attribute-value pair used to compute the charge of a tree may only contain

a constant value. In fact, this restriction is not necessary, because the polarity filtering mechanism proposed by [Perrier, 2003] already includes a means of dealing with cases where "unification does not reduce to identification." Before delving into those details, there are two reasons why it would be useful to handle this more general case.

The first is that we would like to support the use of atomic disjunction in the values. For example, we might imagine that a tree has a substitution node with the value `cat:{np,cl}`; it accepts a node with either an NP or a CL category. Conceptually, this is no different than using a variable value. If in a given feature structure, we define a pair `cat:X`, we are declaring that the value of the `cat` attribute is (i) associated with variable `X` and (ii) an element of T_{cat}, the set of possible attributes that may be associated with `cat`. A variable is a placeholder for a set of values; a constant is an element in that set; an atomic disjunction is halfway between the two, i.e. a subset of values.

The second reason is that it would allow for the polarity filtering mechanism to be more robust. Otherwise, we are forced to detect if any attributes used for polarity filtering are not associated with constants, emit an error message and refuse to do filtering altogether. Supporting the general case would allow us to do filtering even if not all the appropriate constraints are available in the grammar. Furthermore, as we will see in Section 5.4.3, we wish to extend the filtering mechanism to account for adjunction. One implication is that we would need to inspect most of the nodes in a tree, not just the occasional root, foot or substitution node, but all the possible adjunction sites as well. Since there are many more of these nodes than the root/foot/substitution ones, the chances are higher that one of these nodes is underconstrained, by either bug or design.

The generalised mechanism for dealing with these cases comes directly from the electrostatic tagging literature [Bonfante *et al.*, 2004]. The rough idea is to treat polarity charges not as single values like -2, but as *intervals*, for example $[\![-2, 1]\!]$. Each variable or atomic disjunction is an ambiguity — either the attribute value pair does not match the key and gets the charge 0, or it does match it and gets the charge 1 (or -1 for substitution nodes). This range of possibilities is encoded as the interval $[\![0, 1]\!]$ (or $[\![-1, 0]\!]$ for substitution nodes). Here we can introduce a addition over intervals:

$$[\![x_1, y_1]\!] + [\![x_2, y_2]\!] = [\![x_1 + x_2, y_1 + y_2]\!]$$

With this in mind, we can calculate polarities in the same way, except doing addition over intervals instead of integers. Given a tree N and a polarity key $a:k$, the charge $pol(N, a:k)$ of that tree is the sum of charges for each of its nodes:

$$pol(N, a{:}k) = \sum_{n \in N} pol'(n, a{:}k)$$

What is different here is how we assign the polarity of a node. First of all, we say that n matches $a:k$ if there is some $(a_n, k_n) \in n.bot$ such that $a_n = a$. Below, in the cases where we refer to k_n we are assuming that n matches the key. Here also, we assume that values are implemented as sets and that variable values are just the set of all possible values. Note here that k is assumed to be a singleton set.

$$pol'(n, a{:}k) = \begin{cases} [\![1,1]\!] & \text{if } n \text{ is the root node of an initial tree and } k = k_n \\ [\![0,1]\!] & \text{if } n \text{ is the root node of an initial tree and } k \subset k_n \\ [\![-1,-1]\!] & \text{if } n \text{ is a substitution node and } k = k_n \\ [\![-1,0]\!] & \text{if } n \text{ is a substitution node and } k \subset k_n \\ [\![0,0]\!] & \text{if } k \cap k_n = \emptyset \text{ or } n \text{ does not match } a{:}k \end{cases}$$

Notice that the first, third and fifth case are the same as in the original definition, but expressed as trivial interval. These account for constants and non-matches, which we already knew how to deal with. The second and fourth cases account for variables and atomic disjunction. Making this work for polarity automata requires only a modification to the notion of charge (intervals) and consequently to the initial and final state. We show here how to redefine single-key polarity automata (Definition 4, Page 99); extending this to multi-literal n-key automaton is a straightforward affair.

Definition 7 (Single-key interval polarity automaton)**.** A single-key interval polarity automaton is an acyclic, deterministic finite state automaton. It is a tuple $\langle \Sigma, S, s_0, \delta, F \rangle$ where

- $\Sigma, s_0, \delta,$ and F all have the expected meaning.
- S is the set of states, where in each state $\langle l, c \rangle$, with $l \in \mathbb{N}$ and $c \in [\![\mathbb{Z}, \mathbb{Z}]\!]$[16]. The intended interpretation is that l is the index of some literal in the input semantics and c is the charge of that state.

Given an input semantics L and a polarity key $a{:}k$, a single-key interval polarity automaton can be built in the same way as single-key polarity automata, with the following modifications to the start and final states (a state is final if its charge *includes* 0):

1. s_0 is the tuple $\langle 0, z \rangle$, where z is $[\![0,0]\!]$, unless $a{:}k$ is required to appear in the root node (for example, $a{:}k$ is `cat:s`), in which case z is $[\![-1,-1]\!]$.
2. F is the set of edges $\{\langle |L|, [\![x,y]\!] \rangle$ where $x \leq 0$ and $y \geq 0\}$.

Note that allowing for such ambiguities may somewhat degrade filtering. For example, if we have an automaton whose only final state has the interval $[\![-1,3]\!]$ we are allowing for lexical combinations whose net polarities may range from -1 to 3. This is less alarming than it may sound, as intervals overlap and still be considered distinct, so although $[\![1,2]\!]$ is a sub-interval of $[\![-1,3]\!]$ it is treated as a distinct state, and all paths which lead to it are pruned away. Nevertheless, it means that a lexical item with a non-constant value will allow for any combination that uses it and which leads to an unbalanced value. If there are lexical items with non-constant values for more than one literal, the intervals may get larger and allow more combinations to pass through the filter. Polarity filtering thus becomes more permissive about what it accepts, which is unfortunate, but outweighed by the benefit of extra robustness.

[16] That is, the set of intervals over integers

5.4.3 Adjunction

Polarity filtering has so far been focused on matching substitution sites (requirements) to root nodes (resources). If an auxiliary tree does not have any substitution nodes, it becomes entirely invisible to the polarity filter, as it neither requires nor provides resources for substitution.

Adjunction intuitions

To make more use of the auxiliary trees during filtering, we need some means of accounting for adjunction. The essential requirement is that any auxiliary tree in the lexical selection must be adjoined somewhere. In other words, if there is an auxiliary tree, there must also be an adjunction site to go with it. This sounds like it might fit into our basic framework if we treated adjunction sites as "positive" and auxiliary trees as "negative". Alas, adjunction is not a simple matter of balancing resources with their requirements, as substitution is. The subtleties are that (i) not all adjunction sites are necessarily to be adjoined to, i.e. resources need not necessarily be used (ii) we can have embedded adjunctions because auxiliary trees can have their own adjunction sites.[17] We cannot model adjunction by adding polarities.

We can do so, however, by multiplying them together. The accumulation of charges works much the same way, except that multiplication takes the place of addition, and that the initial state of the automata has the charge 1 and its final state a charge of either 0 or 1 (0 to account for cases where there is at least one adjunction site, and 1 otherwise). Adjunction polarities can be interpreted in the following way:

0 - resource If a resource (adjunction site) is available, it does not matter how many times it is used or if it used at all. Its mere availability is enough to make things balanced; likewise, multiplying anything by 0 gives us a "balanced" state 0.

1 - innocent bystander Items which are neither resources nor requirements should not come into play at all. Multiplying anything by 1 gives us the same result. If only innocent bystanders are involved, we get a "balanced" state 1.

2 - requirement It does not matter how many requirements (foot nodes) there are. Once there is at least one requirement of a given type, there must be at least one resource. Multiplying by 2 takes us past the balanced state ≤ 1, and the only way to get back is to multiply by 0. Also, treating any accumulated polarities > 2 as simply 2 can give us the same effect, with potentially smaller automata.

Adjunction automata

To simplify matters, let us ignore variables and atomic disjunction for the moment. We can now introduce a notion of adjunction charge. Given a tree N and a polarity key $a{:}k$, the adjunction charge $pol_*(N, a{:}k)$ is defined as follows

[17] Furthermore, some variants of TAG allow for multiple adjunction, which would be nice to support

- if N is an initial tree,

$$pol_*(N, a{:}k) = \begin{cases} 1 & \text{if no adjunction sites match } a{:}k \\ 0 & \text{otherwise} \end{cases}$$

- if N is an auxiliary tree,

$$pol_*(N, a{:}k) = \begin{cases} 2 & \text{if the foot node matches } a{:}k \\ 1 & \text{if neither the foot node} \\ & \text{nor any adjunction sites match } a{:}k \\ 0 & \text{otherwise} \end{cases}$$

Definition 8 (Single-key adjunction automaton)**.** A single-key adjunction automaton is an acyclic, deterministic finite state automaton. It is a tuple $\langle \Sigma, S, s_0, \delta, F \rangle$ where

- $\Sigma, s_0, \delta,$ and F all have the expected meaning.
- S is the set of states, where in each state $\langle l, c \rangle$, with $l \in \mathbb{N}$ and $c \in \mathbb{N}$. The intended interpretation is that l is the index of some literal in the input semantics and c is the adjunction charge of that state. Note that in contrast with single-key *polarity* automata, we do not need negative numbers (so natural numbers are fine).

Given an input semantics L and a polarity key $a{:}k$, a single-key adjunction automaton can be built in the same way as single-key polarity automata, with the following modifications:

1. Σ is the set of lexically selected items. For convenience, we also define the function $\sigma : \mathbb{N} \to \Sigma$, which given an index l, returns the set of lexically selected items whose semantics consists of the l-th literal in L.
2. S is built in the following way
   a) $\{s_0\}$ is an element of S (see below)
   b) If a state $\langle l, c \rangle$ is an element of S and if $l < |L|$, then for every elementary tree $t \in \sigma(l)$, the state $\langle l + 1, c' \rangle$, with $c' = c * pol(t, a{:}k)$, is also in S
3. s_0 is the tuple $\langle 0, 1 \rangle$.
4. δ is defined for all $l < |L|$ and $t \in \sigma(l)$ as: $\delta(\langle l, c \rangle, t) = \langle l + 1, c' \rangle$ with $c' = c * pol(t, a{:}k)$.
5. F is the set $\{\langle |L|, 0 \rangle, \langle |L|, 1 \rangle\}$.

Combining polarity and adjunction automata

We can combine adjunction automata with each other by using an automaton intersection algorithm. For that matter, we could even combine adjunction automata with polarity automata by the same means. We can even perform these intersections incrementally mixing adjunction automata with polarity automata. The trick would be to generalise the notion of n-key automaton so

that it covers both polarity and adjunction automata. To do this, we need to distinguish between the cases where we are using polarity keys for substitution or adjunction. We can do this by extending the polarity keys with a little marker: $a{:}k_{\downarrow}$ means the key is being used for substitution (polarity automata) and $a : k_*$ means that it is being used for adjunction (adjunction automata). To keep our notation intuitive, we should also rename the *pol* function to $pol_{\downarrow}$ so that it is clear that the former is for substitution and pol_* is for adjunction.

Definition 9 (n-key combined automaton)**.** An n-key combined automaton has exactly the same form as an n-key combined automaton, which we repeat below for convenience. It is an acyclic, deterministic finite state automaton $\langle \Sigma, S, s_0, \delta, F \rangle$, where

- $\Sigma, s_0, \delta,$ and F all have the expected meaning (see Definition 4)
- S is the set of states. Each state is of the form $\langle l, C \rangle$, with $l \in \mathbb{N}$ and C is an n-tuple of integers ($\mathbb{Z}$). The intended interpretation is that l is the index of some literal in the input semantics and C is the (n-dimensional) charge of that state.

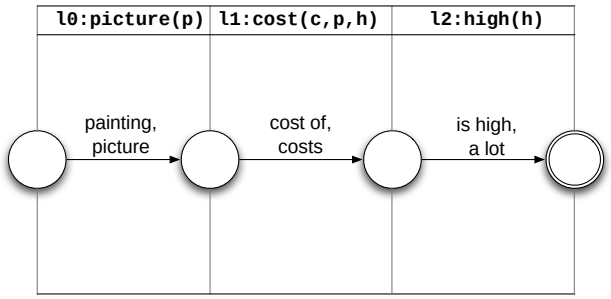


Figure 5.12: A seed automaton

n-key combined automata can be built inductively. Building the seed automaton works exactly as before (see Figure 5.12 for a quick refresher or Page 101 for the full details). As for the induction step, it is not really all that different from before. The only modification we need to make is that we must distinguish between $a : k_{\downarrow}$ and $a : k_*$. For polarity keys of the form $a : k_{\downarrow}$, we combine charges with addition; for the latter, we combine them with multiplication.

Given an n-key polarity automaton, and a polarity key $a' : k'_{op}$, we can build an n'-key automaton $\langle \Sigma, S', s'_0, \delta', F' \rangle$ with $n' = n + 1$. First, the helper function, $next : S' \times \Sigma \rightarrow S'$ is defined as $next(\langle l, c, s \rangle, t) = \langle l+1, c_2, s_2 \rangle$, where $s2 = \delta(s, t)$ and

$$c_2 = \begin{cases} c + pol_{\downarrow}(t, a'{:}k') & \text{if } op = \downarrow \\ c * pol_*(t, a'{:}k') & \text{if } op = * \end{cases}$$

And now the induction step:

1. Σ is as usual.
2. S' is built the same way as before:

a) $\{s'_0\}$ is an element of S (see below)

b) If a state $s' = \langle l, c', s\rangle$ is in S', and if $l < |L|$: for every t such that $\delta(s, t)$ is defined, $next(s', t)$ is in S'.

3. s'_0 is defined as follows:

a) Let $z' = 1$ if $op = *$; otherwise let it be 0, unless $a' : k'$ is required to appear in the root node (e.g., $a' : k'$ is `cat:s`), in which case z' is -1.

b) s'_0 is $\langle 0, z, s_0\rangle$

4. δ' is defined the same way as before, for all $l < |L|$. Given $s' = \langle l, c, s\rangle$, for every t such that $\delta(s, t)$ is defined, $\delta'(s', t) = next(s', t)$.

5. F' is the set of states such that for every $f \in F$ (i.e. every final state in the previous automaton):

a) $\langle l, 0, f\rangle$ is in F'

b) if $op = *$, $\langle l, 1, f\rangle$ is also in F'.

Adjunction and intervals

Adjunction charges can also be extended to non-constant values. If an auxiliary tree has a foot node may with an non-constant one of its possible values is a key $a' : k'$, we assign the interval $[\![1, 2]\!]$. The same reasoning can be applied to adjunction sites; trees with such sites get assigned the interval $[\![0, 1]\!]$ (or simply $[\![0, 0]\!]$ if there is a site which unambiguously matches key k). Multiplication on two intervals is straightforward $[\![x_1, y_1]\!] \times [\![x_2, y_2]\!] = [\![x_1 \times x_2, y_1 \times y_2]\!]$, and as usual, any value > 2 can also just be treated as 2.

5.4.4 Building small automata

Constructing polarity automata can introduce some overhead (see Section 5.5.1 for the theoretical cost). To reduce this overhead, we use a combination of techniques for keeping the polarity automata small.

The first technique consists of sorting the polarity keys by decreasing effectiveness, using the keys which eliminate the most lexical combinations first. Every time we construct a polarity automaton for a given key, we are exploring the space of possible lexical combinations, so it is desirable to shrink this space as early possible, allowing each subsequent automaton to be smaller. Unfortunately, as [Le Roux, 2007a] shows, the general problem of ordering polarity keys to get the smallest possible polarity automata is NP-complete. Discovering a good ordering for polarity keys in practice can be done experimentally, likely with a different order for every grammar.

The second technique is to sort the literals of the input semantics in a similar strategic manner. Since we can impose any arbitrary ordering on the literals, we might as well choose one which produces the smallest automata. The sorting scheme we use is sort the literals in order of increasing ambiguity. The idea is to delay branching of the automata as much as possible to avoid the proliferation of states. To see why we believe this works, consider the two automata in the Figure 5.13. We compare a hypothetical "worst case" automaton with three

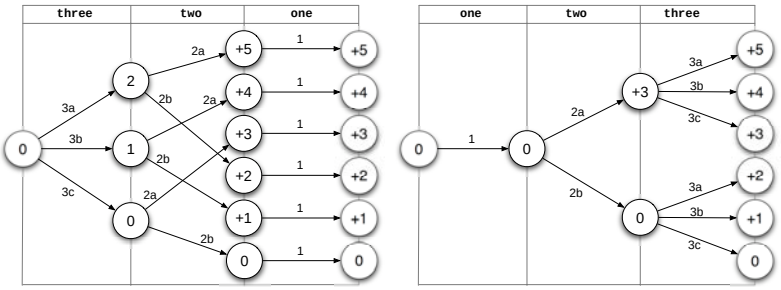


Figure 5.13: Sorting literals by their ambiguity

literals, with an ambiguity of 1, 2 and 3, respectively. Both automata end up with 6 final states, but what happens in the middle differs. The automaton on the left corresponds to an ordering of 3, 2, 1; we branch off 3 states, then 2 states each, then 1 state each (total: $1 + 3 + 4 + 6 = 16$). The automaton on the right corresponds to an ordering of 1, 2, 3; we branch off 1 state, then 2 states each, then three states each (total: $1 + 1 + 2 + 6 = 10$) yielding a somewhat smaller automaton. This may not necessarily be the best way to sort the literals, but it seems fairly reasonable as far as uneducated guesses go.[18] A more linguistically motivated approach would be worth looking into, on the other hand.

The third technique is the easiest to justify. In large grammars, we often have many lexical items for a given literal that have the same charge; this leads to there sometimes being many transitions between the same two states. We can with little effort hasten the construction and intersection of such automata by "bundling" together transitions between the same states. Any two states may only have one transition between them, but that transition stands for a set of lexical items. At the very end, we shall have to "unbundle" these transitions by trying every combination of lexical items along the same automaton path; we will still have the same number of combinations to contend with, but we will just be working with smaller automata in the process.

5.5 Evaluation

5.5.1 Theoretical cost of filtering

Construction It is possible to demonstrate that the construction of an individual polarity automaton is quadratic with respect to the number of literals in the target semantics. [Tapanainen, 1997] shows a more general result which is applicable here: if a finite-state automaton is acyclic, deterministic and has a column-based structure (i.e. as polarity automata do), they can be shown to have a diamond-like shape, with the size of the diamond being constrained. We

[18] Le Roux (personal communication) points out that this technique can be justified by the reasoning in [Tapanainen, 1997]. We have not yet verified this.

show here the rough intuitions for how this would apply to polarity automata, using a more pessimistic, but simpler triangle shape.

Consider any state in the automaton. If we assume that all trees can affect polarity by at most some constant c (around 2 or 3 for realistic grammars) then any state can transition to at most $2c$ possible states each representing a distinct change of polarity ranging from $-c$ to $+c$. An important observation now is that neighbouring polarities will converge on the same states. This can be illustrated with an example:

- if at a given literal, a state has a charge +5, a +2 transition from that state would lead to a +7 charge;
- if at the same literal another state has a +6 charge, a +1 transition from that state would lead converge to the same +7 charge.

Given x states $(s_1..s_x)$ at literal i, the number of states for literal $i+1$ will thus be $x+2c$, that is x because of convergence between nearby polarities, and c at both extremities. If we unfold this induction, we find that the number of states that represent literal i during the construction process is equal to $2ci$. Summing these up leads to a total for the number of states in the worst case polarity automaton:

$$\sum_{1 \leq i \leq n} 2ci = 2c \sum_{1 \leq i \leq n} i = c(n^2 + n)$$

Finally we note that the number of transitions and possible automata are also limited. The number of transitions out of each state is equal to the lexical ambiguity a, so there are $a2c(n^2 + n)$ transitions in the entire automaton. Likewise, if we make the real-world assumption that number of polarity keys is bounded by some p (usually also 2 or 3), we can conclude that the number of transitions that get considered for all the automata is $pa2c(n^2 + n)$. Once again, p, c and a are constants, so the construction of polarity automata can be done in $O(n^2)$ space. Polarity automata are upper-bounded to polynomial size (though they encode an exponential number of paths). We can reason that the time to construct these automata has the same upper bound. The idea is that each transition of the automaton corresponds to some constant number of operations, and so building the automata does requires no more time than space.

Minimisation We can minimise an automaton in $O(q)$ time (with respect to q the number of states in the automaton) [Revuz, 1992]. As long as we use only one polarity key for filtering, a polarity automaton only induces polynomial overhead. On the other hand, working with multiple polarity automata makes things trickier. Constructing and minimising multiple polarity automata is just a matter doing the computations k times, with k the number of polarity keys.

Intersection The main concern in the complexity of polarity filtering is that of automaton intersection. The problem is that the *intersection* of polarity automata balloons into an $O(q^k)$ affair, again q being the number of states in a given automaton and k the number of polarity keys. When computing the

intersection of two automaton, we essentially consider the Cartesian product of their states, giving us $q \times q \cdots \times q$ states to deal with.[19]

This presents an interesting trade-off. On the one hand, intuition tells us that the more polarity keys we use, the better our filtering is. On the other hand, theory tells us that every polarity key we add increases the complexity of filtering by another $\times q$. So what happens in practice? Do the automata get so small and simple as a result of filtering that in practice calculating their intersection becomes very cheap? For that matter, is the overhead of building these automata, minimising them and even taking their intersection less than the equivalent cost of realisation on the unfiltered lexical combinations? These questions depend on the actual grammars being used, and the semantic inputs. We turn therefore to an empirical evaluation.

5.5.2 Practical benefits of filtering

To begin with, it is clear that polarity filtering reduces the effect of lexical ambiguity. There are two ways to measure this, first by seeing the direct effect on the number of lexical combinations and second by measuring indirect artefacts like the number of derived trees built by the surface realiser. Below are the numbers for our worst (largest) test case (with 8 literals excluding thematic roles, and 231 paraphrases):

	no filtering	**filtering**
lexical combinations	2 436 672	4 136
substitutions	26 149	3 284
adjunctions	5 014	630

To obtain the number of lexical combinations, we counted the paths of our polarity automata. Each path corresponds to one combination of lexical items. To get the number of combinations without filtering, we counted specifically the paths on the seed automaton. As for the number of combinations with filtering, we counted the paths on the final automaton, i.e. with all polarity keys taken into account. (Note that the formula we saw in Section 5.1.1 would not do for counting the ambiguity, if multi-literal semantics comes into play; using the seed automata is a more robust way to count.) Comparing these two numbers provides an estimate on the usefulness of the polarity filter. In our suite, the initial automata for each case have 1 to 2 436 672 paths. The final automata have 1 to 4136 paths. This can represent quite a large reduction in search space (we have eliminated 99.9% of the combinations).

Then again, it is not clear what the concrete effect of eliminating these paths is. For example, maybe in practice, a naive surface realiser would quickly "prunes" away parts of the search space that involve lexical combinations that polarity filtering would have thrown out, so perhaps this huge reduction in the lexical combinations does not actually mean very much. A more convincing presentation of the benefits of polarity filtering would measure not just how

[19] The approach of building n-key polarity automaton slightly improves on this by exploiting the literal-by-literal nature of polarity automata. This means that we do not contend with the Cartesian product of *every* state in the automata, only the states that correspond to the same literals. Since we know that the number of states that correspond to a given literal is $2ci$, the complexity of intersecting polarity automata is $O(n(2ci)^k)$, still exponential, but with a smaller base. See [Le Roux, 2007b] for details.

many lexical combinations we throw out, but what their actual consequences are for surface realisation. This is where the substitution and adjunction counts come in. Each of these serves as an estimate of how much work the realiser actually does. Without filtering, the surface realiser performed 26 149 substitutions and 5014 adjunctions without polarity filtering. With filtering, it only performed 3284 substitutions and 630 adjunctions. In short, polarity filtering has a measurable impact both on the size of the search space and on what the surface realiser actually explores.

5.5.3 Cost versus benefit: is it worth the overhead?

We know that polarity filtering can provide a good reduction in surface realisation search space. A somewhat more complicated question to answer is if the overhead of building, minimising and intersecting these automata offsets the gains we make. Indeed, realisation times with and without filtering are comparable for most of the test suite, but for the most complicated sentence in the core suite, polarity filtering makes surface realisation 98.4% faster, producing a result in 25.21 CPU seconds instead of 1615.7.

It would be useful to also look into how the polarity automata grow as we add polarity keys (because of exponential cost of automaton intersection, Section 5.5.1). We have not yet done this, partly for lack of time and partly because the number of polarity keys we use is rather small (4 maximum). To get an idea of how this behaves in practice, here is the growth of our worst-case automaton in number of transitions.

Number of transitions

	unpruned	**pruned**
0-key	155	155
1-key	637	85
2-key	412	135
3-key	321	121
4-key	132	30

As we can see, the number of transitions never exceeds that of the first polarity key (before we have done any automaton intersection). So far this gives the impression that in practice, polarity filtering simplifies the automata enough for intersection not to be a problem. But it will be worth looking into deeper into this as our input lengths grow and as we increase the number of polarity keys at our disposal.

5.6 Related work in lexical disambiguation

5.6.1 LEOPAR: Electrostatic tagging

Polarity filtering is an application of electrostatic tagging to surface realisation [Perrier, 2003].[20] Normally, in supertagging, one thinks of returning one elementary tree per word (or in surface realisation, literal). The original literature on supertagging [Joshi and Srinivas, 1994], or more recently, [Bangalore and Joshi, 1999], uses a trigram-based approach to choose the most appropriate

[20] See also [Bonfante *et al.*, 2004], which is in English

supertag for a word. [Perrier, 2003] prefers to avoid the risk of supertagging errors, which would in turn lead to unrecoverable parsing errors. Instead, he prefers to use an "exact method" which returns all of the valid combinations of supertags instead of just the most probable ones. This is ideal for applications that require parsers to return all parses for a string. Likewise, we want to return all paraphrases for a given input semantics and so an exact method is equally appealing for realisation.

Note that much work has been done in electrostatic tagging since the original proposal by [Perrier, 2003] *et al.* This is summed up in [Le Roux, 2007b], which proposes an Interaction Grammar account of coordination. Allowing for coordination means allowing for very long sentences, so Le Roux proposes some additional techniques for reducing the overhead in building polarity automata, particularly in taming the exponential cost of automaton intersection:

1. Polarity keys are chosen carefully to reduce the number of intersections needed whilst keeping the filter effective.

2. In a polarity key $a{:}k$, the set k is allowed to have more than one element. Such polarity keys reduce the effectiveness of the filter, but also the overhead in building automata. Le Roux uses them to minimise the presence intervals in the polarity automata, which greatly reduces their size.

3. Whenever coordination comes into play, the automata are "chunked". An additional check is inserted to verify that the chunks on both sides the conjunction have the same charge. For example, in a sentence like "John loves Mary and Pierre's wife", the portions of the automata for "Mary" and for "Pierre's wife" must both have a `+cat:np` charge.

It is worth noting that the Interaction Grammar used by Le Roux makes much heavier use of polarities than SemFRaG does. Atomic disjunction comes into play more frequently, and many more polarity keys are used in practice. In any case, future extensions to polarity filtering should take this work into account.

5.6.2 FERGUS: supertagging from trees

Just as polarity filtering is an adaptation of electrostatic tagging to generation, statistical supertagging has its own analogue in generation. Fergus [Bangalore and Rambow, 2000a] is a surface realiser for TAG. As we had mentioned in Section 4.3.1, Fergus has two components, a "Tree Chooser" and a "Linear Precedence" module. We had briefly mentioned the Linear Precedence component and will revisit it again in the next chapter. Here, we are more interested in the "Tree Chooser" component.

The input to Fergus consists in a dependency tree with a lexeme on each node; this is essentially a TAG derivation tree (the difference being that derivation trees associate each node with both a lexeme and an elementary tree). The job of the Fergus "Tree Chooser" is to assign elementary trees to each of these lexemes. This is done using a stochastic tree model trained on the xtag derivation trees that correspond to 1 000 000 words of sentences from the Wall Street Journal.

We have the same reticence to use probabilistic methods for disambiguation as [Perrier, 2003], because we are interested in returning all paraphrases and

	parsing	generation
statistical	supertagging in a sequence [Bangalore and Joshi, 1999]	supertagging in a tree [Bangalore and Rambow, 2000a] FERGUS
symbolic	electrostatic tagging [Perrier, 2003] LEOPAR	polarity filtering [Kow, 2005] GENI

Table 5.3: Electrostatic tagging and polarity filtering

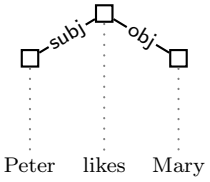
consequently need to preserve all valid sets of supertags. Nevertheless, it would be interesting to see how one might apply such an approach to GENI. A key difference is that the input to GENI ambiguously corresponds to what might be several distinct inputs in FERGUS. [Bangalore and Rambow, 2000a] give the example of `fear(Indians(many), repeat(a, of(experience(that))))` as the dependency tree that corresponds to "many Indians feared a repeat of that experience". The equivalent GENI input would correspond to a set of dependency trees including the one above and perhaps `fear(repeat(a, of(experience(that))), by(Indians(many)))`, "a repeat of that experience was feared by many Indians" and others still. Some possibilities might include unfolding the GENI flat semantic input into a set of dependency trees and perform supertagging on those, or perhaps developing a stochastic model on the kind of graph that our flat semantic inputs represent.

5.6.3 Generation as dependency parsing

[Koller and Striegnitz, 2002] also perform TAG surface realisation from a flat semantics. They observe that the formalism Topological Dependency Grammar (TDG, [Duchier and Debusmann, 2001]) was specifically developed for efficiently parsing free word order languages. As [Kay, 1996] points out, this task is remarkably similar to surface realisation. Their approach thus consists of translating TAG realisation into TDG parsing.

A TDG parser produces a dependency tree with a node for every word (and a word for every node, a one-to-one correspondence) and a label for every edge. In addition to being associated with a node, each word in the sentence is also assigned a set of lexical entries. In other words, there is lexical ambiguity. The job of the TDG parser is to pick one lexical entry for every word/node and to connect the nodes together to form a tree. Since the lexical ambiguities can multiply out, and since the nodes can be connected in any order (because of free word order), there can be theoretically very many trees. Fortunately, there is a mechanism to limit the number of trees actually produced. Each node has some number of outgoing edges and some number of incoming ones. If we connect two nodes together, the outgoing edge of one node is the incoming edge of the other. Each lexical entry can be associated with constraints on both the outgoing and incoming edges. Below is how a TDG lexicon might look:

word	allow incoming	require outgoing
likes	$\emptyset$	$\{\mathsf{subj}, \mathsf{obj}, \mathsf{adv}*\}$
Peter	$\{\mathsf{subj}, \mathsf{obj}\}$	$\emptyset$
Mary	$\{\mathsf{subj}, \mathsf{obj}\}$	$\emptyset$



Here, the hypothetical lexical item "likes" might not have restrictions on its incoming nodes, but might require precisely one outgoing subj edge and one outgoing obj edge and arbitrarily many (or no) outgoing adv edges. We can see in the margin what we would get for the sentence "Peter likes Mary".

Now the question turns to how we might transform a TAG generation problem to a TDG parsing one. The first trick is to treat each literal of the input semantics as a TDG word, along with a special *start* word. As before, each "word" is associated with a set of lexical entries, namely TAG elementary trees. The second trick is to encode TAG substitution and adjunction nodes as TDG label constraints:

- Each substitution node corresponds to a (required) outgoing subst label, and each root node of an initial tree to an (allowed) incoming subst
- Each auxiliary tree corresponds to an allowed (incoming) adj label and each node that accepts adjunction to zero-to-many adj labels

Furthermore, these constraints specify not just the TAG operation that must be on each label, but the node category and semantic index as well.[21,22] A lexicon for TDG-based TAG realisation would be as follows:

atom	allow incoming	require outgoing
start	$\emptyset$	$\{\mathsf{subst}_{S,e,1}\}$
buy	$\{\mathsf{subst}_{S,e,1}\}$	$\{\mathsf{subst}_{NP,c,1}, \mathsf{subst}_{NP,m,1}, \mathsf{adj}_{VP,e}*, \mathsf{adj}_{V,e}*\}$
mary	$\{\mathsf{subst}_{NP,m,1}, \mathsf{subst}_{NP,m,2}\}$	$\{\mathsf{adj}_{NP,1}*, \mathsf{adj}_{PN,m}*\}$
indef	$\{\mathsf{subst}_{NP,c,1}, \mathsf{subst}_{NP,c,2}\}$	$\{\mathsf{adj}_{NP,c}*, \mathsf{subst}_{N,c,1}\}$
red	$\{\mathsf{subst}_{N,c,1}\}$	$\{\mathsf{adj}_{N,c}*\}$
car	$\{\mathsf{adj}_{N,c}\}$	$\emptyset$

All that remains is to feed the input semantics to the TDG parser and retrieve the corresponding dependency tree(s). The constraints ensure that exactly once: each substitution node is filled, each root node of an initial tree is plugged in, and each auxiliary tree is adjoined to some node[23] The dependency tree corresponds exactly to a TAG derivation tree, and building the derived tree from that is trivial.

[21] Technically, we also have to associate each substitution node with a counter, in case there is more than one node with the same label/index. We believe that if TDG allowed for label/valency requirements to be expressed as a multiset instead of a set, we would not need to do this.

[22] In [Koller and Striegnitz, 2002], *indef* is missing an outgoing $\mathsf{subst}_{N,c,1}$ constraint. We believe that this was a typo.

[23] If it seems odd that this be enforced for auxiliary trees — after all the lexicon only stipulates that auxiliary trees *allow* incoming adj labels — it helps to remember that all TDG nodes have to be connected to something, and *that* is what ensures that the auxiliary trees will be used. This is similar to our chart generator's requirement that the output semantics must match the input semantics. The incoming label constraint merely ensures that they are connected by adjunction and not substitution.

Koller and Striegnitz's approach is quite similar in spirit to ours. We both address the issue of lexical ambiguity by using a set of global, mutually interlocking constraints where each substitution node must be plugged and each auxiliary tree adjoined. One interesting aspect of their approach is that they do lexical selection and construction of the derivation trees in one go. In an XTAG style grammar (like we both), getting from a derivation tree to a derived tree is a short step, at least much shorter than our use of chart generation.

Their approach does have some minor warts. The first is that they do not have a treatment of multi-literal semantics. On the other hand, they do have a workaround which consists of building the set of partitions of the semantic input, such that each lexical item is associated with one partition, essentially find all the possible ways to convert into a generation problem without multi-literal semantics. In theory, there are an exponential number of partitions, but they report good results in practice.

Like us, Koller and Striegnitz avoid the issue of null semantic items by appealing to TAG co-anchors. It might also be possible to transpose our technique for dealing with items that have a hidden semantics, since all we are doing ultimately is "discovering" their semantics. Presumably, we could have a pre-processing step of identifying the hidden semantics of any applicable lexical items and then performing TDG parsing as usual.

Another technical problem is that the approach does not support atomic disjunction in values. Then again, this should not be difficult to add. For example, atomic disjunction on a substitution node might translate to an outgoing label constraint that requires zero or one of each legal value.

The authors also note that they do not handle general feature structures, only indices and categories. But they also point out that wrong realisations, from feature mismatches undetected could always be filtered out in a post-processing step. This is really no worse than what we do in GenI, where we perform polarity filtering with only categories and indices and "post-process" the resulting lexical combinations with a chart generator.

Finally, Koller and Striegnitz report good results, converting the XTAG grammar into their TDG formalism and using it for realisation. To get an idea on the performance of their system, they revisited an example sentence from [Carroll *et al.*, 1999] ("Our manager organised an unusual additional weekly departmental conference"), generating it in 470 ms on a 700 MHz Pentium-III (Carroll *et al.*'s reported result was 4.3 seconds; but according to Koller and Striegnitz, a newer version gets this in 420 ms).

Chapter 6

Paraphrase selection

Chapter 5 was dedicated to coping with the lexical ambiguity problem. A question which arises is why lexical ambiguity even exists in the first place. After all, one might argue that lexical ambiguity reveals a fundamental design flaw, that either the grammar or the input semantic formalism are not sufficiently precise. In our the case, however, lexical ambiguity is entirely deliberate; it is far more feature than bug. It exists because the surface realiser and grammar are designed to treat syntactic paraphrases as having essentially the same semantics. For example, a French sentence "Jean aime Marie" (John loves Mary) would have the semantics *l1:jean(j) l2:aimer(e,j,m) l3:marie(m)*, and so would its many paraphrases:

(22) a. Marie est aimée par Jean

b. C'est Jean qui aime Marie

c. C'est Jean de qui est aimée Marie

d. C'est par Jean que Marie est aimée

e. C'est par Jean qu'est aimée Marie

f. C'est Jean dont est aimée Marie

g. C'est Marie que Jean aime

h. ...

On the other hand, having a long list of paraphrases for a given semantic input is not particularly useful for practical NLG applications. In this chapter, we discuss a technique for making the surface realiser output at most one sentence per input semantics. The approach we use is built around the intuition that paraphrase selection is tightly linked to lexical selection. The approach consists essentially of enriching the input with a set of linguistic properties (e.g. *l2:aimer(e,j,m)* must be realised in the active voice) that control the initial choice of lexical items, and consequently, the paraphrases produced.

6.1 Contextual appropriateness

Before exploring the selection mechanism in greater detail, it is worth scrutinising our objectives more carefully. To begin with, there are two *non*-reasons for aiming to produce a single paraphrase. The first is that it might be useful for its own sake, or because NLG applications only require one output. If that were case, we would simply generate all the outputs and choose one at random. The second non-reason is efficiency. Efficiency may be a very compelling motivator, a highly desirable bonus, but it is not the main driver behind this push for a single paraphrase.

Our main motivation can be found in this example, which comes from [Halliday, 1978] and is repeated in the KPML literature [Bateman *et al.*, 1992]:

(23) Now comes the President here. It's the window he's stepping through to wave to the crowd. On his victory his opponent congratulates him. What they are shaking now is hands. A speech is going to be made by him. "Gentleman and ladies. That you are confident in me honours me. We shall, hereby pledge I, turn this country into a place, in which what people do safely will be live, and the ones who grow up happily will be able to be their children."

The passage above may seem to be perfectly reasonable English at first glance, but when read slowly and out loud, it becomes clearer that there is something subtly off about it. Some specific problems might be that the time circumstance "now" has been thematically marked in the first sentence, where "here" would be more natural; a clefted form "It's the window he's..." is used in the second sentence, although no contrasts are being made; and in the sentence "A speech is going to be made by him" the passive voice is used, where the active voice would be better [Halliday, 1985]. None of these are, strictly speaking, grammatical errors. The passage above is full of technically correct, yet contextually inappropriate sentences, sentences which do not fit into the context, including and not limited to the surrounding discourse and user knowledge.

A surface realiser should thus provide facilities for eliminating any contextually inappropriate paraphrases from the output. Better yet, it should aim to produce the one best paraphrase for a given input. The reason that we insist on one paraphrase, and not for example, the best four or five, is an assumption or perhaps a principle that "all linguistic choices are meaningful" as is claimed in the field of Systemic Functional Linguistics [Winograd, 1983]. Every syntactic alternative, for example, between the active and passive voice, represents a deliberate choice. By ensuring that we can produce at most one paraphrase, we ensure that all such choices are made explicit and available to the user. Note that simply being able to restrict ourselves to a single paraphrase does not oblige us to do so. Ideally, the paraphrase selection mechanism should allow the user both to make explicit choices on all alternatives or to cede some of these decisions to the realiser (SURGE, notably, provides this flexibility [Elhadad and Robin, 1999]). In any case, the option to make the explicit choices should always be available.

6.2 Selection mechanism

Paraphrase selection can largely be performed by constraining the lexical selection. We will extend the lexical selection process and the input semantics to express such constraints, but before going into the details, let us mock up a small usage scenario. Consider the input semantics below. We will imagine a hypothetical grammar which associates it with three paraphrases:

(24) *l1:give(e,j,b,m), l2:john(j), l3:mary(m)*
John gave the book to Mary.
Mary was given the book by John.
The book was given to Mary by John.

To restrict the selection, we might specify that the literal *l1:give(e,j,b,m)* be realised by a verb in the passive voice. We rewrite the input semantics as follows with the expected effects:

(25) *l1:give(e,j,b,m)[PassiveForm], l2:john(j), l3:mary(m)*
~~John gave the book to Mary.~~
Mary was given the book by John.
The book was given to Mary by John.

Adding more properties to the input semantics simply narrows down the resulting output even more. We restrict the literal further still:

(26) *l1:give(e,j,b,m)[PassiveForm, CanonicalToObject], l2:john(j), l3:mary(m)*
~~John gave the book to Mary.~~
~~Mary was given the book by John.~~
The book was given to Mary by John.

These tokens with which we enrich the input semantics are called tree properties. We will now see exactly how they are used (Section 6.2.1 and 6.2.2) and where they come from in practice (Section 6.2.3). Building on these ideas, we will also see how the properties can be used to narrow down the semantics so that we produce at most *one* output (Section 6.2.4).

6.2.1 Enriched lexical items and input semantics

The extensions require that we introduce enriched versions of the lexicon and the input semantics, both taking tree properties into account. The basic idea is that linguists use tree properties to describe lexical items and the surface realiser uses them to filter the lexical selection.

Definition 10 (Tree property)**.** A tree property is an identifier. Some examples of tree property are *PassiveForm* and *CanonicalToObject*.

Definition 11 (Enriched lexical item)**.** An enriched lexical item is a triple $\langle T, S, LTP \rangle$. T and S are the elementary tree and lexical semantics as described in Definition 3 (Page 80). LTP is a set of tree properties.

Definition 12 (Enriched input semantics)**.** An enriched input semantics is a set of enriched literals of the form $L[tp_1, \ldots, tp_n]$, where L is a saturated L_U literal and $tp_1, \ldots, tp_n$ is a possibly empty set of tree properties. As a notational convenience, we omit the square brackets when the set of tree properties is empty.

Definition 13 (Plain input semantics)**.** The plain input semantics is the result of stripping away all the tree properties from an enriched semantics. Given an enriched input semantics ES we say that the plain input semantics is the set of literals of the form L_i where $L_i[tp_1, \ldots, tp_n] \in ES$

6.2.2 Enriched lexical selection

Taking tree properties into account consists of filtering the lexical items so that only those with the desired tree properties are retained. Given an enriched semantics ES, we instantiate the lexicon as usual (Section 4.1.1) and return the set of enriched lexical items such that for each item $\langle T, S, LTP \rangle$:

- (as usual) its instantiated semantics S is non-empty and subsumes the plain input semantics;
- for every enriched literal $L[tp_1, \ldots, tp_n]$ in the enriched input semantics ES, if $L \in S$ then $\{tp_1, \ldots, tp_n\} \subseteq LTP$.

6.2.3 Where tree properties come from (metagrammars)

The tree selection mechanism requires that every lexical item in the grammar "possess", i.e. be associated with, a set of tree properties. There are many ways of achieving this intermediate goal. One possible solution might be manual annotation, but this is neither desirable nor necessary. It is undesirable because realistic TAG grammars are large enough to make such a process error-prone and cumbersome. It is unnecessary in the case of SemFRaG because the annotations are already encoded in another resource, the metagrammar from which it was compiled.

Metagrammars are essentially highly factorised representations of the grammar which can then be compiled into the more explicit, familiar form. The use of metagrammars is motivated by the amount of redundancy that can be found in a typical TAG grammar. The system we use in particular is the XMG metagrammar compiler developed in [Crabbé and Duchier, 2004]. We will not go into the precise details on what makes up such a metagrammar. Basically an XMG metagrammar consists of named tree fragments (dominance and linear precedence constraints) which are combined via conjunction and disjunction. To get an idea what the metagrammar looks like, below is a sample of fragments being combined to generate trees in the family of transitive verbs. They are combined in Figure 6.1.

$$\begin{array}{l} Subject \wedge \\ \quad ((ActiveForm \wedge Object) \vee (PassiveForm \wedge CAgent)) \end{array}$$

In XMG terms, an individual elementary tree can be seen as a conjunction of fragments. Their names look very much like the tree properties we have been using in our examples. Indeed, these names are exactly what SemFRaG uses as tree properties. Every elementary tree with the (names of) the fragments from which the tree is built. These names make up the set of tree properties that the elementary tree possesses.

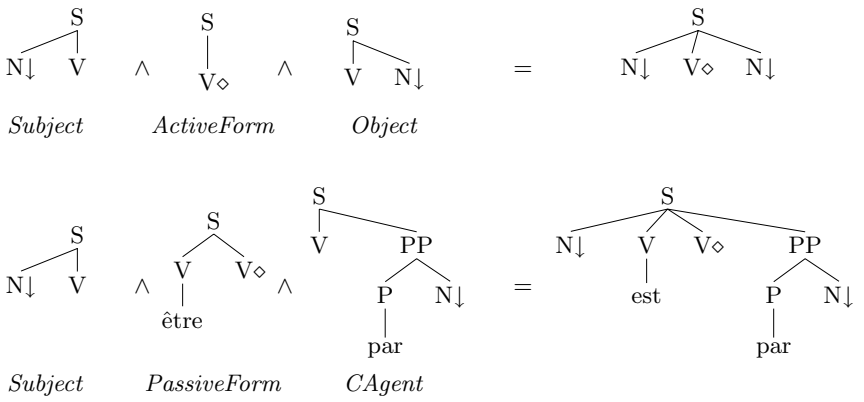


Figure 6.1: XMG Tree fragments (simplified from SemFRaG)

6.2.4 Producing at most one output

The more tree properties used to enrich the input semantics, the fewer lexical items are selected. Pushing this to its logical conclusion, one could conceivably add enough tree properties for each literal to be realised by at most one lexical item each.[1] For the most part having an ambiguity-free lexical selection is enough to guarantee that the surface realiser returns a single paraphrase, the exceptions being inputs where the lack of word order constraints (e.g., intersective modifiers) comes into play. The tree properties associated with each lexical item must *uniquely* identify that item. In other words, each enriched lexical item must be associated with a so-called tree identifier:

Definition 14 (tree identifier)**.** In an FB-LTAG grammar, a tree identifier is a set of tree properties I. If in a set of lexical entries, there is only one enriched lexical item $\langle T, S, LTP \rangle$ such that $LTP \subseteq I$, we say that the tree identifier is unique to the set.

What is important here is not just the fact that tree identifiers are unique. After all, the elementary trees produced by XMG already have unique names like `Tn0Vn1-387` which would technically allow one to achieve the same result. But these do not have the same practical use as proper tree identifiers, the main reason being that they are completely arbitrary, and are not imbued with the same linguistic significance as tree properties. The reason linguistic significance matters is that it gets us closer to the objective of choosing *contextually appropriate* paraphrases. Basically, we need some means of representing linguistic alternatives and selecting from them. We believe that disjunctions in the metagrammar serves as a mechanism for representing the alternatives, and that tree properties provide the mechanism for selecting among them.

[1] We say at most because of lexical items with a multi-literal semantics

6.3 Evaluation

We performed three tests on the possible application of tree identifiers in SemFrag.

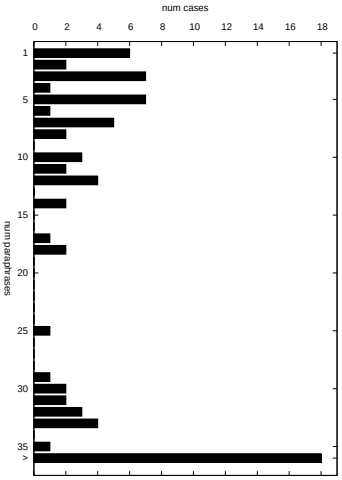
1. As a sanity check, we ran the realiser on a test suite and verified that the grammar can indeed produce paraphrases, i.e. more than one output for an unenriched semantics (Section 6.3.1).

2. Next, we surveyed the grammar to determine how unique its tree identifiers are (Section 6.3.2).

3. Finally, we combined these two results to determine how effective the tree identifiers are for selecting paraphrases in practice (Section 6.3.3).

The test suite has 87 cases. We (i) parsed a set of sentences, building semantic representations for each (ii) hand-selected one semantic representation each[2] (iii) ran GenI on the selected semantic representations and scrutinised the outputs. The cases were selected specifically to highlight the different kinds of paraphrases that the grammar is capable of producing:

- Grammatical variations in the realisations of the arguments (cleft, cliticisation, question, relativisation, subject-inversion, etc.) or of the verb (active/passive, impersonal)
- Variations in the realisation of modifiers (e.g., relative clause vs. adjective, predicative vs. non predicative adjective)
- Variations in the position of modifiers (e.g., pre- vs. post-nominal adjective)
- Variations licensed by a morpho-derivational link (e.g., to arrive/arrival)



6.3.1 Paraphrastic power of the grammar

The test suite yields a total of 1 528 sentences, an average of 18 confirmed paraphrases[3] per case (ranging from 1 to over 50 confirmed paraphrases). The figure in the margin gives a more detailed description of the distribution of the paraphrastic variation. We also grouped the test cases by their complexity, partitioning the input into cases with one, two and three finite verbs. We found that 42% of the sentences with one finite verb accept 1 to 3 paraphrases (cases of intransitive verbs), 44% accept 4 to 28 paraphrases (verbs of arity 2) and 13% yield more than 29 paraphrases (ditransitives). For sentences containing two finite verbs, the ratio is 5% for 1 to 3 paraphrases, 36% for 4 to 14 paraphrases and 59% for more than 14 paraphrases. Finally, sentences containing 3 finite verbs all accept more than 29 paraphrases.

[2] A sentence may sometimes include wrong parses, or just be ambiguous

[3] Paraphrases were validated by hand, so this figure excludes the overgeneration in the grammar.

6.3.2 Uniqueness of tree identifiers

We first determined to what extent the tree identifiers in SemFRaG are unique, essentially by counting tree schemata and tree identifiers. Our findings are that

1. Out of 6414 tree schemata, 2258 (35%) have the same tree identifier as another schema in the same tree family (see Section 4.2.1, Page 85 for a description of tree schemata and tree families).

2. Out of 4156 distinct tree identifiers, 1216 (29%) have duplicates in the same family.

3. Out of 95 tree families, 50 (53%) have non-unique tree identifiers. This makes 85% of the families with more than one member than have non-unique identifiers.

This spells rather bad news for using SemFRaG's tree identifiers for paraphrase selection; however, we found that the mechanism gives better results than these initial numbers would lead one to hope for. The test suite (described below) does not use the whole grammar, but the tree schemata counts for the subset of the grammar used are quite similar:

4. Out of 4730 tree schemata, 1556 (33%) have the same tree identifier as another schema in the same tree family.

5. Out of 3174 distinct tree identifiers, 878 (28%) have duplicates in the same family.

6. Out of 55 tree families, 39 (71%) have non-unique tree identifiers. This makes 89% of the families with more than one member than have non-unique identifiers.

6.3.3 Effectiveness of tree identifiers

In Section 6.3.2, we saw that a large number of tree identifiers in SemFRaG (29%) are not unique. To evaluate the impact of this, we revisited the paraphrased produced from the test suite. Along with each paraphrase, we had the surface realiser produce a full log including the lexically selected items used to produce that paraphrase and the tree identifier of each lexically selected item. These were then used to answer there questions (summed up in Table 6.1).

Do tree identifiers produce unambiguous lexical selections?

For each test case, we grouped the paraphrases into blocks, where each block were produced from the same enriched input semantics (as mentioned above, the enriched input semantics were inferred from the verbose surface realiser output). Out of 743 possible enriched semantic inputs, 603 of them (81%) produced an unambiguous lexical selection. Grouping these results by test case, we found that 64 out of 87 unenriched inputs (74%) produced unambiguous lexical selections using the tree identifiers.

tree identifier ⇒ unambiguous lex sel			
identifiers	603	743	81%
test cases	64	87	74%
unambiguous lex sel ⇒ 1 paraphrase			
identifiers	587	603	97%
test cases	58	64	91%
tree identifier ⇒ 1 paraphrase			
identifiers	587	743	79%
test cases	58	87	68%
pairs	18824	19212	98%

Table 6.1: Paraphrase selection results

Do unambiguous lexical selections produce unique paraphrases?

Focusing now on the enriched inputs which gave us an unambiguous lexical selection, we found that out of 603 such inputs, 587 of them (97%) produced only one paraphrase. Again regrouping these into their unenriched inputs (test cases), having an unambiguous lexical selection led to a unique paraphrase in 58 out of 64 (91%) cases. Note that due to spurious ambiguities, we treated multiple instances of the same string as being one paraphrase.

So do tree identifiers produce unique paraphrases?

Putting these results together, we found that out of 743 enriched inputs, 587 produce only one paraphrase (79%)[4] This makes for 58 out of the 87 test cases (68%).

In [Gardent and Kow, 2007] we reported stronger-looking results. The results we had reported still appear to be correct, but they are weighted towards longer inputs. For each test case, we produced the set of all possible pairs of its paraphrases, of which there are 19212 ($\sum_{i=1}^{n} \frac{p_i*(p_i-1)}{2}$, where n is the number of test cases and p_i is the number of paraphrases of test case i). We found that in 18824 out of these pairs (98%) represented cases where the same tree identifier produces different paraphrases.

6.3.4 Some unique paraphrases

Examples (27a–27c) show some tree identifiers from SemFRaG and their potential uses for paraphrase selection. See Appendix B for a full list of tree properties in the grammar.

(27) a. *l1:jean(j)[ProperName]*
l2:aimer(e,j,m)[CanonicalNominalSubject,

[4]This number need not necessarily match the number of cases where an unambiguous lexical selection leads to one paraphrase; it could always be possible that an ambiguous lexical selection converges on the same the paraphrase, so depending on the grammar, the number could be higher.

ActiveVerbForm, CanonicalNominalObject]
l3:marie(m)[ProperName]
Jean aime Marie
~~Marie est aimé de Jean~~

b. *l1:le(c)[Det]*
l1:chien(c)[Noun]
l2:dormir(e1,c)[RelativeSubject]
l3:ronfle(e2,c)[CanonicalSubject]
Le chien qui dort ronfle
~~Le chien qui ronfle dort~~

c. *l1:jean(j)[ProperName]*
l2:promettre(e1,j,m,e2)[CanonicalNominalSubject,
ActiveVerbForm, CompletiveObject]
l3:marie(m)[ProperName]
l4:partir(e2,j)[InfinitivalVerb]
Jean promet à Marie de partir
~~Jean promet à Marie qu'il partira~~

6.4 Possible extensions

6.4.1 Checking for satisfiability of the enriched input

Tree properties and polarity filtering have one thing in common: they reduce the effects of lexical ambiguity by filtering the lexical selection. Unlike polarity filtering, the use of tree properties is "unsafe" in that it is possible to supply a set of tree properties that overfilter the lexical selection, causing the surface realiser to return no results.

It would be useful if there was some way to check if the tree properties used for enrichment are mutually compatible. Basically, this consists in performing surface realisation with both the enriched input and the unenriched one, that is with and without tree properties. If surface realisation fails with enriched input (and succeeds with unenriched one) we know that some of the tree properties are mutually incompatible. In order for this to be efficient, we would need to use polarity filtering because of lexical ambiguity in the unenriched input. In fact, it would be useful to use polarity filtering for even the enriched input, as the user may have opted for partial use of tree properties (leaving some ambiguity behind) and because the lack of word order could still come into play even if the input were unambiguous.

If we know that the enriched input is unsatisfiable it would be useful further still to know why. Localising the faulty tree properties is a potentially difficult task. We would have to find which subsets of tree properties for each literal are mutually incompatible. Trial and error (removing tree properties and checking for satisfiability) does not appear to be the correct approach, because even if the satisfiability check had a low cost, there would be an exponential number of subsets (for each literal!) of tree properties to search through. This is still an open question.

A possible answer might be to use an SFG network in the hope that some of the knowledge encoded in the network is about the compatibility of tree properties. It is not yet clear to us how to go about using such a network

in the first place. For example, at what level of granularity do SFG network traversals correspond to the input semantics? Presumably, we would need to have multiple (or recursive) traversals, (although perhaps not to the point of making one traversal per literal) and it remains to be seen how independent each individual traversal is of the other. In a sense, the less independent they are, the better, because the more constraints are encoded within.

6.4.2 Global constraints

The selection mechanism relies on "local" constraints, in the sense that each literal of the input semantics has a set of tree properties which filter the lexical selection for that specific literal. One possible shortcoming to this approach is that it does not allow us to express constraints for an *entire* lexical selection, as opposed to the selection for a single given literal. Some global constraints might be stylistic, for example requiring that a sentence use no more than two passive verbs.

In GenI, we have implemented an alternative "global" constraint mechanism which makes use of polarity filtering to restrict the lexical selection as a whole. The rough idea is to extend the input specification language to accept a set of global tree properties. Any global property which is specified on the input semantics will be assigned a negative polarity. GenI will then associate every selected lexical item that possesses the property with the equivalent positive polarity. As a result, the only lexical combinations that pass polarity filtering will be those where exactly one of the lexical items may possesses the desired property.

Another variant of this idea is to extend the input semantics with a set of global tree features. Tree features might be attractive for use by strategic generation components, because they provide a potentially language-independent mechanism for paraphrase selection. As opposed to tree properties, tree features are instantiated with content from the input semantics. An example of a tree feature is `focus:d`, indicating that the focus of the elementary tree is on the semantic index `d`. The essential difference between tree properties and tree features is that the latter allow for paraphrases to be selected on semantic criteria. For example, the input semantics *l0:chase(c,d,c), l1:dog(d), l2:def(d), l3:cat(c), l4:def(c)* could be realised in at least two ways:

(28) a. The dog chases the cat.

b. It is the cat that the dog chases.

The first one puts the focus on `d`, whereas the second puts it on `c`. By specifying `focus:d` or `focus:c` in the lexical selection, we could preselect one or the other. We could also have done it using tree properties (for example, by specifying that *l0:chase(c,d,c)* be realised by a `CleftObject`), but this requires knowing about the syntax of the language in question.

To implement global tree features, we would also need to extend the lexical items. Each lexical item would be augmented with an interface, in addition to its elementary tree and semantics. The interface holds features which can

be thought to be "global" to the tree.[5] The use of global tree features and interfaces would also involve polarity filtering. Given the set of global tree features G, and a lexical item with interface I, for each pair $a:k$ such that $a:k \in (I \cap G)$, the lexical item is assigned a polarity of $+1$ $a:k$ in addition to its other polarities (see Chapter 5). Similarly, the polarity automaton is initialised with a polarity of -1 $a{:}k$ for each $a{:}k \in G$.

Yet another variant still might be to use product polarities for global tree properties/features, that is, to use the same "soft" polarities we use for controlling adjunction instead of the more demanding sum polarities.

6.4.3 Default tree properties

Most surface realisers have some mechanism for choosing a paraphrase by default, when no information is specified to do so explicitly. RealPro does this by marking some feature values as default. SFG-based realisers follow the systemic theory, which also that in some systems a feature might be unmarked and chosen by default, whereas its alternatives might be marked and chosen for some explicit reason. Finally, the statistical realisers do this by simple virtue of ranking; one paraphrase naturally bubbles up to the top and can thus be considered the default. GenI has no such mechanism. If the input semantics is not enriched or not sufficiently enriched, all possible paraphrases are returned without any means of saying if one should be preferred to another. A simple workaround would be to add a statistical ranker to GenI (see Section 6.5.3).

Then again, it might also be possible to accomplish the same task by using linguistic information encoded in the grammar. We could go about this by simply marking one tree in every TAG family as default. If no tree properties are provided to constrain the choice for that lexical item, the default is selected. But this solution has three shortcomings. First, it does not account for partially constrained lexical selection, i.e. where some tree properties are assigned to filter the selection for a given literal, but not down to only one lexical item. Second, the defaults chosen for two different literals may somehow conflict in that they select for lexical items that are mutually incompatible. Finally, it is not clear how we would go about encoding this information into the metagrammar (the metagrammar tells us about classes, not about trees).

A gentler approach might be to group the tree properties into mutually exclusive sets, making it so that an elementary tree may contain no more than one property from a given set. An example of a mutually exclusive set would be { `ActiveForm`, `PassiveForm` } — a tree may not be both active or passive. We could then mark one member of each tree property set as a default (here, `ActiveForm`). This allows us to effectively encode "defaultness" into the metagrammar, in that we can automatically calculate a defaultness score for every lexical item, for example, by counting the number of default tree properties it possesses. And since we are working directly from the metagrammar classes and not trying to refer directly to the trees, it is much simpler to encode.

Ranking the lexical items by defaultness means that we can also select from a partially constrained lexical selection by simply picking the most default

[5] In practice, the interface is used for anchoring a syntactic lexicon entry to its tree schema. Both tree schemata and the lexicon entry have an interface, and when the lexicon entry is selected, its interface is unified with that of its tree schema. The proposed modification is thus to retain the interface (instead of discarding it, as we normally do).

item from the ones available. The notion of ranking could be used to solve the problem of mutually incompatible defaults. The problem is that we can not determine if two default lexical items will be compatible without using them for surface realisation (the polarity filtering mechanism only detects some incompatibilities). So, rather than using defaultness to restrict the lexical selection, we could use it as a basis for ranking the outputs. That is, we perform lexical selection and realisation as usual (with or without the tree properties mechanism from this chapter) and return the paraphrase that has the most default lexical items.

In this version of the proposal, we naively assume that defaultness of tree properties are mutually independent. But it is not clear that this is the case in practice. Defaults are harmless because they only act as soft constraints (they change the ranking of items), but even soft constraints would be nicer if they were more fine-grained. This notion could perhaps be improved upon, again by referring to an SFG network, the idea being that different branches of the network invoke different sets of defaults.

6.5 Related work in paraphrase selection

6.5.1 RealPro: Meaning Text Theory in practice

The Meaning Text Theory (MTT), on which RealPro is based, divides generation into seven layers: a semantic representation is translated into a deep syntactic structure, which is then converted to a shallow one which is in turn translated into a (deep/shallow) morphological structure and then a (deep/shallow) graphical one. Each one of these could in theory be handled by a individual component. RealPro in particular starts from a Deep Syntactic Structure (DSyntS), i.e. a dependency tree with content words and features as nodes and labelled dependencies as arcs [Lavoie and Rambow, 1997].

```
SEE []
( I boy [ number:pl ]
     ( ATTR THIS1 )
  II Mary [ class:proper_noun ] )
```

For example, the DSyntS above uses the words "see, boy, this, Mary", putting "boy" in the subject argument (`I`) of "see" and "Mary" in its object argument (`II`). The resulting sentence is "These boys see Mary".

RealPro supports a high amount of syntactic variation. Many of the grammatical variations can be controlled by specifying different features. Setting the `question` feature to `-` in the DSyntS above, for example, would produce the sentence "Do these boys see Mary?". Similarly, the sentences below can be generated from more or less the same DSyntS, except that in (29a), the word "often" is associated with `starting-point:+`, and in (29b), it is associated with `rheme:+`. The features are not obligatory either; if the user does not specify any, they will be insert by a system of defaults, for example, inserting a `pre-verbal:+` feature to produce (29c).

(29) a. Often, John eats beans

b. John eats beans often

c. John often eats beans

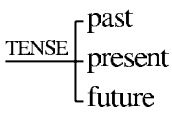
RealPro does not have any non-determinism, so a DSyntS could not be used to underspecify for a set of paraphrases. Moreover, the range of sentences that it would associate with the same DSyntS (modulo features) is smaller than what GenI would consider as paraphrases. For example, to produce the sentence "Mary is seen by these boys", we would have to swap the roles (I) and (II) in the DSyntS above, so that "Mary" is the subject and "boy" is the object. Of course, this should not be interpreted as a shortcoming in RealPro, but a sign of modularity. It merely reflects a different division of labour with an extra emphasis on *surface* realisation, and a sentence planner which does more of the paraphrase selection work. In highly approximative MTT terms, GenI is more concerned with converting semantic representations into deep morphological structures — the morphological system in GenI has been somewhat neglected, with lemmas being produced instead of inflected forms. On the other hand RealPro starts "later" in the pipeline, but reaches all the way to the end of processing; it can inflect sentences, add punctuation and even produce formatted output, for example, via HTML.

That being said, the linguistic resources used by RealPro consist of rules that pattern-match on fragments of DSyntS to produce fragments of SSyntS (Surface Syntactic Structure) in its place, and similar rules for matching on SSyntS and so on [Reiter and Dale, 2000]. While this is highly suited to the task at hand, it does not seem to lend itself to reversibility. In general, parsing in MTT does not yet appear to be very well understood, so it would be worth revisiting the question in the future.

6.5.2 SFG based realisers

A lot of work in generation is based on Systemic Functional Grammar (SFG). SFG and the larger tradition of Systemic Functional Linguistics are framed around the use of language to achieve goals, be they social, linguistic or otherwise. Here we focus on linguistic goals. As an example of a linguistic goal, the function of a noun group might be to serve as the subject of a clause. What makes systemic grammar different is that they map from function to linguistic structure (whereas traditional grammars can be seen as doing the opposite). SFGs are often seen as highly suitable for generation, because the inputs used for generation are closer to linguistic goals than they are to linguistic structures.

At the heart of every SFG is a collection of systems. Each system represents a minimal grammatical alternative, a mutually exclusive choice between one use of language or another. In the margin, we an example of a system named TENSE. It presents a choice between three features, 'past', 'present' and 'future'. Systems can be connected together to form a network. Figure 6.2 a network in which the TENSE system participates. The interconnection between systems makes it such that one choice leads to another. For example, choosing the 'interrogative' feature in the INDICATIVE-TYPE system leads to the choices within the INTERROG-TYPE system. The connections between systems can be quite rich. Sometimes a single choice will fork into multiple, parallel systems. For example, choosing the 'indicative' feature in the MOOD system opens up the INDICATIVE-TYPE and TENSE systems which have to be traversed in parallel.



Every feature in an SFG can be associated with operations for putting output fragments together. What these fragments consist of and how they combine varies from implementation to implementation. It might entail a simple string concatenation or more sophisticated operations like feature unification or tree adjunction. To use an SFG for surface realisation, one supplies an input specification, telling how to traverse the network (which choices to make) and what initial output fragments (lexical items) to use. The input specification can be recursive, containing other input specifications, each of which corresponds to a separate traversal of the network. SFG-based realisation thus consists of performing these traversals, collecting the realisation instructions it encounters along the way and using them to stitch all the output fragments together.

In this section, we compare GenI with three SFG-based surface realisers, KPML, SURGE and ISOFT.

KPML/Nigel

The Komet-Penman Multilingual system (KPML) is an SFG realiser and grammar-development tool.[6] The system is often used with Nigel, a large systemic grammar of English under development since the late 70s.

KPML supports on the one hand a more abstract level of input than GenI. The following two sentences in a KPML-based grammar would have the same core semantics (whereas in GenI, they would probably be distinct):

(30) a. **Lions** are almost extinct [species consisting of individuals]

b. The **lion** is almost extinct [species as class]

Selecting between the two paraphrases in KPML would consist of specifying if species of lions is being "denoted with respect to some relation or relations among its features, sub-species, or any other aggregation as a multiplicity of structural aspects."

Whilst much work in KPML has been done in dealing with such high-level concepts, less attention has been paid to the lower level syntactic aspects of

[6] Penman and Komet were respectively an early SFG surface realiser and some systemic grammars for various languages

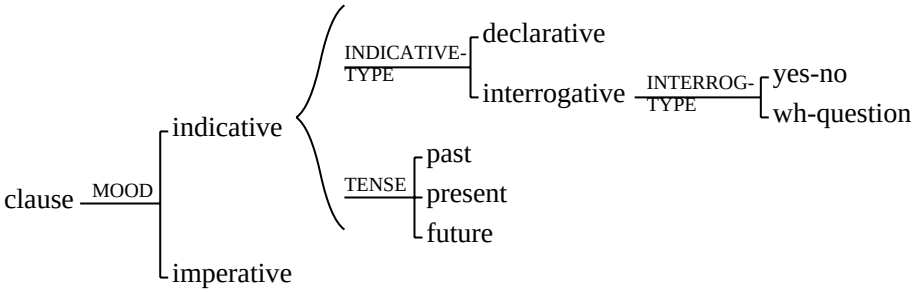


Figure 6.2: An SFG network for mood

grammar [Bateman *et al.*, 1992]. A key problem lies in the representation of linguistic structure. In the KPML version of SFG, output fragments are strings, and the instructions for combining them together are realisation statements which impose an order on them. This does not allow for complex linguistic structures to be built in a natural way. In an example from [Yang, 1992], one might expect that the sentence "Who did John believe the tall man saw?" be built from the fragments "John believed X" and "Who did the tall man see?". But realisation statements can only order fragments and not nest them. As Yang argues strings are not appropriate for use as base linguistic structures. Something richer, like TAG would be a better choice. TAG also has the advantage of being syntax-oriented, which means that it can easily be used for parsing, which SFG cannot.

Finally, a third difference between KPML and GenI is that the former does not return all of the paraphrases for a given input. KPML uses a system of defaults so that if an input is underspecified the default choice is always selected. This is mainly a question of efficiency; one could always simulate returning all the results by generating the space of possible KPML inputs, but the size of the search space would be exponential with respect to the number of possible choices. Something like chart generation would be needed to make this more efficient.

FUF/SURGE: Systemic grammar in a unification framework

FUF (Functional Unification Formalism) can be seen as a grammar formalism, a programming language or as a natural language generator. [Elhadad, 1991]. FUF grammars are essentially feature structures with disjunctions.

Using FUF as a generator consists of unifying an input specification with the grammar to get a syntactic structure and linearising that structure to get a sentence. FUF unification is a recursive process. When we unify the input with the grammar, the resulting feature structure may have sub-constituents (either from the original input or added in by the grammar); these sub-constituents are then recursively unified with the grammar until there are no more sub-constituents left.

The most notable grammar for FUF is SURGE (Systemic Unification Realisation Grammar of English) [Elhadad and Robin, 1999]. The grammar is based mainly on Systemic Functional Grammar, although it also has some influences from HPSG and other sources. No special mechanisms are introduced to implement the system network. Choice points are encoded as FUF alternatives, subsystems as subfeatures, and system traversal as standard FUF unification.

FUF/SURGE supports the interspersing of output fragments in a way that KPML does not. Whereas KPML manipulates strings as output fragments, FUF manipulates patterns. A FUF pattern is a special feature used to constrain the order of constituents. An example FUF pattern might be `prot verb goal` which says that the `prot` constituent must come directly before the `verb` one, which must itself come before `goal`. What makes FUF patterns interesting is that they can also contain "dots", so that for example, the pattern `prot ... verb` now says that `prot` must come *anywhere* before `verb`. The pattern features are treated with special unification rules described in [Elhadad, 1991].

FUF grammars can be seen as being much more declarative than KPML realisation statements (the order that realisation statements are invoked in becomes important in KPML), but they cannot yet be considered bidirectional. In other words, it is not yet clear how to use SURGE and other FUF-based grammars for parsing. The problem seems to be how one would map the surface string on to FUF's output patterns. On the other hand, the L_U-augmented FB-LTAG accepted by GenI can be used for both parsing and generation.

ISOFT: Combining SFG with TAG

ISOFT (Integrated System of Functional-systemics and TAGs) can be seen as using SFG pre-processing in a TAG generator. Alternatively, it could be seen as using TAG as a syntactic back-end to an SFG generator. ISOFT retains the core algorithm from other SFG-based systems: it breaks the input specification into fragments, performs an independent network traversal and realisation for each fragment, and reassembles the resulting pieces of output.

In the place of strings as output fragments, ISOFT uses TAG elementary trees. ISOFT can use the backbone of systemic network and some core realisation statements, such as KPML's `conflate` and `preselect`. Other structure-building operations are stripped away and replaced with instructions that select and instantiate elementary trees. Once selected, the elementary trees are combined using TAG substitution and adjunction. The processing strategy can be summed up into two steps; a "descent" phase which amounts to building a TAG derivation tree from the top-down (using the systemic networks to select the elementary tree for each node), and a bottom-up phase which assembles the derived tree according to specification.

ISOFT distinguishes itself by making a clear separation of concerns between the functional aspects of the grammar and the syntactic ones. This is partly achieved by splitting the network itself into two pieces; the first one focuses on purely functional choices, and the second (the TAG network) on syntactically oriented ones like verb transitivity. The functional network is navigated on the basis of the input specification, and the syntactic network on the choices made in the functional traversal. The TAG network is a mechanism for associating a set of functional choices with a TAG elementary tree, c.f. GenI's lexical selection mechanism. As Yang argues, separating the networks in this way increases their modularity, which makes them easier to design, maintain and verify.

ISOFT is less mature than KPML and SURGE, but its approach is promising. The key lesson from ISOFT that it is possible to use the systemic network to guide the selection of elementary trees. GenI only has tree properties for now, which is a first step, but lower level than what a systemic network could offer.

On the other hand, there are two details which could stand to be fleshed out: adjunction and non-determinism. ISOFT does adjunction in two different ways. When the auxiliary tree is a modifier, the adjunction site is selected by walking up the path from the anchor to the root and picking the first available site. When the auxiliary tree is a functor, the site is selected by the tree being adjoined to, according to the functional features assigned to that tree. The overall scheme appears to be somewhat complicated and *ad hoc*[7] and should perhaps be replaced by a more principled approach. As for

[7]The adjunction of functors is handled as follows: In the lexicon, each candidate adjunc-

non-determinism, the ISOFT lexical selection never allows for ambiguity. If the input is underspecified, the choice of an elementary tree is forced by a systemic network default. On the other hand, it may be useful not to rely on defaults, but to actually deal with the non-determinism (for example, returning the set of possible paraphrases) and do so in an efficient manner. GENI has something to offer for both issues: a clearly defined syntax/semantics interface with L_U-augmented FB-LTAG, and a chart generation algorithm with polarity filtering. Future research should attempt to combine the advantages of the two systems.

6.5.3 Statistical surface realisation

Statistical methods for generation are relatively recent introductions, both with respect to generation and to statistical NLP. The most common approaches seem to be variants of the two-stage architecture by [Knight and Hatzivassiloglou, 1995]: first, a symbolic generator produces a packed representation of the possible paraphrases, and then a statistical ranker chooses the most probable one. The packed representations have become increasingly compact and sophisticated — from the original word lattices to packed forests [Langkilde, 2000] and perhaps better — but the basic approach remains the same.

The observation that gave rise to these first statistical realisers (NITROGEN and its successor HALOGEN) was that symbolic realisers, e.g. KPML, require a "daunting amount of linguistic detail" in their input [Langkilde, 2000]. Such detail is often not available to client applications, so their developers often resort to simpler template based generators, which are less demanding, but also less flexible. The key insight is that corpus based linguistic knowledge often can make up for gaps in linguistic knowledge, both in the forms of (i) imperfect "grammar, ontology, lexical, collocations and mappings between them" [Knight and Hatzivassiloglou, 1995] and (ii) underspecified input. Statistical realisers show us that it is possible to do surface realisation without a lot of linguistic knowledge (HALOGEN uses 255 "mapping rules", which is considerably lighter weight than SEMFRAG's 6000 tree schemata).

The experimental results are encouraging too: [Langkilde-Geary, 2002] converted section 23 of the Penn Treebank into a set of inputs, of which HALOGEN was able to produce an output for 80% of the inputs, with a 94% accuracy for fully specified inputs (measured with the NIST simple string accuracy metric and BLEU scores) and 51% accuracy for minimally specified ones (with no determiners, adjunct markers, properties such as tense and voice).

The main thing that HALOGEN and GENI have in common is that they both embrace the idea of providing a variable degree of control over the input. The user can specify a lot of linguistic detail to get exactly the sentence desired, or specify less detail to let the realiser choose. In a sense, this choice represents two distinct tasks, ensuring grammatical output and picking a good default. HALOGEN essentially uses its statistical component for both tasks. GENI does

tion site of every elementary tree is annotated with a condition (e.g., `wh-q`). The conditions are fulfilled if the elementary tree is associated with the corresponding functional feature during lexical selection. If the condition is fulfilled, the annotated site is where adjunction takes place. For example, the conditions that annotate candidate adjunction sites can migrate to account for multiple adjunction. But if this is the case, what happens when, as the result of migration, there is more than one node annotated by the same conditions? Pointers would be appreciated if this actually is answered in [Yang, 1992] or elsewhere.

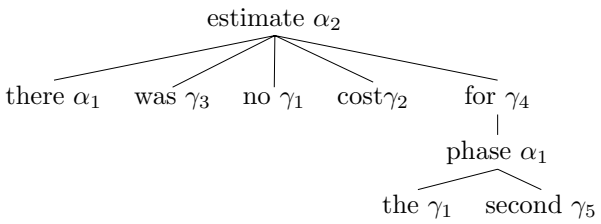


Figure 6.3: FERGUS annotated dependency tree

not do the latter task; it returns all paraphrases with no preferences, which is as good as making an arbitrary choice. As for ensuring grammaticality, it uses a relatively rich and constrained grammar.

Two things are worth pointing out about these differences. First, a rich grammar is inherently interesting to have from a linguistic point of view. This is not so much a theoretical argument — we are in the surface realisation business and not that of building linguistically interesting objects — but a practical one in the sense that the grammars have to be built anyway so the time spent putting them to use is not really wasted. Second, as [Langkilde-Geary, 2002] points out, language models richer than n-grams over words can provide for greater accuracy.

Statistical realisation with rich syntax

NITROGEN and HALOGEN have since inspired a new breed of statistical realiser that combines grammars from parsing-oriented formalisms like HPSG, CCG and TAG with a statistical ranker. The LKB generator, includes a "selective unpacking" step where the only the best paths through the generation forest are expanded [Carroll and Oepen, 2005]. OPENCCG incorporates an optional "anytime search" feature that prunes from the chart the worst-ranked intermediate structures according to n-gram based scoring [White, 2004].

A system which perhaps deserves special attention is FERGUS, a TAG based statistical surface realiser [Bangalore and Rambow, 2000b]. FERGUS accepts as input a dependency tree. It works in three phases:

1. Perform statistical lexical selection, using a process akin to supertagging, to associate each lexical item with an elementary tree. The result is an annotated dependency tree where each node (i.e. word) is associated with an elementary tree.

2. "Unravel" the derivation tree into a word lattice.

3. Statistically rank the paths in the output lattice.

The annotated dependency tree that FERGUS produces in its first step allows for a large amount of variation in the word lattice. To begin with, they do

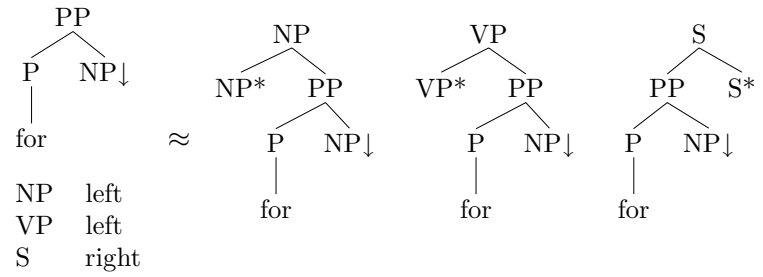


Figure 6.4: Gamma trees can be thought of as underspecified auxiliary trees

not specify what are the substitution and adjunction sites used in each operation. In Figure 6.3, we know that γ_2 ("for") was adjoined into α_2 ("estimate"), but we do not know where. Furthermore, FERGUS uses so-called gamma trees instead of the standard TAG auxiliary trees. Gamma trees are like auxiliary trees, except that they do not fully specify what kind of node they may adjoin to, nor on what side. Instead they provide a list of possible nodes/sides in which adjunction may take place. So not only would we not know where γ_2 adjoins to, we do not know exactly what kind of node (S? NP? VP?), nor on what side. Such decisions are made by the statistical ranker.

GENI is basically one corpus and ranking module away from being a full-fledged statistical realiser. It would be useful to follow the example of the FERGUS and the LKB generator by adding a corpus-based post processing step. This would be one possible solution to the intersective modifiers problem, for example. Having a unique lexical selection does not always guarantee a single paraphrase because adjuncts can be included in any order. A statistical ranker would be useful for choosing the most natural one when other linguistic knowledge is not available. Similarly, it could also be used in conjunction with the tree constraint mechanism we propose in this chapter; if there is not sufficient information to fully constrain the lexical selection, we could fall back on to statistical ranking as worst case default.

Chapter 7

Reducing overgeneration

As we saw in the previous chapter, the SemFRaG grammar is produced using a metagrammar compiler. XMG, the compiler, assembles a grammar out of a more abstract specification. For debugging purposes, it associates each elementary tree produced with a set of tree properties, allowing the grammar writer to trace a tree back to its metagrammatical origins. We saw in the previous chapter that tree properties also provide linguistic information about elementary trees. Indeed, they can be pressed into service as a set of paraphrase pre-selection filters. In this chapter, we will see how to use the tree properties for their original purpose: debugging (meta)grammars. Specifically, we will use the surface realiser as a tool for identifying overgeneration in the grammar and localising its causes within the metagrammar.

7.1 Overgeneration

A generative grammar should describe all and only the sentences of the language it describes. In practice however, most grammars both under and overgenerate. Undergeneration occurs when a grammar does not recognise all of the strings in the target language. It means that we can fail to parse legitimate sentences or to produce all the valid paraphrases for a semantic input. Overgeneration, on the other hand, occurs when the grammar licenses too many strings. It means that surface realisers like GenI will produce not just valid paraphrases, but unnatural looking sentences as well, strings which either do not belong in the target language or which should not be associated with the semantic input. Overgeneration can also cause parsers to *accept* such strings or pick the wrong parse for a valid sentence. In short, both undergeneration and overgeneration bugs can cause problems with parsing and generation. Here, we focus on overgeneration.

There are several reasons why grammars overgenerate. As is now well-known, grammar engineering is a highly complex task. It is in particular, easy to omit or mistype a constraint thereby allowing for an illicit combination and indirectly, an illicit string. Moreover, a computational grammar is a large object and predicting all the interactions described by even a medium size grammar is difficult, if not impossible.

In our particular case, the grammar is compiled from a more abstract specification, a metagrammar. This helps us to achieve coverage relatively quickly

(it helps us to avoid undergeneration); however, with a higher level of abstraction, we also run the risk of licensing more elementary structures than we had bargained for. Elementary trees in XMG are built by combining "classes" together, and sometimes classes combine in unexpected ways, producing trees that should not be in the grammar, trees that may also induce overgeneration.

This is why a surface realiser that produces all the strings associated with a given semantics is a valuable tool: it permits checking these predictions on concrete examples. A parser is an excellent tool for detecting undergeneration — just give it a sentence and see what happens — but as [Boguraev *et al.*, 1988] points out, it is less suitable for detecting overgeneration, because one would have to invent wrong sentences and try them out. Having a proper generator takes the guesswork out of this process.

7.2 Grammar debugging

The (meta)grammar being debugged is SemFRaG, the French FB-LTAG presented in Chapter 4. We have observed with SemFRaG that the wrong sentences for a given input semantics tend to be wrong in a similar way. This led us to believe that it is typically a small number of flaws which lead to a great number of errors, and that systematically identifying what the wrong sentences have in common will reveal these flaws to us. The idea is to first, (manually) annotate the realiser output as being wrong or not, and then to use the annotated data to automatically spot the items (e.g. elementary trees, tree properties) which systematically occur only in overgeneration cases. More specifically, the procedure we defined to reduce overgeneration can be sketched as follows:

1. Surface realisation is applied to a graduated test suite of input semantics thus producing a detailed derivation log of all the derivations associated with each input in the test suite

2. The outputs given by the derivation log are (manually) classified into PASS or OVERGENERATION sentences, the overgeneration mark indicating strings that either do not actually belong in the target language, or should not be associated to the input semantics.

3. The annotated output is used to automatically produce a suspects report which identifies a list of suspects i.e., a list of TAG trees or derivation steps which are likely to cause the overgeneration because they only occur in overgeneration cases.

4. The grammar is debugged and re-executed on the data

5. The derivations results are compared with the previous ones and any discrepancy (less or more sentences generated) signalled.

In a sense, this is an approach that might already be widespread in generation: produce some output, and correct the grammar possibly with the aid of a derivation log. Our contributions are a systematic, incremental approach; a high level of automation, which increases our throughput by focusing human attention on correcting the grammar rather than the unrelated details;

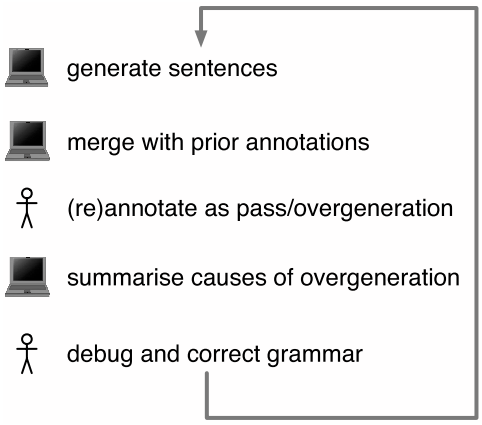


Figure 7.1: Test harness

and summarisation of the derivations log, which makes it easier to identify the source of error.

7.3 An incremental approach

First experiments with SemFRag showed that the grammar strongly overgenerates both because it was initially developed for parsing and because it is automatically compiled from an abstract specification. Indeed for some inputs, the realiser produced over 4000 paraphrases, a large portion of them being overgeneration. More generally, the number of outputs for a given input varies between 0 and 4908 with an average of 201 outputs per input and median of 25.

To avoid having to manually annotate large amounts of data, we relied on a graduated test suite (the one from Chapter 6) and proceeded through the data from simplest to more complex. We focused on one test case at a time, eliminating its overgeneration before moving on to more complex ones. This means we incrementally reduced the overgeneration at each case, thereby diminishing the number of output to be annotated in the next.

We enforced this incremental discipline with the help of a test harness interleaving manual annotations with machine-generated output. Three points are worth stressing. First, the suspects report is produced automatically from the annotated derivation log. That is, except for the derivation log manual annotation, the identification of the suspects information is fully automated. Second, regression testing is used to verify that corrections made to the grammar do not affect its coverage (all pass are still produced). Third, the harness provides a linguist friendly environment for visualising, modifying and running the grammar on the inputs being examined.

```
output: c'est paul demander jean qu'il vient
venir:Tn0V:n5 <-(a)- demander:Tn0Vs1int
venir:Tn0V:n4 <-(s)- paul:Tpropername
demander:Tn0Vs1int:n3 <-(s)- jean:Tpropername

output: c'est paul que demander jean il vient
venir:Tn0V:n8 <-(a)- demander:Tn0Vs1int
venir:Tn0V:n4 <-(s)- paul:Tpropername
demander:Tn0Vs1int:n3 <-(s)- jean:Tpropername

demander Tn0Vs1int-6199
  CanonicalSententialObjectInterrogativeFiniteWithoutComplementizer
  InvertedNominalSubject SententialInterrogative
venir Tn0V-6686
  CleftObject ImpersonalSubject NonInvertedNominalSubject
  activeVerbMorphology
jean Tpropername-2472
paul Tpropername-2472
======================================================================
output: c'est jean qui demande il vient paul
demander:Tn0Vs1int:n10 <-(s)- venir:Tn0V
demander:Tn0Vs1int:n4 <-(s)- jean:Tpropername
venir:Tn0V:n4 <-(s)- paul:Tpropername

demander Tn0Vs1int-6182
  CanonicalSententialObjectInterrogativeFiniteWithoutComplementizer
  CleftSubject NonInvertedNominalSubject SententialInterrogative
venir Tn0V-6678
  CanonicalObject ImpersonalSubject NonInvertedNominalSubject
jean Tpropername-2472
paul Tpropername-2472
```

Figure 7.2: A derivation log

7.3.1 The derivation log (steps 1 and 3)

The derivation log produced by GenI contains detailed information about each of the derivations associated with a given input. More specifically, for each generated string, the derivation log will show the associated derivation tree together with the tree family, tree identifier and tree properties associated with each elementary tree composing that derivation tree.

The derivation log (Figure 7.2) tells us where sentences come from, but what it is missing, crucially, is whether these sentences are correct or not. To extract from it information that points more directly to the likely causes of overgeneration, we manually annotate the log, replacing for each sentence, the token `output` with `pass` if we believe the sentence incorrect or `overgeneration` otherwise.[1] This sounds tedious, but is more tolerable than one may think

[1] We were fairly liberal with our annotations in that we erred on marking things as pass if there was any uncertainty.

```
<suspects-report> ::= <sr-item>*
<sr-item> ::= (<lemma-item> <EOL>)*
              (<derivation-item> <EOL>)*

<lemma-item> ::= <lemma> <EOL>
                 <family-item> <EOL>
                 (<tree-item> <EOL>)*
<lemma>       ::= <string>
<family-item> ::= <tree-family> "(all)"? <tree-property>*
<tree-item>   ::= " [" <tree-number> "]" <tree-property>*

<derivation-item> ::= <tree-1> <arrow> <tree-2>
<tree-1>  ::= <tree-id> ":" <node-number>
<tree-2>  ::= <tree-id>
<arrow>   ::= "<-(" <op> ")-"
<op>      ::= "s" | "a"

<tree-id>       ::= <lemma> ':' <tree-family> '-' <tree-number>
<tree-family>   ::= <identifier>
<tree-number>   ::= <number>
<tree-property> ::= <identifier>
<node-number>   ::= <number>
```

Figure 7.3: EBNF for the suspects report

because the annotation process is broken up into manageable chunks by the incrementality of our approach. Each test case only demands that a small number of sentences be annotated; additionally, the real mental effort in each "iteration" of our test harness is spent determining how to correct the grammar. Compared to this, the few minutes it takes to annotate a hundred sentences is barely noticeable.

That said, it is not by poring over these logs that we will uncover the bugs in our grammar. The problem is that these logs can be both long and repetitive. The above example only shows three sentences and it already spans half a page! We have found that the best use of the log is in producing the suspects report.

7.3.2 The suspects report (step 4)

To make full use of the derivation log, we must summarise and synthesise the information within. We do this automatically, using a script that reads the annotated derivation log and outputs a much shorter suspects report. This report identifies "suspects", that is, likely causes for overgeneration. It does not so much summarise the derivations log as extracts the most useful information from it. For each semantic input, the report lists the trees, sets of trees or derivation items that *only* occur in overgeneration, i.e., the strings associated with that input which have been marked as OVERGENERATION.

Figure 7.3 shows the layout of a suspects report as an EBNF grammar. As we can see, the report shows us the sources of overgeneration for each test case in the suite. The test cases are independent from each other. Each test case in the report is represented by a suspects report item, or a SR-ITEM for short. The report gives basically two kinds of information about each SR-ITEM : information about individual elementary trees that occur in overgeneration (and their associated tree properties), i.e. lemma items, and information about interactions between these trees, i.e. derivation items. For convenience, we will say that when something (e.g. a tree property) consistently occurs in overgeneration, it is "bad".

Lemma items Each lemma item consists of a lemma and a set of elementary trees that possess bad tree properties. To simplify the presentation, we assume that a lemma is only associated with one TAG family.[2] The lemma item has basically two parts: information about the whole family, and information about the individual trees within that family.

If all the elementary trees associated with that family are bad (i.e. the ones which are used in the generation process for the case considered and are associated with that family), we signal the fact by printing out `(all)`. This suggests to the user that any problems with this family may come from a single source. Also, we list all of the bad tree properties that occur in all of the bad elementary trees of that family. This is not strictly necessary, but it avoids some redundancy in the suspects report.

The remaining bad tree properties are listed next to the particular elementary tree in which they appear. For example, a line of the form

```
[649] CanonicalGenitive dePassive
```

indicates that the tree properties `CanonicalGenitive` and `dePassive` are bad and they occur in elementary tree number 649. We only print out the tree number, because we are referring to trees in the same family (by convention, XMG-produced elementary trees are associated with a tree number). Note that we sometimes print out a tree number without any tree properties. This occurs when the bad tree properties associated with this tree are common to the whole family and have already been printed out.

Derivation item A derivation item ($t_1^n \xleftarrow{Op} t_2$) refers to an edge in some TAG derivation tree. It indicates that the tree t_2 has been inserted into t_1 at node n using the TAG operation Op (either substitution or adjunction). Only bad derivation items are listed in the suspects report. We can usually learn from derivation items that either t_1 or t_2 are under-constrained, leading to a forbidden adjunction.

Suspect report examples

Displaying the commonalities between suspects makes it easier for the linguist to understand the likely cause of overgeneration. For instance, if all the trees of a given family lead to overgeneration, then it is likely that the grammar is not

[2]The actual suspects report supports more than one family per lemma; each distinct family for the lemma produces a separate lemma item.

sufficiently constrained to block the use of this family in the particular context considered. To better illustrate the type of information contained in a suspects report, let us now go through a few examples.

Example 1: "il faut partir/? devoir partir" Given the input semantics for "il faut partir (we must go)", the suspects report tells us that the presence in a derivation of any trees of the family `SemiAux` leads to overgeneration, and, moreover, that these trees have the tree property `SemiAux`.

```
consistent overgeneration for devoir
TSemiAux (all) - SemiAux
  [506]
```

In this context (i.e., given the input semantics considered), the use of a `SemiAux` tree results in the production of such strings as "devoir partir (to have to go)" which are grammatical but do not yield a finite sentence as output. If desired, this particular overgeneration bug can be fixed by constraining the generator output to be a finite sentence.

Example 2: "Jean dit accepter/*C'est par Jean qui accepte qu'être dit". In the previous example, the SR-ITEM indicates that all trees of a given family lead to overgeneration but there is only one tree in that family. A more interesting case is when there are several such trees. For instance, the SR-ITEM below indicates that all derivations involving an n0Vn1 tree anchored with *dire* lead to overgeneration and that there are 6 such trees (trees 699 ... 750). Moreover the tree properties information indicates that all these trees share the `InfinitiveSubject Passive` tree properties. Inspection of the data shows that these trees combine with a finite form of "accepter" to yield highly ungrammatical strings such as "c'est par Jean qui accepte qu'être dire" (instead of e.g., "Jean dit accepter (Jean says to accept)"). In short, the SR-ITEM indicates that the grammar is not sufficiently constrained to block the combination of the infinitive passive form of the n0Vn1 trees anchored with "dire" with some of the trees associated by the grammar with "accepter".

```
input t90
Lemma: dire
Tn0Vn1 (all) - InfinitiveSubject Passive
  [699] CanonicalCAgent Passive
  [746] CanonicalGenitive dePassive
  [702] CleftCAgentOne Passive
  [752] CleftDont dePassive
  [751] CleftGenitiveOne dePassive
  [750] RelativeGenitive dePassive
```

Example 3: "Jean doit partir/*C'est Jean il faut que qui part" Sometimes overgeneration will only occur with some of a family's trees. In this case the third line of the SR-ITEM indicates which are those trees and their distinguishing properties (i.e. the properties that always result in overgeneration). For instance, the suspects report for the input semantics of "Jean doit partir (Jean must leave)", contains the following single SR-ITEM :

```
Input t30
consistent overgeneration for partir
Tn0V - CleftSubject
  [604]
```

This indicates that all derivations including tree 604 of the n0V family anchored with "partir" lead to overgeneration. Indeed such derivations license highly ungrammatical sentences such as "C'est Jean il faut que qui part" where a cleft subject tree for "partir" combines with the canonical tree for "il faut". This overgeneration bug can be fixed by constraining n0V cleft subject trees to block such illicit combinations.

Example 4: "L'homme riche part/* riche l'homme part" Finally, overgeneration may sometimes be traced back to a specific derivation item, i.e., to a specific tree combination. This will then be indicated in the last line of the trace item. For instance, the following SR-ITEM indicates that adjoining the adjective auxiliary tree `Tn0vA-90` to the root of a determiner tree always leads to overgeneration. Indeed such an adjunction results in sentences where the adjective precedes the determiner, which in French is ungrammatical.

```
Input t70
consistently overgenerating derivation
item
le:Tdet-17:n0 <-(a)- riche:Tn0vA-90
```

7.3.3 The progress report (steps 5 and 2)

Using the suspects report helps the linguist to localise the grammar bugs that lead to overgeneration. Even so, not all of our "corrections" to the grammar have the intended result; sometimes they have no effect on the error or sometimes they introduce errors of their own. Part of our protocol is thus to run the test harness on the same test case until all sentences are marked PASS. Each time we run the surface realiser, GENI produces a new unannotated derivation log. We use a small script to import the annotations from the previous log into the new one. The script also produces a progress report which lets the user know at a glance if her modifications had the intended effect.

```
New output?
  jean dit c'est l'homme volontaire qui part

Oops! We lost these passes:
  jean dit l'homme volontaire part

Hooray! no longer overgenerates:
  dit part l'homme volontaire jean
  dit jean part l'homme volontaire
```

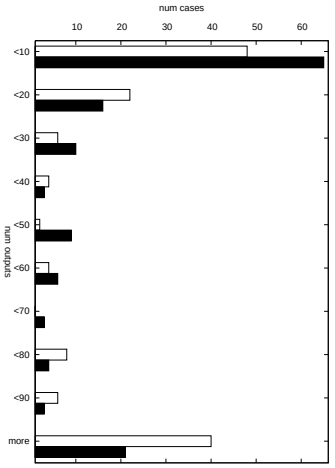
In this (contrived) example, we report three things to the user:

- If there are any new sentences are produced by the grammar. These are sentences for which we do not have annotations. This is a fairly rare occurrence, and it is not necessarily an error. However, we signal these so that we can ensure that the user will add the appropriate PASS or OVERGENERATION annotations.

- If any previous PASS sentences are now missing from the output. This might indicate an overzealous correction to the grammar. It might also mean that we were wrong to annotate the sentence as PASS the first time, but in any case, it is worth knowing.
- If any previous OVERGENERATION sentences are now missing from the output. This helps the user to verify that her modifications have had the desired effect.

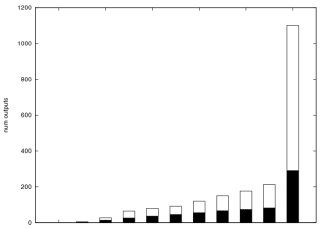
7.4 Evaluation and results

7.4.1 Before and after



We used the test harness over a period of one week, roughly 12 consecutive man hours. Over that period we ran over ten iterations of the test harness, making 13 modifications (30 lines) to the grammar. In the process of revising this grammar, we studied 40 cases (under one third of the whole suite) and manually annotated 1389 outputs with pass/overgeneration judgements. On the whole 140 cases of the test suite, the original grammar produced 28 167 outputs (4908 for the worst case, 201 mean, 25 median). The revised grammar produces 70% fewer outputs, leaving behind 8434 sentences (201 worst case, 60 mean, 12 median). These numbers are repeated below for convenience:

	total	max	mean	median
before	28167	4908	201	25
after	8434	710	60	12



The reductions ranged from 42% in our smallest inputs to 74% in our largest ones. We can see these reductions in the figure in the margins, where we focus purely on the number of outputs that are produced by each test case (without looking at their distribution in the suite). The cases are sorted and grouped together by the number of outputs they produce in the new grammar. We see in the graph how many sentences each group of test cases produced on average before and after, notably that the difference in outputs is greatest in the larger test cases. The key idea is that sources of overgeneration will combine together and that eliminating these sources can have a great impact.

Also, it is very well to be cutting out overgeneration, but only so long as we are not cutting out linguistically valid sentences along the way. The test suite had been built semi-automatically by parsing some sentences and hand-picking the valid semantic representations among the proposed outputs. As a sanity check, we reparsed the original sentences with the new grammar and found that 136 out of 140 sentences were parsed successfully, 4 less than with the original grammar. The difference was due to an over-restrictive constraint and was easily corrected.

7.4.2 The bug collection

Let us now look at the types of errors which, we found, induce overgeneration.

Missing constraints Unsurprisingly, the main source of overgeneration was the lack of sufficient constraints to block illicit tree combinations. For instance, the grammar overgenerated the string "devoir c'est Jean qui part" (instead of "c'est Jean qui doit partir") because the tree for "devoir" was not sufficiently constrained to block adjunction on the VP node of cleft trees. In such cases, adding the relevant constraints (e.g., CEST $= -$ on the foot node of the "devoir"-tree and CEST $= +$ on the VP node of the cleft-tree for *partir*) eliminates the overgeneration.

Incomplete constraints and incorrect feature percolation In some cases, we found that the constraint was only partially encoded by the grammar in that it was correctly stated in one of the combining trees but incorrectly or not at all in the other. Thus for instance, the adjective tree was correctly constrained to adjoin to DET $= -$ N-trees but the corresponding DET $= +$ constraint on the root node of determiner trees was missing. In other cases, the feature was present but incorrectly percolated. In both cases, the partial implementation of the constraint lead to a lack of unification clash and thereby to an overgenerating combination of trees.

Illicit elementary trees A third type of errors was linked to the fact that the grammar was produced semi-automatically from an abstract grammar description. In some cases, the linguist had failed to correctly foresee the implications of her description so that an incorrect elementary tree was produced by the compiler. For instance, we had to introduce an additional constraint in the metagrammar to rule out the formation of trees describing a transitive verb with impersonal subject (in French, transitive verbs cannot take an impersonal subject).

Incorrect semantics A more complex type of error to deal with corner cases where the semantics is insufficiently constrained, thereby allowing for illicit combinations. For instance, in the imperative form, the grammar failed to constrain the first semantic argument to be YOU i.e., the hearer denotation. As a result, the input for sentences such as "Jean demande si Paul part" incorrectly generated strings such as "demande à Jean si Paul part". In such cases thus, it is the semantics associated by the metagrammar with the elementary tree that needs to be modified.

Lexical exceptions As is well known, grammatical generalisations often are subject to lexical exceptions. For instance, transitive verbs are generally assumed to passivise but verbs of measure such as "to weigh" are transitive and do not. In GenI, as is usual in TAG, such exceptions are stated in the lexicon, thereby blocking the selection of certain trees (in this case, all the passive trees) for the lexical items creating the exception (here the measure type verbs). Relatedly, some of the overgeneration cases stem from insufficient lexical information.

7.4.3 Small changes and great effects

Debugging grammars for overgeneration need not be slow and tedious. We have found that with a certain dose of automation — a test harness to mechanise the regression-testing parts of the process, and computer-generated summaries to identify trouble spots — we can obtain major reductions in overgeneration with little effort. It is worth stressing that this reduction, 70% in 12 man-hours and 31 lines of code, is no accident. It comes from our use of a metagrammar system, from GenI's ability to output all the strings for a semantic input and from our incremental testing discipline.

The metagrammar provides a very compact description of the grammar. In particular, shared tree fragments are factored out and used in the production of several trees. As a result, one change to the metagrammar usually induces a change in not one but several (sometimes hundreds of) TAG trees. For instance, a modification stated in the fragment describing the verb spine of the active verb form will affect all trees in the grammar that realise an active verb form i.e., several hundreds of trees. This is not just a matter of factorisation either. The real advantage of a metagrammar architecture is that it allows us to generalise. Suppose, for instance, that a given SR-ITEM indicates that the grammar incorrectly allows the adjunction of a given type of auxiliary tree β to a subject cleft tree. It might be the case that in fact, the grammar should be modified to block the combination of β with *all* cleft trees (not just the subject ones). Then the metagrammar architecture makes it possible to state the required modification at the level of the cleft description so that, in effect, all cleft trees will be modified. In this way, the identification of an overgeneration cause linked to a specific example can be generalised to a larger class of examples.

Making changes quickly is one thing; identifying *what* are the changes to be made is another. This is where GenI and the test methodology come into play. Basically, the suspects report allows for a quick identification of the overgeneration sources. But ultimately, it is because we produce all strings for the input that we know what the wrong ones are. It is the completeness of the output that makes the suspects report really meaningful. Another factor that was beneficial to us was that the input data was organised in a graduated test suite where simple sentences were considered first, then sentences of complexity 2 (cases whose canonical verbalisation involve two finite verbs), then sentences of complexity 3 (three finite verbs). By proceeding incrementally through the test suite, we ensured that early modifications propagate to more complex cases.

7.5 Possible extensions

Overgeneration bugs have not by far been eradicated from our grammar. Encouraging as our initial results may be, it would be worthwhile to look into more efficient approaches to tracking down such errors.

7.5.1 Finding bugs in pairs

One shortcoming of our current approach is that we focus mostly on isolated sources of overgeneration: a single elementary tree, tree property or derivation

operation that consistently occurs in overgenerated strings. However, grammar flaws also consist of unexpected interactions between more than one item (here, "item" could refer to anything, an elementary tree, a derivation operation; it is not particularly relevant to this discussion).

What might therefore be useful is to look for *pairs* of sources that consistently overgenerate. This is subtly different from what the suspects report currently offers. It is true that if two items consistently overgenerate, the suspects report would flag them both. But we do not detect cases where the items do not consistently overgenerate, *except* in the presence of another item.

Suppose we have three items X, Y and Z. X consistently overgenerates in the presence of Y, but not in the presence of Z. The current generation of suspects report would simply report that Y consistently overgenerates, but it would not mention X at all. An improved suspects report might give us more information, telling us that it is actually the pair (X,Y) that is at fault.

This would not necessarily be a subset of the current suspects report either. Again, suppose we have three items A, B and C. A and B consistently overgenerate, but only when they are together. Left to their own devices, A produces some good sentences, as does B. In the current suspects report, we would report neither A nor B, but in the improved version, we would detect that the pair (A,B) should indeed be flagged.

As usual, this sort of flagging would return some false positives. A and B may consistently overgenerate in each others presence, but it does not necessarily mean that it is because they interact.

7.5.2 Automatic pass/overgeneration annotations

Another shortcoming of our approach is that it requires us to be disciplined in our pass/overgeneration annotations. If we mismark a sentence as pass, the derivation summariser will neglect every tree property or derivation item that occurs in that sentence, as it is only looking for items that consistently overgenerate. Perhaps a more robust approach would instead return items that *tend* to occur with overgeneration. This would make it more tolerant to imperfect annotations and as we saw in Section 7.6.3, more amenable to large scale error mining.

Producing these annotations is time-consuming. It would be worthwhile to explore some automatic means of making pass/overgeneration judgements on a large number of sentences, for example, using an n-gram based language model, like one that would be employed by a speech recogniser. We could then take the best N% of the sentences as passes or establish a threshold of improbability, below which sentences will be considered as overgeneration. We could also use more sophisticated tools, e.g., a statistical parser or a symbolic one with a wide coverage grammar in an alternate formalism. Even a relatively liberal parser which itself overgenerates might be useful in that (i) it may overgenerate in different areas than our grammar (ii) anything that *it* marks as a failure would be highly suspicious indeed.

The annotations do not need to be produced by a full-fledged parser either. Indeed, for each sentence that it produces, the surface realiser outputs its parse tree. So another way to classify the generated strings might be through assessing not the quality of the strings themselves but of their parses. For example, we could determine if the elementary trees that were used to build a sentence

are likely to occur together in the same sentence. This kind of information can be extracted from a systemic functional grammar. If we associated each linguistic choice in the SFG network with a set of tree properties from our TAG grammar, we would have an encoding of what tree properties go together. If the sentence contains a set of tree properties for which there is no equivalent system network traversal, it would be flagged as suspicious.

7.6 Related work

Our test cycle is essentially a combination of four ideas:

1. debugging with a generator,
2. rapid development with a metagrammar,
3. localising mistakes with error mining and
4. systematic testing with an organised test suite.

These ideas all have roots in previous work. We can trace the discussion of generation and metagrammars back to work by [Karttunen and Kay, 1985] and [Briscoe *et al.*, 1987] respectively. Error mining may be relatively recent, [van Noord, 2004] and is presumably inspired by the rise of statistical methods in natural language processing. What makes our approach distinct is that we combine the three elements — error mining, generation and metagrammars — and furthermore integrate them through our use of XMG tree properties. Now let us turn to each of these approaches individually.

7.6.1 Finnish Generation

[Karttunen and Kay, 1985] primarily discuss the linguistic issues behind building a (Functional Unification) grammar for Finnish, as well as the computational aspects of parsing such free word order languages. Karttunen and Kay make an offhand remark which is particularly relevant to our work here: they test the correctness of the grammar by taking an incomplete functional description and "produc[ing] from it all the realizations that the grammar allows," which effectively sums up what we have been working on in this chapter.

While an incomplete FD is much like an input semantics (see Section 6.5, Page 141), it is also not entirely the same thing, because it can underspecify for anything in the grammar (unification permitting). For instance, one of Karttunen and Kay's example FDs can realise both "The small child sleeps" and "The small child doesn't sleep", among other things. Underspecification can be useful, but for debugging overgeneration, it would be useful to know that strings being produced really are meant to have the same propositional meaning, in other words that the problem is more likely the grammar being too loose than the input itself.

Finally, Karttunen and Kay do not seem to exploit any links from the generator outputs back to the grammar. Such links may well exist. They perform generation in two stages, first passing through a syntactic generator and then a morphological generator (essential for Finnish). The output of the first generator is a fully specified FD, so perhaps it would be possible to work

backwards from this FD. For example, one could highlight the grammar rules (or disjunction branches) that correspond to a feature value. Doing so might result in something akin to the derivation log and may be the basis of something like a suspects report.

7.6.2 GDE: Metagrammar and generation

[Briscoe *et al.*, 1987] propose a Grammar Development Environment (GDE) which compiles a "source" metagrammar into an "object" grammar, particularly in the Generalised Phrase-Structure Grammar formalism. Their metagrammar has some remarkable similarities to ours; they both allow for an abstract representation of grammar rules, separating feature structures, linear precedence, and immediate dominance. Another striking similarity between our approaches lies in the integration of the metagrammar with the grammar debugging process. Namely, Briscoe *et al.* also use the names of their metagrammar rules to track errors in parsing, much the same way that we trace our way back from generation errors to metagrammar errors, through the tree properties.

One interesting aspect of the GDE approach is that they also include a natural language generator in their grammar debugging process [Boguraev *et al.*, 1988]. GDE emphasises interactive debugging, whereas we focus more on discovery of overgeneration and its causes. Indeed, the GDE generator has more sophisticated debugging tools than does GenI. For one thing, GDE allows the user to disable a set of grammar rules for generation, for example suppressing the rules for coordination, so that the more interesting rules can be tested with little interference. It also has an interactive mode, which provides fine-grained control over the generator. Specifically, the user can select grammar rules from a menu to expand the current node in the generation algorithm, building the syntax tree interactively. Borrowing these features for GenI should be straightforward and worthwhile. GenI also supports interactive debugging, but it only allows the user to step through the surface realisation process, allowing little more than to go back and forth in time. Adding a more hands-on debugger should just be a matter of implementation. We should also be able to get much the same effect as selectively disabling grammar rules, if we took advantage of the paraphrase selection mechanism from Chapter 6. This allows us to restrict which elementary trees are used for generation.

Overall, we have gotten by with a relatively impoverished debugger because our error mining techniques greatly reduces the guesswork out of grammar development. For example, having an incremental approach and a graduated test suite means that we largely avoid the need for isolating grammar rules from one another. The elementary trees we use are debugged in the context of simple inputs, and will tend to be well behaved in the more complex examples. Likewise, our suspects report tends to point us directly to faulty tree properties of our the grammar, and this reduces our need for interactive debugging of the grammar. In a sense, our approach takes [Boguraev *et al.*, 1988] a step further. They use the generator to help identify flaws in the grammar. We add error mining techniques for doing this systematically and on a larger scale.

7.6.3 Alpino: Error mining

One can think of our work in this chapter as scaling up the detection of errors in a grammar. But if we look at similar work in parsing, it is clear that we have done so far is only the very beginning. We take error mining to a larger scale; [van Noord, 2004] takes it to an industrial one. He detects undergeneration in a Dutch grammar with the help of the ALPINO parser, just as we would use a realiser for overgeneration. His techniques essentially consist (i) parsing a large number of sentences — compare his 3 000 000 sentences to our 140 test cases — and (ii) ranking all word sequences in the corpus by parsability. Parsability is a simple metric; it counts, among the sentences that contain a word sequence $w_i...w_j$, the ratio of those sentences that have a successful parse:

$$R(w_i...w_j) = \frac{C(w_i...w_j|\text{OK})}{C(w_i...w_j)}$$

van Noord appears to literally consider all substrings in the corpus, although he retains only those which occur frequently enough (e.g. 5 times) and whose parsability is lower than that of all its sub-substrings (e.g. a length 4 substring is only worth looking at if it has a lower parsability than all the length 2 and length 3 substrings it contains). He uses parsability to great effect, detecting with it tokenisation errors, mistakes in the lexicon, archaic expressions or idioms unknown to the parser, and incomplete grammatical descriptions. (He also catches spelling errors in the corpus, typos, foreign expressions, etc.)

These two approaches to error mining, van Noord's and ours, seem rather different on the surface. He uses parsing, we use surface realisation; he studies undergeneration, we do overgeneration; he looks for frequent failures, we look for consistent failures; he parses a massive batch of sentences, we go forth incrementally. But the underlying philosophy is the same. We both try to make the detection of grammar flaws more efficient by automatically discovering fragments (n-grams in his case and tree properties in ours) that go wrong. On the other hand, it is also worth paying attention to the difference between n-grams and tree properties. Tree properties present a direct link to the (meta)grammar; if there is a problem with a specific tree property, e.g. $ExtractedSubject$, we go directly to the corresponding $ExtractedSubject$ class of the metagrammar. In other words n-grams tell us the symptoms of the disease, whereas tree properties tell us its cause. This suggests a possible avenue to explore. It may perhaps be useful as a post-processing step, to extract a list of something akin to tree properties that tend to be associated with low parsability word sequences.

So what can we learn from [van Noord, 2004]? Scale matters. It may however be more difficult to perform a large scale test of overgeneration as we do not have an automatic means of determining if a sentence is overgeneration or not. Parsability works well for undergeneration, because one only needs to say if the parser returned a result or not. The question then is what the equivalent for parsability would be for generation. A possible answer is to use exactly the same metric as one would use for parsing, that is to apply van Noord's metric to the sentences *produced* by the generator. Clearly, we cannot use the grammar itself to judge the parsability of sentences; all sentences produced would have a parsability of 1. But perhaps we could substitute something else for a grammar, maybe a team of annotators. The idea would be to treat a sentence as parseable if it is associated with a PASS judgement. The idea is that we can do the same

exact thing as van Noord: hunt for n-grams with a low "parsability". It is not clear how useful this would be for generation. Parsability could certainly help for finding things like tokenisation errors and missing lexical entries, whereas our overgeneration errors tend to play on syntax and misplaced clauses. It would be interesting to see if such errors can be detected over small n-grams, like the trigrams (or shorter) employed by van Noord, because that would make the parsability metric genuinely useful. If somebody were to solve the automatic annotations problem, this would be something worth trying out.

7.6.4 TSNLP: Test suite construction

No discussion on error mining would be complete without at least a brief look at the test data being used. [van Noord, 2004] uses a collection of articles from four Dutch newspapers. This represents quite a large amount of text, slightly under three million sentences and is particularly interesting because it is the kind of text that one finds in the real world.[3] The downside is that there is not any particular structure or design behind the text. Sometimes, it may also useful to evaluate parsers, generators and grammars under the controlled settings of a systematically designed, structured test suite. A test suite can be an entirely different creature from a corpus because whatever items are in the suite go in by conscious decision. This means that the suite can offer (i) control over test data, with linguistic phenomena being illustrated alone or in "controlled combination" (ii) systematic coverage of phenomena (iii) negative examples, explicitly ungrammatical examples. The TSNLP is just such a test suite [Balkan *et al.*, 1994]. It covers three languages, English, German and French, and provides 4500 items for each. It is important to keep in mind, especially when comparing this to the 3 million sentences we saw above, that this small number contains only sentences that are distinct from each other in some minimal but linguistically relevant manner, that isolate linguistic phenomena that they can be properly dissected, that cover a broad range of linguistic phenomena, sentences that are "interesting" to study. Another important characteristic of the TSNLP is that is highly structured; sentences are divided into a set of core phenomena (for example complementation, tense-aspect-modality, coordination) and further sub-classified into syntactic domains, specifically: sentences (S), clauses (C), noun phrases (NP), adjectival phrases (AP), prepositional phrases (PP) and adverbial phrases (AdvP).

The test suite we used is largely inspired from the TSNLP. It is far smaller (140 test cases to the TSNLP 4500) and focuses on the phenomena that our grammar, SemFRaG, covers, namely syntactic variation on verbs (this would correspond to the clause and sentence classifications of TSNLP). The TSNLP was scrupulously designed to avoid redundancy; however, from the standpoint of a surface realiser, many of its test cases were essentially identical. For example, the TSNLP C_Agreement and C_Complementation suites contained items which differ only by inflected form. For our current needs, "Elle les accepte (She accepts them)" and "Nous vous acceptons (We accept you)" are virtually the same sentences. The test suite was also inadequate in that it did not provide for the sort of controlled complexity we wanted, namely sentences with more

[3] It is also relatively tame and well-behaved text, compared to say Usenet postings or weblog entries, but at the current stage of SemFRaG's development, newspaper articles well ambitious enough.

than one finite verb such as “Jean dit que l’homme part (Jean says that the man leaves)”.

Chapter 8

Conclusion

8.1 Summary

The driving vision behind this thesis was to build a surface realiser which is reusable, reversible, efficient and contextually aware. Reusability comes from using a domain independent linguistic resource and input language. Reversibility comes from the type of grammar used, in our case, an L_U augmented FB-LTAG which has already proved its usefulness in parsing. Efficiency and contextual awareness are where our contributions come into play. Each of these contributions takes the form an extension to the surface realiser GenI. While the extensions are mutually independent and otherwise very different from each other, they are tied to a common theme of ambiguity and determinism.

Polarity filtering The first of these extensions deals with the problem of lexical ambiguity, having more than one lexical item correspond to each literal of the input semantics. This ambiguity is definitely a feature because it allows the grammar to do the same things in different ways. But to support this feature, we must ensure that the surface realiser is not bogged down by interactions between sets of lexical items that ultimately are not made for each other. The solution is to polarise the grammar so the lexical items know about the resources they provide and the requirements they hold. We then add a filtering step to eliminate all sets of items with non-neutral polarities, that is, with resources and requirements mismatched. Adding this filter makes it much more practical to use an ambiguous grammar for realisation.

Paraphrase selection Having a realiser that could actually *use* lexical ambiguity, we set out to generate all the paraphrases of a given input. While the paraphrases are all grammatically correct (in theory), not every one fits into every context. Our next objective was then to make the realiser a little bit more like the generation-oriented ones (e.g. KPML and SURGE) by increasing its contextual awareness. Ideally, we would be able to generate from the same kind of functional features that such realisers accept. This is not yet possible with GenI, but we have a step in that direction, by adding the ability to preselect our outputs from a set of linguistic criteria, called tree properties. Tree properties are used to restrict our lexical selection. They appear as annotations in the input semantics and in the grammar. Adding tree properties to a

grammar is a straightforward consequence of compiling that grammar from a more abstract metagrammatical formalism. Adding these annotations to the input semantics is a question for future research.

Reducing overgeneration Assuming that all output be grammatical is optimistic, to say the least. Real grammars overgenerate and real grammars built from highly abstract representations overgenerate a lot. This can be problematic for two reasons, (i) that the undeserved ambiguity makes realisation less efficient and (ii) that the issue of choosing a contextually appropriate output is confused with that of choosing a grammatical correct one. However, it turns out that the thing which makes our grammars prone to overgeneration (compilation from an abstract representation to unexpected syntactic structures), can also be used to reduce overgeneration. The idea is to reuse the tree properties we had employed for paraphrase selection. We generate all the strings that are associated with an input semantics, isolate the cases of overgeneration, print out their tree properties and work our way back to the metagrammar.

8.2 Future work

In the previous three chapters, we suggested some possible improvements to the techniques of filtering, selection and grammar debugging. In addition to these individual enhancements, here are some future directions to explore using the surface realiser as a whole.

8.2.1 Morphological generation

The first improvement we should probably make is to add morphological generation to the process. The lack of inflected forms is a hindrance to using the realiser for applications, especially for a language like French with more prominent use of morphology than English. It also complicates experiments we would like to play with the grammar, for example, having a parse-generate loop where we would parse a sentence, do surface realisation from the resulting semantics and then reparse the output. GenI does perform some rudimentary morphological generation based on unification with a morphological lexicon, but (i) the grammar contains too much ambiguity in its leaf nodes for this to work well (ii) the approach does not capture regularities in the language and (iii) orthographic conventions like the transformation of "que il aime" into "qu'il aime" are not accounted for. It would be worthwhile to spend some time reducing the (morphological) ambiguity in the grammar, perhaps using the debugging techniques we saw in this thesis, and to integrate GenI with a proper morphological generator, say Functional Morphology [Forsberg and Ranta, 2004].

8.2.2 Sentence planning

As we look towards the back end of the "pipeline" (morphological generation), we should also consider looking towards the front end (sentence planning). It would be worth considering what relationship GenI has with the sentence planner. For example, is it reasonable to accept bags of literals with tree property annotations, or would it be more appropriate to read in a more abstract

specification? Should the realiser embrace feature-creep and integrate sentence planning tasks à la SPUD, or would we be better off with a strict separation of components? For that matter are there off the shelf microplanners that we could use with the realiser?

8.2.3 Scalability

Continuing along the practical front, there is still work to be done in making the realisation process as efficient as possible. It is possible that paraphrase selection might make the issue of efficient generation moot, but there is always the issue of intersective modifiers to worry about (especially if we start to produce very long outputs). Besides, we saw in Chapter 7, there are applications for which we will not be able to use the preselection method, applications for which we really want the realiser to output everything it can. Two areas worth exploring are improvements to the chart generator and the polarity filter.

The chart generator could stand to have a proper implementation of subtree sharing.[1] Perhaps a useful form of packing would be to deal with multiple/embedded adjunctions by, instead of directly adjoining onto a node, keeping track of pending *sets* of adjunctions on that node. This would contain the combinatorial explosion from intersective modifiers, at least until we moved on to the unpacking phase.

8.2.4 Becoming more generation-oriented

The introduction of tree properties may help us to bridge the gap between reversible realisers like GenI and generation-oriented ones, like KPML and FUF.

The next question to ask is where tree properties are supposed to come from. One possibility would be take a cue from [McDonald and Pustejovsky, 1985] and [Yang, 1992] by using a systemic grammar. The idea might to be some higher level realisation module that takes something more abstract representation and to traverses a system network to collect the appropriate tree properties.

Generation-oriented resources (i.e. SFG grammars) map functional features to linguistic structures. Reversible resources (e.g. TAG grammars) tend not to contain this information, because the question has never come up in parsing. On the other hand, generation-oriented resources also contain "hidden" information about syntax and constituency which is duplicated by reversible resources. The failure of reversible realisers to produce contextually appropriate output comes from the fact that there is not enough functional linguistics encoded in their grammars. The idea that we might traverse a system network to collect tree properties is really a way of saying that we should jettison the redundant syntactic information from the generation grammars, and use both purified resources in conjunction with each other.

8.3 Putting GenI to work

As the surface realiser matures, it may become worthwhile to steer its development by actually trying to use it for actual applications. There are three

[1] As mentioned in the footnote on Page 78, we ought to work out if we actually have sharing for free or not, due to the Haskell implementation.

basic ways we could put the realiser to work. We could have it participate in comparative evaluations with other realisers, we could apply it to what we call "NLP metatasks", or we could fit it into a text generation or a dialogue system.

8.3.1 NLP metatasks

Metatasks are NLG applications which mainly serve to advance NLP research. We saw an example of an NLP metatask in Chapter 7, where we used the surface realiser as a grammar debugging tool. It would be useful to see what other applications there might be for the realiser, especially for its paraphrasing capabilities. For example, one of the problems that comes up in evaluating NLG systems is to have large number of reference texts [Stent *et al.*, 2005; Belz and Reiter, 2006]. Perhaps an exhaustively paraphrasing realiser can be put to use to bootstrap such a collection of texts. We start from an input sentence, parse it, realise all of its variants and use manual verification to get high quality texts out of the pool of results.

8.3.2 NLG tasks

GenI is going to be put to use as a practical generation component. Alexandre Denis is using it as part of a French dialogue system to provide feedback about the system's understanding of the users utterances. Also, Luciana Benotti is using it as part of a multi-lingual French/English dialogue system for research into presupposition accommodation [Beaver and Zeevat, 2007]. Generation is not the main topic of these research projects, so we hope that the realiser will "just work" for them.

8.3.3 Comparative evaluation

It would be useful to see the surface realiser being put through some kind of evaluation process, maybe in comparison with other surface realisers. One possible handicap is that we only have an FB-LTAG grammar for French, whereas most generation work seems to be done with English in mind. There is always XTAG, which provides an English grammar, but it does not have a suitable semantic dimension. Aside from these technical details, there remain the question of what to evaluate for and how to evaluate it.

For example, [Callaway, 2003] tests the linguistic coverage of FUF/SURGE, comparing it with that of HALogen. Starting from the Penn Treebank (newspaper texts with human-verified parse trees), he converts the syntactic trees into functional descriptions (FUF/SURGE inputs), generates strings and compares the resulting strings with their original Penn Treebank entries. Callaway uses the standard train and test methodology, spending several months running iterations over Sections 0-22 and 24 of the Penn Treebank and improving the SURGE grammar rules (and the tree-to-fd translator) by hand.[2] He then unleashes the grammar on the test corpus, Section 23 of the Penn Treebank and compares the results using a number of automatic evaluation methods (for example, the NIST Simple String Accuracy measure). Here, the ingredients for the comparison are (i) a set of reference texts paired with input representations

[2]Perhaps a test harness with automatic pinpointing of errors like in Chapter 7 could reduce that training time.

and (ii) automatic scoring. The key insight behind [Callaway, 2003] seems to be that improving a grammar by hand (with the corpus as a guide) is akin to the training process that statistical realisers undergo. In other words, there is no real reason for symbolic surface realisers not to participate in comparative evaluations against statistical ones. Then again, as Callaway cautions, this is also partly an evaluation of the process which "converts the Penn Treebank notation into the specifications [the realiser] expects."

If a comparative evaluation is going to be made to work, we should probably narrow the window down to tools that have at least comparable objectives. RealPro and kpml are probably out because they expect really different levels of abstraction in the inputs. Maybe LinGo and OpenCCG are more likely candidates, both being realisers that accept a flat semantic language. The only questions that remain would be what exactly we are trying to compare and why.

Appendix A

SemFraG families

family name	# trees
AvoirAux	1
CliticT	1
Copule	1
CopuleVide	1
DetAdj	1
EpithAnte	1
EpithPost	1
EtreAux	1
InvertedSubjClitic	1
Nden1	1
SemiAux	1
SemiAuxDe	1
TempNounSAnte	1
TempNounSPost	1
TempNounVPost	1
advAdjAnte	1
advLoc	1
advSAnte	1
advSPost	1
advVPost	1
complementiser	1
complexAdvDeDeterminer	1
detNegQuantifier	1
detQuantifier	1
dummyAdjective	1
estceque	1
expletive	1
negLeft	1
negativeQuantifier	1
noun	1
pronoun	1
propername	1
s0Cs1Fronted	1
sententialAdv	1
stddeterminer	1
whdeterminer	1
advAdvAnte	2
ilV	2
ilVcs1	2
negPas	2
s0Pcs1	2
s0Ps1	2
s0Pn1	3
complexNDeDeterminer	4

family name	# trees
ilVn1	9
n0ClV	11
n0seV	11
n0vNCopula	11
Coordination	12
n0vN	12
n0Vnbar1	13
n0vA	14
s0vA	16
n0V	20
n0vpN	22
n0ClVs1int	26
n0Vas1	26
n0Vcs1	26
n0Vs1int	26
n0vAdes1	27
n0vNas1	28
n0vNdes1	28
s0Vcs1	30
n0ClVpn1	39
n0Vpn1	39
n0vApn1	42
n0ClVden1	44
n0Vden1	44
n0seVden1	44
n0vNden1	45
n0vAden1	47
n0ClVn1	48
n0Vloc1	48
n0seVn1	48
n0Vdes1	56
s0Vn1	57
n0Vcs1den2	62
n0vAdes1pn2	69
n0vPred	69
s0Van1	69
n0vNan1	78
n0vAan1	80
n0Van1	88
n0Vn1	172
n0Vcs1an2	220
n0Vs1intan2	220
n0Vn1Adj2	240
n0Van1decs2	350
n0Van1decs2ControleObjet	350
n0Vn1decs2ControleObjet	350
n0Vn1pn2	470
n0Vn1cs2	480
n0Vn1cs2ControleObjet	480
n0Vn1den2	491
n0Vn1an2	1052

Appendix B

Tree properties from SemFRaG

EpithAnte
EpithPost
dummyAdjective
n0vA
s0vA
n0vAden1
n0vAan1
n0vApn1
n0vAdes1
n0vAan1pn2
n0vAan1den2
n0vAdes1pn2
advVPost
TempNounVPost
advSPost
TempNounSPost
advSAnte
TempNounSAnte
advAdjAnte
advAdvAnte
advNAnte
GenericPPModifierPost
GenericPPModifierAnte
ppSModifierPost
ppSModifierAnte
ppVModifier
ppNModifier
s0Pn1post
s0Pn1ante
s0Pv1post
s0PLoc1
n0Pn1
s0Pdes1
s0Pques1
s0Ps1
s0Pcs1
s0Pn1
advLoc
prepLoc

abstractCleftPPMod
cleftPPModOne
cleftPPModTwo
ExtractedPPMod
whPPMod
RelPPMod
TenseAux
AvoirAux
EtreAux
Copule
CopuleVide
SemiAux
SemiAuxNPde
SemiAuxDe
complexAdvDeDeterminer
complexNDeDeterminersg
complexNDeDeterminerpl
complexNDeDeterminer
pureDeterminer
DetAdj
stddeterminer
whdeterminer
detQuantifier
detNegQuantifier
n0Nmod
PrepositionalPhrase
s0Cs1middle
s0Cs1Fronted
s0Cs1
negLeft
negPasFinitePost
negPasInfAnte
negPas
ExclamativeQue
GenericCoord
ConstituentCoord
NominalCoord
SententialCoord
PrepCoord
FakePrepCoord
PrepCoordA
PrepCoordDe
PrepCoordAvec
PrepCoordDans
PrepCoordSans
PrepCoordEn
PrepCoordEntre
PrepCoordsHacked
AdjCoord
AdvCoord
Coordination
sententialAdv
nounWithCompl
npWithCompl

n0vN
n0vpN
n0vNden1
n0vNden1des2
n0vNdes1
n0vNas1
Nden1
n0vNan1
n0vNan1den2
propername
pronoun
complementiser
CliticT
expletive
negativeQuantifier
noun
InvertedSubjClitic
estceque
estcequeVP
EmptySubject
InvertedNominalSubject
InvertedIlSubject
NonInvertedNominalSubject
InfinitiveSubject
ImperativeSubject
InterrogInvSubject
CanonicalSubject
CliticSubject
ImpersonalSubject
CanonicalSententialSubjectFinite
CanonicalSententialSubjectInFinitive
CanonicalObject
CanonicalNBar
CanonicalCAgent
CanonicalGenitive
CanonicalIobject
CanonicalOblique
CanonicalLocative
Infinitive
Subject-Control
CanonicalSententialObjectFinite
CanonicalSententialObjectInFinitive
CanonicalSententialObjectInFinitiveDe
CanonicalSententialObjectInFinitiveA
CanonicalSententialObjectInterrogativeFiniteWithoutComplementizer
CanonicalSententialObjectInterrogativeFiniteWithComplementizer
CanonicalSententialObjectInterrogativeInFinitive
CliticObjectII
CliticIobjectII
CliticObject3
CliticIobject3
CliticGenitive
CliticLocative
reflexiveClitic

```
reflexiveAccusative
reflexiveDative
whObject
whLocative
whGenitive
whCAgent
whOblique
RelativeObject
whIobject
RelativeGenitive
RelativeLocative
RelativeIobject
RelativeCAgent
RelativeOblique
CleftObject
CleftDont
CleftIobjectOne
CleftGenitiveOne
CleftCAgentOne
CleftObliqueOne
CleftLocativeOne
CleftLocativeTwo
CleftIobjectTwo
CleftGenitiveTwo
CleftCAgentTwo
CleftObliqueTwo
whSubject
RelativeSubject
CleftSubject
ObjAttributeCan
SententialSubject
SententialCObject
SententialDeObject
SententialAObject
SententialInterrogative
ObjAttribute
AdjectivalPredicativeform
NominalPredicativeform
PrepositionalPredicativeformWithNP
PrepositionalPredicativeformWithAdj
PrepositionalPredicativeform
nOvPred
PredicativeCopula
PredicativeNoun
PredicativeAdjective
PredicativePP
PredicativePrepN
PredicativePrepAdj
activeVerbMorphology
passiveVerbMorphology
middleVerbMorphology
AccReflexiveMorphology
DatReflexiveMorphology
nOvNCopula
```

n0vACopula
ilV
ilVn1
ilVcs1
n0V
n0ClV
n0seV
n0ClVn1
n0seVn1
n0ClVpn1
n0ClVden1
n0seVden1
s0V
n0Vn1
dian0Vn1Active
dian0Vn1Passive
dian0Vn1dePassive
dian0Vn1ShortPassive
dian0Vn1ImpersonalPassive
dian0Vn1middle
dian0Vn1Reflexive
n0Van1
dian0Van1Active
dian0Van1Reflexive
n0Vden1
dian0Vden1
n0Vpn1
n0Vloc1
s0Vn1acs2
n0Van1den2
n0Vden1pn2
n0Vn1pn2
n0Vn1loc2
n0Vn1an2
n0Vn1den2
n0Vcs1
n0Vas1
n0Vs1int
n0ClVs1int
n0Vdes1
n0Vdes1ControleObjet
dian0Vcs1Passive
dian0Vcs1shortPassive
dian0Vdes1ImpersonalPassive
dian0Vdes1Active
n0Vn1sint2
n0Vs1intan2
n0Vn1acs2
n0Vn1cs2ControleObjet
n0Vn1cs2
n0Vcs1decs2
n0Vcs1den2
n0Van1decs2
n0Van1decs2ControleObjet

```
n0Vn1decs2ControleObjet
n0Vdes1pn2
n0Vcs1an2
s0Vn1
s0Van1
s0Vcs1
n0Vnbar1
n0Vn1Adj2
```

Appendix C

Deductive realisation and unification

Beware of bugs below.

C.1 Kay1996 with unification

Given an input semantics **sem** and lexicon L:

Axioms

$$[w; A(a_1, \ldots, a_f)\bullet; lexsem(w)] \quad A \rightarrow w,\ w \in L,\ lexsem(w) \subseteq sem$$

Goals

$$[w_1..w_n; S\bullet; \mathbf{sem}]$$

Inference rules

$$\frac{[w_1..w_i; B(b_1, \ldots, b_g)\bullet; sem_b]}{[w_1..w_i; A((a'_1, \ldots, a'_f)\theta_b) \rightarrow B(b_1, \ldots, b_g) \bullet C((c'_1, \ldots, c'_h)\theta_b); sem_b]} \text{ see below}$$

$$A(a'_1, \ldots, a'_f) \rightarrow B(b'_1, \ldots, b'_g)C(c'_1, \ldots, c'_h),$$
$$\theta_b = mgu((b'_1, \ldots, b'_g), (b_1, \ldots, b_g))$$

(Comp1)

$$\frac{[w_1..w_i; A(a_1, \ldots, a_f) \rightarrow B(b_1, \ldots, b_g) \bullet C(x, c_2, \ldots, c_h); sem_b] \quad [w_i..w_j; C(x, c_2, \ldots, c_h)\bullet; sem_c]}{[w_1..w_j; A((a'_1, \ldots, a'_f)\theta_b\theta_c\theta_a)\bullet; sem_b \cup sem_c]} \text{ see below}$$

$$sem_b \cap sem_c = \emptyset$$
$$A(a'_1, \ldots, a'_f) \rightarrow B(b'_1, \ldots, b'_g)C(c'_1, c'_2, \ldots, c'_h),$$
$$\theta_b = mgu((b'_1, \ldots, b'_g), (b_1, \ldots, b_g))$$
$$\theta_c = mgu((c'_1, \ldots, c'_h)\theta_b, (x, c_2, \ldots, c_h)\theta_b)$$
$$\theta_a = mgu((a'_1, \ldots, a'_f)\theta_b\theta_c, (a_1, \ldots, a_f)\theta_b\theta_c)$$

(Comp2)

C.2 GenI with unification

Substitution phase

Given an input semantics **sem** and lexicon L:

Axioms

$$[\tau; s; rn; (n_1, \ldots, n_x)] \quad \langle \tau, s \rangle \in lexselection(\mathbf{sem}, L),\ rn \text{ is the root node of } \tau,\ (n_1, \ldots, n_x) \text{ are the subst nodes of } \tau$$

Goals

$$[\tau; s; rn; ()]$$

Inference rules

$$\frac{[\tau_p; s_p; rn_p; (n_{p1}, \ldots, n_{px})] \qquad [\tau_c; s_c; rn_c; (n_{c1}, \ldots, n_{cy})]}{[\tau_{pc}; s_p \cup s_c; rn_p\theta; (n_{p2}\theta, \ldots, n_{px}\theta, n_{c1}\theta, \ldots, n_{cy}\theta)]}$$

$\theta = mgu(top(rn_c), top(n_{p1}))$,
$s_b \cap s_c = \emptyset$
$\tau_{pc} = subst(\tau_c, \tau_p, \theta)$

(Sub)

Adjunction phase

Given an input semantics **sem**, and goal edge items GS from the previous phase

Axioms

$$[\tau; s; rn; fn; (n_1, \ldots, n_x)] \quad \langle \tau, s \rangle \in GS,\ (n_1, \ldots, n_x) \text{ are adjoinable nodes of } \tau,\ rn \text{ is the root node of } \tau,\ fn \text{ is the foot node of } \tau \text{ (or } - \text{ if initial)}$$

Goals

$$[\tau; \mathbf{sem}; s; -; (n_1, \ldots, n_x)]$$

Inference rules

$$\frac{[\tau_p; s_p; rn_p; -; (n_{p1}, \ldots, n_{px})] \qquad [\tau_c; s_c; rn_c; fn_c; (n_{c1}, \ldots, n_{cy})]}{[\tau_{pc}; s_p \cup s_c; rn_p\theta_t\theta_b; -; (n_{p2}\theta_t\theta_b, \ldots, n_{px}\theta_t\theta_b, n_{c1}\theta_t\theta_b, \ldots, n_{cy}\theta_t\theta_b)]} \quad \text{see below}$$

$\theta_t = mgu(top(rn_c), top(n_{p1}))$
$\theta_b = mgu(bot(fn_c\theta_t), bot(n_{p1}\theta_t))$
$s_b \cap s_c = \emptyset$
$\tau_{pc} = adj(\tau_c, \tau_p, \theta_t, \theta_b)$

(Adj)

Appendix D

GenI pseudocode

```
1: function Realise(Grammar, InputSem)
2:     ElementaryTrees ← LexSelection(Grammar,InputSem)
3:     DerivedTrees ← CombineTrees(ElementaryTrees,InputSem)
4:     return {sentence : sentence is the leaves of dtree, dtree ∈ DerivedTrees}
5: end function
```

D.1 Lexical selection

```
1: function LexSelection(Grammar, InputSem)
2:     selected ← ∅
3:     for all ⟨Tree, LexSem⟩ ∈ Grammar do
4:         aligned ← AlignSem(InputSem,LexSem)
5:         if aligned ≠ FAIL then
6:             ⟨Tree2, LexSem2⟩ ← unify aligned and LexSem, performing
                   variable substitution on Tree as needed
7:             if ⟨Tree2, LexSem2⟩ ≠ FAIL then
8:                 selected ← selected ∪ {⟨Tree2, LexSem2⟩}
9:             end if
10:        end if
11:    end for
12:    return selected
13: end function

14: function AlignSem(InputSem,LexSem)
15:    ★ is LexSem ⊆ InputSem modulo unification?
16:    i ← 0
17:    aligned ← ∅
18:    for all L ∈ LexSem do
19:        if i ≥ the length of InputSem then
20:            return FAIL
21:        end if
22:        I ← literal i of InputSem
23:        if I has the same predicate and arity as L then
24:            aligned ← aligned ∪ {I}
25:            ★ aligned is a multiset, as are InputSem and LexSem
```

```
        end if
        i ← i + 1
    end for
    return aligned
end function
```

D.2 Realisation proper

```
function CombineTrees(LexSelection,InputSem)
    Agenda ← {t : t ∈ LexSelection, t is an initial tree or has a ↓ node}
    AuxAgenda ← LexSelection \ Agenda
    Chart ← ∅
    output ← ∅

    ★ Substitution phase:
    while Agenda ≠ ∅ do
        aTree ← any tree ∈ Agenda
        Agenda ← Agenda \ {aTree}
        if IsResult(aTree) then
            output ← output ∪ aTree
        else if t has no ↓ nodes and is auxiliary then
            AgendaA ← AgendaA ∪ aTree
        else
            Chart ← ChartInsertion(Chart,aTree)
            for all cTree ∈ ChartRetrieval(Chart,aTree) do
                derived1 ← Substitution(cTree, aTree)
                derived2 ← Substitution(aTree, cTree)
                Agenda ← Agenda ∪ derived1 ∪ derived2
            end for
        end if
    end while

    ★ Chart rotation:
    Agenda ← {t : t ∈ Chart, t has no ↓ nodes}
    Chart ← AuxAgenda
    AuxAgenda ← ∅

    ★ Adjunction phase:
    while Agenda ≠ ∅ do
        aTree ← any tree ∈ Agenda
        Agenda ← Agenda \ {aTree}
        if IsResult(aTree) then
            output ← output ∪ aTree
        else
            for all cTree ∈ ChartRetrieval(Chart,aTree) do
                derived ← Adjunction(cTree, aTree)
                Agenda ← Agenda ∪ derived
            end for
        end if
```

```
    end while
    return output
end function
```

D.3 Helper functions

```
function IsResult(tree,InputSem)
    return t has no ↓ nodes and t.sem = InputSem
end function

function ChartRetrieval(Chart,tree)
    ★ fancier charts require fancier retrieval, see Section 5.3
    return Chart
end function

function ChartInsertion(Chart,tree)
    return Chart ∪ {tree}
end function
```

Bibliography

[Abeillé, 1990] A. Abeillé. Lexical and syntactic rules in a Tree Adjoining Grammar. *Proceedings of the 28th conference on Association for Computational Linguistics*, pages 292–298, 1990.

[Abeillé, 2002] A. Abeillé. *Une grammaire éléctronique du français*. CNRS Editions, 2002.

[Appelt, 1987] D.E. Appelt. Bidirectional grammars and the design of natural language generation systems. *Theoretical Issues in Natural Language Processing*, 3:185–191, 1987.

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

[Balkan *et al.*, 1994] Lorna Balkan, Klaus Netter, Doug Arnold, and Siety Meijer. Tsnlp — test suites for natural language processing. In *Proceedings of Language Engineering Convention*, pages 17–22, Edinburgh, 1994. ELSNET.

[Bangalore and Joshi, 1999] S. Bangalore and A.K. Joshi. Supertagging: an approach to almost parsing. *Computational Linguistics*, 25(2):237–265, 1999.

[Bangalore and Rambow, 2000a] S. Bangalore and O. Rambow. Corpus-based lexical choice in natural language generation. *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 464–471, 2000.

[Bangalore and Rambow, 2000b] S. Bangalore and O. Rambow. Using TAGs, a tree model and a language model for generation. In *Proceedings of TAG+5*, Paris, France, 2000.

[Bateman *et al.*, 1992] J.A. Bateman, M. Emele, and S. Momma. The nondirectional representation of Systemic Functional Grammars and Semantics as Typed Feature Structures. *Proceedings of COLING*, 92:916–920, 1992.

[Beaver and Zeevat, 2007] David Beaver and Henk Zeevat. Accommodation. In Gillian Ramchand and Charles Reiss, editors, *The Oxford Handbook of Linguistic Interfaces*, Studies in Theoretical Linguistics, pages 503–539. Oxford University Press, Oxford, 2007.

[Belz and Reiter, 2006] A. Belz and E. Reiter. Comparing automatic and human evaluation of NLG systems. *Proceedings of EACL*, pages 313–320, 2006.

[Boguraev *et al.*, 1988] B. Boguraev, J. Carroll, T. Briscoe, and C. Grover. Software support for practical grammar development. *Proceedings of the 12th conference on Computational linguistics*, pages 54–58, 1988.

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

[Bonfante *et al.*, 2004] G. Bonfante, B. Guillaume, and G. Perrier. Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation. *Proceedings of CoLing*, pages 303–309, 2004.

[Bos, 1995] J. Bos. Predicate logic unplugged. *Proceedings of the 10th Amsterdam Colloquium*, pages 133–143, 1995.

[Brew, 1992] C. Brew. Letting the cat out of the bag: Generation for shake-and-bake MT. In *Proceedings of COLING '92*, Nantes, France, 1992.

[Briscoe *et al.*, 1987] E. Briscoe, C. Grover, B. Boguraev, and J. Carroll. A formalism and environment for the development of a large grammar of English. *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 703–708, 1987.

[Calder *et al.*, 1989] J. Calder, M. Reape, and H. Zeevat. An algorithm for generation in unification categorial grammar. *Proceedings of the fourth conference on European chapter of the Association for Computational Linguistics*, pages 233–240, 1989.

[Callaway, 2003] C.B. Callaway. Evaluating Coverage for Large Symbolic NLG Grammars. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 811–817, 2003.

[Carroll and Oepen, 2005] J. Carroll and S. Oepen. High efficiency realization for a wide-coverage unification grammar. *Proceedings of the 2nd International Joint Conference on Natural Language Processing. Jeju, Republic of Korea*, 2005.

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

[Copestake *et al.*, 2005] A. Copestake, D. Flickinger, C. Pollard, and I.A. Sag. Minimal Recursion Semantics: An Introduction. *Research on Language & Computation*, 3(4):281–332, 2005.

[Crabbé and Duchier, 2004] B. Crabbé and D. Duchier. Metagrammar redux. In *International Workshop on Constraint Solving and Language Processing - CSLP 2004, Copenhagen*, 2004.

[Danlos, 1984] L. Danlos. Conceptual and linguistic decisions in generation. *Proceedings of the 10th international conference on Computational linguistics*, pages 501–504, 1984.

[Danlos, 1998] L. Danlos. G-TAG : un formalisme lexicalisé pour la génération de textes inspiré de TAG. *Traitement Automatique des Langues - T.A.L.*, 2, 1998.

[Duchier and Debusmann, 2001] D. Duchier and R. Debusmann. Topological dependency trees: a constraint-based account of linear precedence. *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 180–187, 2001.

[Dymetman and Isabelle, 1988] M. Dymetman and P. Isabelle. *Reversible Logic Grammars for Machine Translation*. Canadian Workplace Automation Research Centre, Dept. of Communications Canada, 1988.

[Elhadad and Robin, 1999] M. Elhadad and J. Robin. SURGE: a comprehensive plug-in syntactic realization component for text generation. *Computational Linguistics*, 1999.

[Elhadad, 1991] M. Elhadad. FUF: The universal unifier user manual version 5.0. *Columbia University*, 1991.

[Forsberg and Ranta, 2004] M. Forsberg and A. Ranta. Functional morphology. *ACM SIGPLAN Notices*, 39(9):213–223, 2004.

[Gabbay, 1996] D.M. Gabbay. *Labelled deductive systems. 1*. Oxford University Press, 1996.

[Gardent and Kallmeyer, 2003] C. Gardent and L. Kallmeyer. Semantic construction in FTAG. In *Proceedings of the 10th EACL*, Budapest, Hungary, 2003.

[Gardent and Kow, 2005] C. Gardent and E. Kow. Generating and selecting grammatical paraphrases. *Proceedings of the ENLG*, Aug 2005.

[Gardent and Kow, 2007] C. Gardent and E. Kow. A symbolic approach to near-deterministic surface realisation using Tree Adjoining Grammar. *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 328–335, 2007.

[Gardent, 2006] C. Gardent. Integration d'une dimension semantique dans les grammaires d'arbres adjoints. *TALN*, 2006.

[Gerdemann and Hinrichs, 1995] D. Gerdemann and E. Hinrichs. Some open problems in head-driven-generation. *Linguistics & Computation, S*, pages 165–197, 1995.

[Gerdemann, 1991] Dale Gerdemann. *Parsing and Generation of Unification Grammars*. PhD thesis, University of Illinois, 1991.

[Halliday, 1978] M.A.K. Halliday. *Language as Social Semiotic: The Social Interpretation of Language and Meaning*. Edward Arnold, 1978.

[Halliday, 1985] MAK Halliday. An Introduction to Functional Linguistics. *London: Edward Arnold*, 94, 1985.

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

[Joshi and Schabes, 1997] 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.

[Joshi and Srinivas, 1994] A.K. Joshi and B. Srinivas. Disambiguation of super parts of speech (or supertags): almost parsing. *Proceedings of the 15th conference on Computational linguistics-Volume 1*, pages 154–160, 1994.

[Joshi *et al.*, 1975] A.K. Joshi, L. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

[Joshi, 1987] A.K. Joshi. *The Relevance of Tree Adjoining Grammar to Generation*. University of Pennsylvania, School of Engineering and Applied Science, Dept. of Computer and Information Science, 1987.

[Kallmeyer and Romero, 2004] L. Kallmeyer and M. Romero. LTAG Semantics with Semantic Unification. *Proceedings of TAG*, 7:155–162, 2004.

[Karttunen and Kay, 1985] L. Karttunen and M. Kay. Parsing in a free word order language. *Natural Language Parsing*, pages 279–306, 1985.

[Kasami, 1965] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. *Air Force Cambridge Research Laboratory report AF-CRL-65-758, Bedford MA*, 1965.

[Kay, 1996] M. Kay. Chart Generation. In *34th ACL*, pages 200–204, Santa Cruz, California, 1996.

[Knight and Hatzivassiloglou, 1995] K. Knight and V. Hatzivassiloglou. Two-level, many-paths generation. *Proc. ACL*, 95, 1995.

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

[Kow, 2005] E. Kow. Adapting polarised disambiguation to surface realisation. In *17th European Summer School in Logic, Language and Information - ESSLLI'05, Edinburgh, UK*, Aug 2005.

[Kroch and Joshi, 1985] A.S. Kroch and A.K. Joshi. *The Linguistic Relevance of Tree Adjoining Grammar*. University of Pennsylvania, Moore School of Electrical Engineering, Dept. of Computer and Information Science, 1985.

[Langkilde-Geary, 2002] I. Langkilde-Geary. An empirical verification of coverage and correctness for a general-purpose sentence generator. *Proceedings of the 12th International Natural Language Generation Workshop*, pages 17–24, 2002.

[Langkilde, 2000] I. Langkilde. Forest-based statistical sentence generation. *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 170–177, 2000.

[Lavoie and Rambow, 1997] B. Lavoie and O. Rambow. RealPro–a fast, portable sentence realizer. *ANLP'97*, 1997.

[Le Roux, 2007a] J. Le Roux. Intersection optimisation is NP-complete. Submitted to CIAA 2007, 2007.

[Le Roux, 2007b] J. Le Roux. *La coordination dans les grammaires d'interaction*. PhD thesis, Université Henri Poincaré, 2007.

[Levine, 1990] J.M. Levine. PRAGMA—a flexible bidirectional dialogue system. *AAAI-90 [8]*, pages 964–969, 1990.

[McDonald and Pustejovsky, 1985] David D. McDonald and James D. Pustejovsky. TAGs as a grammatical formalism for generation. In *Proceedings of the 23rd annual meeting on Association for Computational Linguistics*, pages 94–103, Morristown, NJ, USA, 1985. Association for Computational Linguistics.

[Neumann, 1994] G. Neumann. *A Uniform Computational Model for Natural Language Parsing and Generation*. PhD thesis, 1994.

[Parmentier, 2007] Yannick Parmentier. *SemTAG : une plate-forme pour le calcul sémantique à partir de Grammaires d'Arbers Adjoints*. PhD thesis, Université Henri Poincaré, 2007.

[Perrier, 2003] G. Perrier. Les grammaires d'interaction, 2003. Habilitation à diriger les recherches en informatique, université Nancy 2.

[Reiter and Dale, 2000] Reiter and Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[Revuz, 1992] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[Schabes and Shieber, 1994] Y. Schabes and S.M. Shieber. An Alternative Conception of Tree-Adjoining Derivation. *Computational Linguistics*, 20(1):91–124, 1994.

[Schabes *et al.*, 1988] Y. Schabes, A. Abeille, and A.K. Joshi. Parsing strategies with'lexicalized'grammars: application to tree adjoining grammars. *Proceedings of the 12th conference on Computational linguistics-Volume 2*, pages 578–583, 1988.

[Scheffler, 2003] Tatjana Scheffler. Generation with TAG - a semantics interface and syntactic realizer. Diplomarbeit, Computerlinguistik, Universität des Saarlandes, Saarbrücken, Germany, 2003.

[Shemtov, 1996] H. Shemtov. Generation of paraphrases from ambiguous logical forms. In *COLING*, pages 919–924, 1996.

[Shieber *et al.*, 1990] 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.

[Shieber *et al.*, 1995] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995.

[Shieber, 1985] Stuart Shieber. Using restriction to extend parsing algorithms for complex feature based formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics*, 1985.

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

[Shieber, 1993] Stuart M. Shieber. The problem of logical-form equivalence. *Comput. Linguist.*, 19(1):179–190, 1993.

[Stent *et al.*, 2005] A. Stent, M. Marge, and M. Singhai. Evaluating Evaluation Methods for Generation in the Presence of Variation. *Proceedings of CICLING*, pages 341–351, 2005.

[Stone and Doran, 1997] M. Stone and C. Doran. Sentence planning as description using tree adjoining grammar. *Proceedings of the 35th conference on Association for Computational Linguistics*, pages 198–205, 1997.

[Stone *et al.*, 2003] M. Stone, C. Doran, B. Webber, T. Bleam, and M. Palmer. Microplanning with Communicative Intentions: The SPUD System. *Computational Intelligence*, 19(4):311–381, 2003.

[Striegnitz, 2000] K. Striegnitz. Interleaving Sentence Planning With Contextual Reasoning: a system description. Technical report, 2000.

[Striegnitz, 2001] Kristina Striegnitz. *A Chart-based Generation Algorithm for LTAG with Pragmatic Constraints*. PhD thesis, Saarland University, 2001.

[Striegnitz, 2004] Kristina Striegnitz. *Generating Anaphoric Expressions — Contextual Inference in Sentence Planning*. PhD thesis, Saarland University, November 2004.

[Tapanainen, 1997] P. Tapanainen. Applying a finite-state intersection grammar. *Finite-State Language Processing. MIT Press, Cambridge*, page 311327, 1997.

[Tomita, 1987] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.

[van Noord, 1989] Gertjan van Noord. BUG: A directed bottom up generator for unification based formalisms. Technical Report 4, Leuven, 1989.

[van Noord, 2004] G. van Noord. Error mining for wide-coverage grammar engineering. *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, 2004.

[Vijay-Shanker and Joshi, 1988] K. Vijay-Shanker and AK Joshi. Feature Structures Based Tree Adjoining Grammars. 55:v2, 1988.

[Wedekind, 1988] J. Wedekind. Generation as structure driven derivation. *Proceedings of the 12th International Conference on Computational Linguistics*, pages 732–737, 1988.

[White, 2004] M. White. Reining in CCG chart realization. In *INLG*, pages 182–191, 2004.

[Whitelock, 1992] P. Whitelock. Shake-and-bake translation. *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92), Nantes, France*, 1992.

[Winograd, 1983] T. Winograd. *Language as a Cognitive Process.* Addison-Wesley, 1983.

[XTAG Research Group, 2001] XTAG Research Group. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania, 2001.

[Yang, 1992] G. Yang. An integrated approach to generation using systemic grammars and tree adjoining grammars. 1992.

[Younger, 1967] DH Younger. Recognition and parsing of contextfree languages in time n cubed. *Information and Control*, 10(2):189–208, 1967.

Résumé

La réalisation de surface est une partie du processus global de génération de langue naturelle. Elle peut être vue comme l'inverse de l'analyse en ce sens où, étant donné une grammaire et une représentation du sens, le réalisateur de surface produit une chaîne en langue naturelle que la grammaire associe à un sens donné en entrée. Cette thèse présente trois extension de GenI, un algorithme de réalisation pour une grammaire de type FB-LTAG. La première extension augmente l'efficacité du réalisateur pour le traitement de l'ambiguïté lexicale. C'est une adaptation de l'optimisation par « étiquetage électrostatique » qui existe déjà pour l'analyse, qui consiste à associer les items lexicaux à des ensembles de polarités, et à éliminer les combinaisons dont les polarités ne sont pas neutres. La deuxième extension concerne le nombre de sorties retournées par le réalisateur. En temps normal, l'algorithme GenI retourne *toutes* les phrases associées à une même forme logique. Alors qu'on peut considérer que ces entrées ont le même sens, elles présentent souvent de subtiles nuances. Il est important que les systèmes de génération contrôlent ces facteurs supplémentaires. Ici, nous montrons comment la spécification de l'entrée peut être augmentée d'annotations qui permettent un tel contrôle des sorties. L'extension est permise par le fait que la grammaire FB-LTAG utilisée par le générateur a été construite à partir d'une « métagrammaire », mettant explicitement en œuvre les généralisations qu'elle code. La dernière extension donne la possibilité au réalisateur de servir d'environnement de débuggage de la métagrammaire. Les erreurs dans la métagrammaire peuvent avoir des conséquences importantes pour la grammaire. Comme le réalisateur donne en sortie toutes les chaînes associées à une sémantique d'entrée, il peut être utilisé pour trouver ces erreurs et les localiser dans la métagrammaire.

Mots clés Réalisation de surface, grammaires d'arbres adjoints, metagrammaires, génération

Abstract

Surface realisation is a subtask of natural language generation. It may be viewed as the inverse of parsing, that is, given a grammar and a representation of meaning, the surface realiser produces a natural language string that is associated by the grammar to the input meaning. This thesis presents three extensions to GenI, a realisation algorithm for Feature-Based Tree Adjoining Grammar (FB-LTAG). The first extension improves the efficiency of the realiser with respect to lexical ambiguity. It is an adaptation from parsing of the "electrostatic tagging" optimisation, in which lexical items are associated with a set of polarities, and combinations of those items with non-neutral polarities are filtered out. The second extension deals with the number of outputs returned by the realiser. Normally, the GenI algorithm returns *all* of the sentences associated with the input logical form. Whilst these inputs can be seen as having the same core meaning, they often convey subtle distinctions in emphasis or style. It is important for generation systems to be able to control these extra factors. Here, we show how the input specification can be augmented with annotations that provide for the fine-grained control that is required. The extension builds off the fact that the FB-LTAG grammar used by the generator was constructed from a "metagrammar", explicitly putting to use the linguistic generalisations that are encoded within. The final extension provides a means for the realiser to act as a metagrammar-debugging environment. Mistakes in the metagrammar can have widespread consequences for the grammar. Since the realiser can output all strings associated with a semantic input, it can be used to find out what these mistakes are, and crucially, their precise location in the metagrammar.

Keywords Surface realisation, Tree Adjoining Grammar, metagrammars, generation