Generalized Categorial Grammar for Unbounded Dependencies Recovery

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Luan Viet Nguyen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

William Schuler

January, 2014

© Luan Viet Nguyen 2014
ALL RIGHTS RESERVED

Acknowledgements

First I would like to thank my family: my wife, Thuy, for her forever love, companionship and years long support and encouragement throughout the best and hardest times; my two sons Khoa (7yrd) and Khiem (4yrd) for their being good boys and understanding that dad is busy on his research and not having enough time to play with them. I thank my widowed mother Chuyen and my grand mother Lien for their nurturing me and my two sisters Nhan and An when my dad passed away as a result of the Vietnam war when I was only 4 year old. These two women have the most influence in my life. Without their support, affection and love throughout the hardest times, I would not be the person who wrote this thesis.

There are many people that have earned my great deal of gratitude for their contribution to my growth as a researcher in graduate school. First of all, I want to thank my advisor, William Schuler. I could not get this far into writing this thesis without his years long supports and encouragements. He is always there for me through the good times and the bad times on my graduate school journey. Without him, I would have given up after a couple unsuccessful tries to push a paper published, and would never see the bright light when we finally got the Best Paper Award at COLING 2012. He is the one who took the responsibility and went extra miles to correct a registration mistake that I made that could otherwise had a financial impact on me. Even though I never took any of his classes, he is no doubt the best single source of knowledge about NLP that I have learned from through out the years.

The second professor that have earned my great gratitude since the time I was an undergraduate student here is Maria Gini. She was the one who fought to keep me from being expelled from this school when I made a mistake with the admission office back in the late 90's. Her classes that I took were always eye-opening for me to enter the world of AI and robotic. That knowledge helped build the foundation for me to go on the NLP research later. I also want to thank professor Richard McGehee for encouraging me to keep going for a PhD after I got done

with my Master a couple years back, and professor Daniel Boley for his willingness to be on my committee.

Thanks to my lab members: Tim Miller, Stephen Wu, Lane Schwartz, Andy Exley, and Dingcheng Li for all their collaborations and bouncing of ideas and questions. I do miss the time when we were all there in the NLP lab. I also want to thank Marten van Schijndel, a graduate student from Ohio State University for the wonderful collaboration work on the unbounded dependency recovery. This work steered a successful direction and helped put the last piece of the puzzle for this thesis.

I have been working as a professional software developer during my time in graduate school, so there are some people not in the academic field that have earned my gratitude. I want to thank Wayne Ziebarth, the President of WebScurity, for providing some tuition support and encouragement when I first started my graduate school. I also want to thank Saviz Artang, Jason Haertel-Strehlow, and Rajiv Narang, the Manager and Team Leads of the West LegalEdCenter group of Thomson Reuters for accommodating my odd schedules at work so that I can be in school to take or help teach classes and do my research. Without that support, this thesis work is not possible.

Last but not least, I want to thank professors Mats Heimdahl, John Collins, and Mike Whalen for providing me the teaching assistantship and having the trust in me to help them out in the MSSE program in the last couple years.

Dedication

I dedicate this thesis to my family including my extended family for nursing me with all the love and affections, and for their dedicated partnership for success in my life.

Abstract

Accurate recovery of predicate-argument dependencies is vital for interpretation tasks like information extraction and question answering, and unbounded dependencies may account for a significant portion of the dependencies in any given text. This thesis describes a Generalized Categorial Grammar (GCG) which, like other categorial grammars, imposes a small, uniform, and easily learnable set of semantic composition operations based on functor-argument relations, but like HPSG, is generalized to limit the number of categories used to those needed to enforce grammatical constraints.

The thesis also describes a system for automatically reannotating syntactically-annotated corpora for the purpose of refining linguistically-informed phrase structure analyses of various phenomena. In particular, it describes a method for implementing syntactic analyses of various phenomena through automatic reannotation rules, which operate deterministically on a corpus like the Penn Treebank (Marcus et al., 1993) to produce a corpus with desired syntactic analyses. This reannotated corpus is then used to define a probabilistic grammar which is automatically annotated with additional latent variable values (Petrov and Klein, 2007) and used to parse the constituent and syntactic dependencies from input sentences of the Wall Street Journal and from a minimal but special corpus introduced by (Rimell et al., 2009) that contains only sentences having Object extraction from a relative clause, Object extraction from a reduced relative clause, Subject extraction from a relative clause, Free relatives, Object wh-questions, Right node raising, and Subject extraction from an embedded clause. This corpus was designed specifically to test various parsers on their capability to recover these unbounded dependencies as studied by (Rimell et al., 2009, Nivre et al., 2010). Our system achieves the best result with noticeable margin on unbounded dependency recovery task compared to the results of all 7 other major systems studied by (Rimell et al., 2009, Nivre et al., 2010). The first paper describing this system earned the attention from the NLP research community after it won the Best Paper Award at the international conference COLING 2012.

Contents

Acknowledgements **i**

Dedication **iii**

Abstract **iv**

List of Tables **viii**

List of Figures **x**

1 Introduction **1**

2 Background **4**

2.1 CCG and HPSG 4

2.2 Reannotation from Penn Treebank to CCG and HPSG 5

2.3 System Evaluations 5

3 Generalized Categorial Grammar **7**

4 Automatically Reannotating TreeBank **15**

4.1 Reannotation rules for initial and final argument attachment (**-a/-b**) 19

4.1.1 Reannotation rules for initial argument attachment (**-a**) 20

4.1.2 Reannotation rules for final argument attachment (**-b**) 23

4.2 Reannotation rules for initial and final modifier attachment 54

4.2.1 Reannotation rules for initial modifier attachment 55

4.2.2 Reannotation rules for final modifier attachment 65

4.3 Reannotation rules for coordinating conjunctions (**-c/-d**) 76
4.4 Reannotation rules for filler attachment 87
4.4.1 Reannotation rules to apply gapped clauses to modificands (Fa and Fd) 88
4.4.2 Reannotation rules to apply gapped clauses to relative phrases (Fb) . . 88
4.4.3 Reannotation rules to apply gapped clauses to interrogative phrases (Fc and Fe) 88
4.5 Reannotation rules to hypothesize gap as initial argument, final argument, or modifiers 90
4.6 Reannotation rules for relative pronoun attachment (**-r**) 93
4.6.1 Relative pronoun attachment for nominal phrase 94
4.6.2 Relative pronoun attachment for verb phrase 96
4.6.3 Relative pronoun attachment for sentential phrase 96
4.7 Reannotation rules for argument elision (**-a/-b**) 98
4.8 Reannotation rules for right node raising (**-h**) 99
4.9 Reannotation rules for type changing 99
4.10 Preprocessing Penn Treebank trees 102
4.10.1 Normalize PTB inconsistencies 102
4.10.2 Head Percolation 102

5 Syntax Evaluations **113**
5.1 Syntax Evaluation on GCG 114
5.1.1 Syntax Parsing in GCG 114
5.1.2 Evaluating GCG Parse Result 118
5.2 Syntax Evaluation on CCG 118
5.2.1 Syntax Parsing on CCG 118
5.2.2 Evaluating CCG Parse Result 123
5.3 Significance Tests on Syntax Evaluations for GCG vs CCG 123
5.3.1 Student's t-test on Syntax Evaluations for GCG vs CCG 126
5.3.2 McNemar's test on Syntax Evaluations for GCG vs CCG 126
5.4 Relaxed Syntax Evaluations 130
5.4.1 "Onlyval" Syntax Evaluations 130
5.4.2 "Unlabeled" Syntax Evaluations 131

6 Dependency Evaluations **137**
6.1 Dependency Evaluation on GCG 139
6.2 Dependency Evaluation on CCG 140
6.3 Significance Tests on Dependency Evaluations for GCG vs CCG 141
6.3.1 Student's t-test on Dependency Evaluations for GCG vs CCG 142
6.3.2 McNemar's test on Dependency Evaluations for GCG vs CCG 142

7 Unbounded Dependency Evaluations **146**

8 Conclusion and Discussion **155**

References **157**

Appendix A. Additional Reannotation Rules **161**
A.1 Other reannotation rules for initial and final argument attachment 161
A.2 Other reannotation rules for initial and final modifier attachment 163
A.3 Other reannotation rules for filler attachment 166
A.4 Other reannotation rules for type changing 173
A.5 Miscellaneous Rules 176

List of Tables

5.1 Syntax evaluation for GCG on Berkley Parser. 120
5.2 Syntax evaluation for CCG on Berkley Parser. 125
5.3 Student's t-test for Syntax Evaluation between GCG and CCG on the Berkley parser using the R stats software. 127
5.4 McNemar test for Syntax Evaluation between GCG and CCG on the Berkley parser 128
5.5 "onlyval" syntax evaluations result for GCG on the left and CCG on the right. These relaxing results also consistently show that GCG is a better parsing tool than CCG on WSJ section 23. 132
5.6 significance test results for "onlyval" evaluations: Student's t-test (above) and McNemar test (below). Both tests confirm that GCG is significantly more accurate than CCG on this relaxed syntax parsing task. 133
5.7 "unlabeled" syntax evaluations result for GCG on the left and CCG on the right. These relaxing results also consistently show that GCG is a better parsing tool than CCG on WSJ section 23. 135
5.8 significance test results for "unlabeled" evaluations: Student's t-test (above) and McNemar test (below). Both tests confirm that GCG is significantly more accurate than CCG on this relaxed syntax parsing task in general. 136
6.1 Dependency evaluation for GCG. 140
6.2 Dependency evaluation for CCG. 141
6.3 Student's t-test result for Dependency Evaluation between GCG and CCG . . . 144
6.4 McNemar test result for Dependency Evaluation between GCG and CCG . . . 144

7.1 Unbounded dependency results compared to those of other systems studied by Rimell et al. (2009) and Nivre et al. (2010) over a variety of constructions: object extraction from relative clauses (Obj RC), object extraction from reduced relative clauses (Obj Red), subject extraction from relative clauses (Sbj RC), free relatives (Free), object wh-questions (Obj Q), right node raising (RNR), and subject extraction from embedded clauses (Sbj Embed). Evaluated parsers are C&C (Clark and Curran, 2007), Enju (Miyao and Tsujii, 2005), DCU (Cahill et al., 2004), Rasp (Briscoe et al., 2006), Stanford (Klein and Manning, 2003), MST (McDonald, 2006), Malt (Nivre et al., 2006a,b). This system used the Berkley parser (Petrov and Klein, 2007) run on the reannotated categorial grammar. 154

List of Figures

3.1 Graphical representation of predicate-argument dependencies for the sentence *The person who officials say stole millions.* 9

3.2 Example categorization of the noun phrase *the person who officials say stole millions.* This derivation yields the following lexical relations: $(0\ \mathbf{i_1})$=the, $(0\ \mathbf{i_2})$=person, $(0\ \mathbf{i_3})$=who, $(0\ \mathbf{i_4})$=officials, $(0\ \mathbf{i_5})$=say, $(0\ \mathbf{i_6})$=stole, $(0\ \mathbf{i_7})$=millions, and the following argument relations: $(1\ \mathbf{i_2})$=$\mathbf{i_1}$, $(1\ \mathbf{i_3})$=$\mathbf{i_2}$, $(1\ \mathbf{i_5})$=$\mathbf{i_4}$, $(2\ \mathbf{i_5})$=$\mathbf{i_6}$, $(1\ \mathbf{i_6})$=$\mathbf{i_3}$, $(2\ \mathbf{i_6})$=$\mathbf{i_7}$. The semantic dependency relations for this sentence are represented graphically in Figure 3.1. 14

3.3 Example categorization of the noun phrase *creditors investors and employees of the company.* This derivation yields the following lexical relations: $(0\ \mathbf{i_1})$=creditors, $(0\ \mathbf{i_2})$=investors, $(0\ \mathbf{i'_2})$=$(0\ \mathbf{i_3})$=and, $(0\ \mathbf{i_4})$=employees, $(0\ \mathbf{i_5})$=of, $(0\ \mathbf{i_6})$=the, $(0\ \mathbf{i_7})$=company, and the following argument relations: $(2\ \mathbf{i_1})$=$\mathbf{i_5}$, $(2\ \mathbf{i_2})$=$\mathbf{i_5}$, $(2\ \mathbf{i_4})$=$\mathbf{i_5}$, $(1\ \mathbf{i'_2})$=$\mathbf{i_1}$, $(2\ \mathbf{i'_2})$=$\mathbf{i_3}$, $(1\ \mathbf{i_3})$=$\mathbf{i_2}$, $(2\ \mathbf{i_3})$=$\mathbf{i_4}$, $(1\ \mathbf{i_5})$=$\mathbf{i_7}$, $(1\ \mathbf{i_7})$=$\mathbf{i_6}$. 14

4.1 Sample sed-like reannotation rule introducing a gap tag at the top of a relative clause (a), and an application of this rule to the movement-based notation in the Penn Treebank (b) to produce a binary-branching categorial grammar derivation using gap arguments (c). Rules are applied to every constituent from the top of the tree down, using parentheses to delimit constituents above the current constituent, carets to delimit the current constituent, angle brackets to delimit child constituents, and square brackets to delimit constituents below children. Delimiters are then updated at every iteration. 17

4.2 Branch off final possessive *'s*. This example has α=D and γ='s. 20

4.3 Branch off initial specifier **N** measure. This example has α=R-aN. 21

4.4 **[VIBLAG]** sentence: branch off initial **N** subject. This example has α=V and β=$\varnothing$. 21
4.5 **[VIBLAG]** sentence: branch off initial **N** subject. This example has α=V and β=$\varnothing$. 22
4.6 Branch off initial determiner (non-wh). This example has α=N and β=$\varnothing$. . . . 22
4.7 Branch off initial determiner (non-wh). If γ=**NP[^]*** then it must have a descendant **(POS 's)** or **(POS ')** in δ. This example has α=N and β=-bV-bO. . . . 23
4.8 Branch off initial determiner (wh). This example has α=N, β=γ=δ=$\varnothing$. 23
4.9 Branch off final **VP** as argument **B-aN**. If ε=**V[^]*** then ζ must be either *do*, *does*, or *did*. If $\theta \neq \varnothing$ then its top-left-most branch must be a **(RB .*)** e.g. **(RB not)**. The three examples show different types of possible categories of the left child. 25
4.10 Branch off final VP as argument **L-aN** (w. special cases because *'s* is ambiguous between *has* and *is*). If ε=**'d** then its parent must be a **VBD**. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be a **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. This example has α=V, β=-aN, γ=$\varnothing$. . . 26
4.11 Branch off final argument **L** (*had I known* construction). 27
4.12 Branch off final VP as argument **L-aN**. This example has α=V, β=-aN, γ=$\varnothing$. . 28
4.13 Branch off final **PRT** as argument **Pword** particle. This example has α=V and β=-aN-bN. 29
4.14 Branch off final verbal or predicative adjectival phrase as argument **A-aN** if it occurs after a copular *be*. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If ε=**'s** then its immediate parent must be a **VBZ**. If θ=**SBAR[^]*-PRD** then its left-most child must not be a **WH[^]*** or **(IN that)**. This example has α=V, β=-aN, γ=$\varnothing$. 30
4.15 Branch off final predicative noun phrase following the copular *be* as argument **A-aN** and then unary transforming to **N**. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If ε=**'s** then its immediate parent must be a **VBZ**. This example has α=V, β=-aN, γ=$\varnothing$. 31

4.16 Branch off final adjectival or adverbial predicative as argument **A-aN**. If ε=**VP** then its first **VP**-head must be a **VB[NG]**. This means there's no child of category **VB|JJ|MD|TO** between the left-most child and the **VB[NG]**-head. This example has α=B, β=-aN and γ=$\varnothing$. 31
4.17 Branch off final argument embedded question **S** with quotations. 32
4.18 Branch off final **SQ|SINV** as argument **S**. 32
4.19 Branch off final argument **O**. This example has α=N and β=-rN. 33
4.20 Branch off final argument **O**. The magenta example is for parent node of noun phrase category **N**. It has α=N and β=$\varnothing$. The blue example is for parent node of verbal categories. It has α=V-aN and β=$\varnothing$. 33
4.21 Branch off final argument **O**. This example has α=N and β=$\varnothing$. 34
4.22 Branch off final argument **G-aN**. This example has α=R-aN and β=$\varnothing$. 35
4.23 Branch off final argument **G**. This example has α=R-aN and β=$\varnothing$. 35
4.24 Branch off final argument **N**. This example has α=V-aN. 35
4.25 Branch off final argument **N**. This example has α=B-aN and β=$\varnothing$. 36
4.26 Branch off final **SBAR** as argument **N** (nom clause): If ε=$\varnothing$ then the left-most leaf on ζ must be **(when|whether)** as depicted in this example. No node in ζ can be of category ending in **-TMP**. This example has α=V-aN and β=$\varnothing$. . . . 36
4.27 Branch off final **SBAR** as argument **I-aN**: 37
4.28 Branch off final **SBAR** as argument **V-iN**. This rule matches when either (1) γ $\neq$**(-ADV|-TMP)** and ζ=**(whether|if)** or (2) δ=**WH[^]*** and neither ζ=**that** nor ε=**-NONE-**. 38
4.29 Branch off final **SBAR** as embedded argument **E**. The top-left node in β must not be a **[^]*-ADV** and its sibling, if β has more than one top-level node, must not be a **([^]* :)**. 39
4.30 Branch off final **SBAR** as complementized argument **C**. 39
4.31 Branch off final **INTJ** as argument **C**. 39
4.32 Branch off final **S** with empty subject as argument **[VIBA]-aN**. 40
4.33 Branch off final **S** with empty subject as argument **[VIBA]-aN**. This example has γ=I. 40
4.34 Branch off final **S** as modifier **[VIBA]**. This example has γ=I. 41
4.35 If β=**ADVP[^]*** then its leftmost child must be an **R-aN**. 41

4.36 Branch off final argument **N**, special handling for "no matter". 42
4.37 Branch off final argument **N**. This example has α=A-aN and β=$\varnothing$. 42
4.38 Branch off final argument **N**. This example has α=R-aN and β=$\varnothing$. 42
4.39 Branch off final argument **N**. This example has α=R-aN and β=$\varnothing$. 43
4.40 Branch off final **VP|ADJP** as argument **A-aN**. This example has α=R-aN. . . 43
4.41 Branch off final **SBAR** as argument **E-gN** (*tough for X to Y* construction): . . . 44
4.42 Branch off final **SBAR** as argument **I-aN-gN** (*tough to Y* construction): 45
4.43 Polar question: branch off initial **B-aN**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ = VBZ, but β and γ could be different. If γ=**VB[A-Z]*** then δ=(**[Dd]oes|[Dd]o|[Dd]id|'d**). 47
4.44 Polar question: branch off initial **N**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ as it is the case almost all the time in the corpus. 47
4.45 Polar question: branch off initial **Q-bA**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ = VBZ but β and γ may be different. 48
4.46 Polar question: branch off initial **L-aN**-taking auxiliary. The γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ as it is the case almost all the time in the corpus. 48
4.47 Embedded sentence: Branch off initial complementizer. This example has α=$\varnothing$. 49
4.48 Branch off final **S** as argument **V**. 49
4.49 Embedded sentence: Branch off initial complementizer. This example has α=$\varnothing$. 50
4.50 Embedded sentence: Branch off initial complementizer. This example has α=β=$\varnothing$. 51

4.51 Branch off final predicative phrase after the copular *be* as argument **A-aN**. Top-right-most node in δ must not be an **RB**. Subtree ε, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If γ=**'s** then its immediate parent must be a **VBZ**. If ζ=**SBAR[^]*-PRD** then its left-most child must not be a **WH[^]*** or **(IN that)**. This example has α=N. 51
4.52 Branch off final post nominal phrase as argument **A-aN**. Top-right-most node in δ must not be an **RB**. Subtree ε, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If γ=**'s** then its immediate parent must be a **VBZ**. This example has α=N. 52
4.53 Gerund: branch off final argument **N**. All but three sentences in the corpus have the need to have $\beta \not\equiv \delta$ as shown by the magenta example. This requires that δ not have any category in **VB|JJ|MD|TO|NN**. The rest of the matches of this rule are when $\beta \equiv \delta$ as depicted by the blue example. 53
4.54 Branch off final **S** as modifier **[VIBA]**. This example has γ=V. 53
4.55 Branch off final argument **L-aN**. 54
4.56 Branch off initial modifier **R-aN** with colon. This example has α=$\varnothing$. 56
4.57 Branch off initial modifier **R-aN** and **I-aN**. This example has α=S. 57
4.58 Branch off initial modifier **R-aN** and **A-aN**. This example has α=V-aN. 58
4.59 Branch off initial modifier **R-aN** from **SBAR**. If β=**IN** then γ must not be either $that$, for, $where$ or $when$. 59
4.60 Branch off initial **RB** and **JJS** as modifier **R-aN** (e.g. "at least/most/strongest/weakest"). The left-most branch of ε that started from α must not be **CC**. 59
4.61 Branch off initial modifier **R-aN**. The top-left node of γ must not be a **CC**. This example has α=S. 60
4.62 Branch off initial modifier **R-aN** (including determiner, e.g. *both in A and B*. The top-left node of γ must not be a **CC**. This example has α=S. 61
4.63 The left-most branch of γ started from α must not be a **PP** or **WHPP**. This example has α=N. 61
4.64 Branch off initial modifier **R-aN** (including determiner, e.g. *both in A and B*. The top-left node of γ must not be a **CC**. This example has α=N. 62

4.65 Branch off initial modifier **R-aN-x-iN/R** of **A-aN/R-aN**. The δ in this rule must not have any PTB category that contains a vertical bar such as **PRT|ADVP**. This example has α=R-aN and β=-iN. 62
4.66 Branch off initial modifier **R-aN** of **A-aN** or **R-aN**. If β=**IN** then δ must also be of category **IN**. If β=**SBAR[^]*** then its left-most branch must be of category **IN** and the child of this child must not be $that$. This example has α=R-aN. . . 63
4.67 Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=A-aN-x and β=-iN. 63
4.68 Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. The magenta example has α=A-aN-x and β=$\varnothing$. The blue example has α=A-aN-x and β=-iN. 64
4.69 Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=R-aN-x and β=$\varnothing$. 64
4.70 Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=R-aN-x and β=$\varnothing$. 64
4.71 Branch off middle modifier **A-aN** colon. 65
4.72 Branch off final **SBAR** as modifier **A-aN**. If γ is not ending with **(-LOC|-TMP)** then the top-left node of δ must be an **IN** and its child must not be a **that**. . . . 66
4.73 Branch off final modifier **A-aN** appositive **N**. In this rule, α must not have a child of category **CC**. 66
4.74 Branch off final modifier **A-aN** infinitive phrase (with **TO** before any **VB**). In this rule, γ must not contain any **VB**. 67
4.75 Branch off final modifier **A-aN**. If γ=**SBAR[^]*** then its left-most child must be **IN** covering something not a $that$. This example has α=β=γ=$\varnothing$. 68
4.76 Branch off final modifier **AP** infinitive phrase (with **TO** before any **VB**). . . . 68
4.77 Branch off final modifier **R-aN** (extraposed from argument) This example has α=L, β=-aN, γ=$\varnothing$. 69
4.78 Branch off final **(IN|TO) + NP** as modifier **R-aN**. 70
4.79 Branch off final **S-ADV** with empty subject as modifier **R-aN**. 71
4.80 Branch off final **S-ADV** with empty subject as modifier **R-aN**. 72
4.81 Branch off final **S-ADV** with empty subject as modifier **R-aN**. 72
4.82 Branch off final 'so' + **S** as modifier **R-aN**. This rule either has $\delta \equiv \gamma$ or δ is the left-most leaf branch of γ. 73

4.83 Branch off final **S-ADV** or **S-PRP** as modifier **R-aN**. This example has γ=A. . 73
4.84 Branch off final **SBAR** as modifier **R-aN** colon. The top-left node in β must not be a **[^]*-ADV** and its sibling, if β has more than one top-level node, must not be a **([^]* :)**. 74
4.85 Branch off final **SBAR** as modifier **R-aN**. If γ is not ending with **-ADV**, **-LOC**, **-TMP**, or **-CLR** then the top-left node of δ must be an **IN** and its child must not be a **that**. 74
4.86 Branch off final modifier **R-aN** colon. The top-left node in β must not be a **.*-ADV** and its next sibling, if β has more than one top level node, must not be a **(.* :)**. 75
4.87 Branch off final modifier **R-aN**. This rule does not allow γ to have **-PRD**. . . . 75
4.88 Branch off final modifier **R-aN**. The γ of this rule must not contain a **-PRD**. . 76
4.89 Pinch ... CC ... -NONE- and re-run. This example has α=N-aD. 77
4.90 Branch off initial colon in colon...semicolon...semicolon construction. This example has α=A-aN and β=$\varnothing$. 78
4.91 Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. If ε is a single character category such as **:** then it must be a pre-terminal and ζ=**;**. This example has α=V-aN. 79
4.92 Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. This example has α=A-aN. 80
4.93 Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. If ε is a single character category such as **:** then it must be a pre-terminal and ζ=**;**. This example has α=V-aN. 81
4.94 Branch off initial conjunct prior to comma delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. This example has α=N. 82
4.95 Branch off initial conjunct prior to conjunct delimiter. 83
4.96 Branch off initial conjunct prior to conjunct delimiter. 83

4.97 Branch off initial semicolon delimiter. β=α if α is not a composite category. Otherwise, β={α}. If η=**ADVP** then the left-most pre-terminal tree in θ must either be **(RB then)** or **(RB not)**. If η is a single character category such as **:** then it must be a pre-terminal and θ=**;**. This example has α=N, β=α=N, and γ=$\varnothing$. 84

4.98 Branch off initial comma delimiter. If η=**ADVP** then the left-most pre-terminal tree in θ must either be **(RB then)** or **(RB not)**. This example has α=A-aN, β={α}={A-aN}, and γ=$\varnothing$. 84

4.99 If ζ=**ADVP** then the left-most pre-terminal tree in η must either be **(RB then)** or **(RB not)**. β=**-c**α or **-c{**α**}** depending on whether α is a primitive or composite category. Subtree ε must not be empty. This example has α=A-aN, β=-c{α}=-c{A-aN}, γ=-pPc and δ=$\varnothing$. 85

4.100 Branch off initial conjunct prior to conj delimiter (and don't pass -p down). If ζ=**ADVP** then the left-most pre-terminal tree in η must either be **(RB then)** or **(RB not)**. If ζ is a primitive category such as **:** then it must be a pre-terminal and η=**;**. β=α or **{**α**}** depending on whether α is a primitive or composite category. Subtree ζ must not be empty. This example has α=V-aN, β=-c{α}=-c{V-aN}, γ=-pPc and δ=-gN. 85

4.101 Branch off initial conjunct prior to semicolon delimiter. β=**-c**α or **-c{**α**}** depending on whether α is a primitive or composite category. Neither δ nor η could be empty. This example has α=N, β=-cα=-cN and γ=-pPs. 86

4.102 Branch off initial conjunct prior to comma delimiter. β=α or **{**α**}** depending on whether α is a primitive or composite category. Neither δ nor η could be empty. This example has α=N, β=-cα=-cN and γ=-pPc. 86

4.103 Branch off initial conjunct delimiter and final conjunct (no -p to remove). If δ=**ADVP** then the left-most pre-terminal tree in ε must either be **(RB then)** or **(RB not)**. If δ is a primitive category such as **:** then it must be a pre-terminal and ε=**;**. β=**-pP[cs]-c**α or **-pP[cs]-c{**α**}** depending on whether α is a primitive or composite category. This example has α=A-aN, β=-pPc-c{α}=-pPc-c{A-aN}, and γ=$\varnothing$. 87

4.104 Branch off initial conjunct delimiter and final conjunct (and don't pass -p down). If δ=**ADVP** then the left-most pre-terminal tree in ε must either be **(RB then)** or **(RB not)**. This example has α=A-aN, β=-c{A-aN}-pPc, and γ=-hN. 87
4.105 Branch off final **SBAR** as modifier **I-aN-gN**. This is an Fa rule **N** + **I-aN-gN** = **N**. 89
4.106 Topicalized sentence: branch off initial topic **N**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules. 90
4.107 Branch off initial relative adverbial phrase with empty subject ('when in rome'). This example has α=V-rN, β=$\varnothing$ and i=1. This is an Fb rule **R-aN-rN** + **A-aN-g{R-aN}** = **A-aN-rN** followed by a type changing rule to change an **A-aN** to a **V**. 91
4.108 Embedded question: branch off initial interrogative **R-aN** of $whether$ or if. This example has α=$\varnothing$. This is one of the Fc rules. 91
4.109 Embedded question / nom clause: branch off initial interrogative **R-aN** and final modifier **I-aN** with **R-aN** gap 92
4.110 Branch off final **SBAR** as modifier **C-rN**. This is a relative pronoun attachment rule Ra. 94
4.111 Branch off final **SBAR** as modifier **C-rN** (that|nil). This is a relative pronoun attachment rule Ra. 95
4.112 Branch off final **SBAR** as modifier **V-rN**. This is a relative pronoun attachment rule Ra. 96
4.113 Branch off final **SBAR** as modifier **V-rN**. This is a relative pronoun attachment rule Ra. 97
4.114 Branch off initial modifier **V-rN** from **SBAR-ADV** or **SBAR-TMP**. This is a relative pronoun attachment rule Rb. 98
4.115 Branch final right-node-raising modifier **A-aN**. This example has α=N-aD. . . 99
4.116 Branch final right-node-raising complement **N**. This example has α=V-aN. . . 100
4.117 Branch off final **SBAR** as modifier **A-aN** then **N** (nominal clause): Direct children of α in β must not be any **CC**. 101
4.118 Imperative sentence: delete empty **NP**. Top level node in δ must not be a **VP** head, i.e. one of (**VB**, **JJ**, **MD**, **TO**) 101
4.119 Transform a modifier of null-subject-**S** into a null-subject-**S** with a modifier of **VP**. 103

4.120 Transform a conjunction of multiple null-subject-**S**'s into a null-subject-**S** with a **VP**-conjunction. 104
4.121 Percolate a **VP-TOBEVP**-head if this head has a direct child of category **VB[ZDP]** or **MD**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**. 105
4.122 Percolate a **VP-TOBEIP**-head if this head has a direct child of category **TO**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**. 106
4.123 Percolate a **VP-TOBEBP**-head if this head has a direct child of category **VB**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**. 106
4.124 Percolate a **VP-TOBEAP**-head if this head has a direct child of category **VB[GN]**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**. 106
4.125 Percolate a **VP-TOBE*x*P**-head if this head is the head of a conjunction and the left conjunct has been annotated with a **VP-TOBE*x*P** already. This is a variation of Figure 4.126 where it looks at the left conjunct instead of the right one. . . . 107
4.126 Percolate a **VP-TOBE*x*P**-head if this head is the head of a conjunction and the right conjunct has been annotated with a **VP-TOBE*x*P** already. This is a variation of Figure 4.125 where it looks at the right conjunct instead of the left one. 107
4.127 Percolate a **VP-TOBE*x*P**-head if this head has a direct child **VP-TOBE*x*P**-head already. 108
4.128 Percolate a **VP-TOBEVP**-head if this head has only one child which is a null element. 108
4.129 Percolate a **VP-TOBEAP**-head if this head has no direct child of category **VP.***. This means the β in this rule has not top-level node of type **VP.***. 108
4.130 The null-subj usually is **(NP-SBJ (-NONE- .*))**. Percolate an **S-TOBEAS**-head if it has a direct child of category **[^]*-PRD** as this is the PTB marker of "predicative". 109
4.131 If **S** covers a null-subject and a **VP-TOBE*x*P** then append a **-TOBE*x*S** to it. This is a more specific rule of the one in Figure 4.132 where we don't even need a null-subject. 110
4.132 If **S** covers a **VP-TOBE*x*P** then append a **-TOBE*x*S** to it. This is a more general version of the rule in Figure 4.131. 110

4.133 If **S** covers a conjunction and one of the conjunct was already percolated with a **-TOBE*x*S** then append a **-TOBE*x*S** to it. This rule has the conjunct as a left one. Figure 4.134 as a similar version of this rule but for the right conjunct. . . 111
4.134 If **S** covers a conjunction and one of the conjunct was already percolated with a **-TOBE*x*S** then append a **-TOBE*x*S** to it. This rule has the conjunct as a right one. Figure 4.133 as a similar version of this rule but for the left conjunct. . . . 111
4.135 If **S** covers another **S** that was already percolated with a **-TOBE*x*S** then keep percolating on to this parent **S**. 112
4.136 If **S** covers only a null-element, append it with a **-TOBEVS**. 112
4.137 The β in this rule has no top-level node of type **VP.***. If **S** has no children of type **VP.*** then it is percolated with a **-TOBEAS**. 112
5.1 Syntax parsing for GCG. The darker color rectangles denote the commands with arrows coming in as inputs and going out as outputs. The number *i* in the yellow circle at the upper right corner of the command is referred to as *step i* in the writeup description of these commands for the figure. The lighter color rectangles denote the extensions of the files being generated or consumped by the commands. The special lighter color rectangles with a shade denote the files from corpora, i.e. "PTB" for PennTreebank corpus. There are two kinds of arrows: the black ones are for the flow of gold data and the blue ones are for the hypothesis data. If a command has at least one incoming blue arrow then its outgoing arrow must be a blue one. The two outputs going out of this Figure are ".parsed.linetrees" (hypothesis) and ".gcg13.linetrees" (gold). They will be used in a number of different syntax, dependency, filler-gap, and proposition evaluations for GCG. 119
5.2 Syntax evaluation for GCG. The hypothesis file ".parsed.linetrees" and gold file ".gcg13.linetrees" inputs to the standard "evalb" script are coming from the outputs toward the end of Figure 5.1. The output ".gcg13.syneval" of this command is the official result of the GCG evaluation on syntax parsing. This result is shown in Table 5.1. 121

5.3 Syntax parsing for CCG. The darker color rectangles denote the commands with arrows coming in as inputs and going out as outputs. The number *i* in the yellow circle at the upper right corner of the command is referred to as *step i* in the writeup description of these steps. The lighter color rectangles denote the extensions of the files being generated or consumped by the commands. The special lighter color rectangles with a shading denote the files from corpora, i.e. "CCG Bank" for CCG Treebank corpus. There are two kinds of arrows: the black ones are for the flow of gold data and the blue ones are for the hypothesis data. If a command has at least one incoming blue arrow then its outgoing arrow must be a blue one. The two outputs going out of this Figure are ".ccg.parsed.linetrees" (hypothesis) and ".ccg.linetrees" (gold). They will be used in a number of different syntax, dependency, and filler-gap evaluations for CCG. 124

5.4 Syntax evaluation for CCG. The hypothesis ".ccg.parsed.linetrees" and gold ".ccg.linetrees" inputs to the standard "evalb" of *step 1* are coming from the outputs toward the end of Figure 5.3. The output ".ccg.syneval" of this command is the result of the CCG evaluation on syntax parsing. This result is shown in Table 5.2. The standard "evalb" used in this Figure is exactly the one used in Figure 5.2. This shows that the syntax evaluations for GCG and CCG are done in the exact same way, only different in the grammar formalism. 126

5.5 Student's t-test to measure significance for syntax evaluation between GCG and CCG. The two inputs ".gcg13.syneval" and ".ccg.syneval" are coming from the last outputs of Figure 5.2 and Figure 5.4, respectively. The final output produced is ".gcg13.ccg.ttestsignif" as shown in Table 5.3. This result shows the syntax evaluation on GCG is significantly more accurate than that of CCG. 128

5.6 McNemar's significance test for syntax evaluation between GCG and CCG. The upper pair of inputs are ".parsed.linetrees" and ".gcg13.linetrees", coming from the outputs toward the end of Figure 5.1 to represent the syntax parsing on GCG. The lower pair of inputs are ".ccg.parsed.linetrees" and ".ccg.linetrees", coming from the outputs toward the end of Figure 5.3 to represent the syntax parsing on CCG. The *step 1* is duplicated in 2 places to make it clear by reducing the number of its inputs and outputs. This *step 1* extracts only the perfect matches in parsing on either GCG or CCG. The ".corr" file name implies taking only the perfectly "correct" parsed sentences. The *step 2* is an R script to compute the McNemar significance test. This result as shown in Table 5.4, one more time, confirms that GCG is significantly more accurate than CCG on syntax parsing. . 129

5.7 "onlyval" relaxation drops the check for correctness of each individual primitive category as they were all **X**'ed out. This also drops the **-l** tag to not care about the local predicate-argument dependencies. This relaxation is meant to only check the correct compositional structure of the syntax category at each node as well as the correct structure of the tree overall. 131

5.8 "unlabeled" relaxation turns each syntax category at every node into just an **X**. This relaxation therefore only evaluates the parsing on its capability to recover the tree structure. 134

6.1 The format of these dependencies presented as "word1/number/word2" to mean "word2" is a numeric argument "number" of predicate "word1." For example, in the phrase "... it expects ... sales remain ...", the "expects" predicate has 3 arguments: "it" is its argument 1, "remain" is its argument 2, and "sales" is its argument 3. Argument 0 is used as the identify relation of each word and usually tagged along with the syntax category of the word, a pound sign (#) as a delimiter, and the word itself. For brevity in the code, we use the lowercase initial of the word instead of the word itself over and over again at all the non-zero relations. 138

6.2 A complete example of GCG tree for the sentence: *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990* . 138

6.3 Compute Dependency Relations for GCG. 139

6.4 Dependency evaluation for GCG. 140

6.5 Compute Dependency Relation for CCG. 140
6.6 Dependency evaluation for CCG. 141
6.7 Student's t-test to measure significant for dependency evaluation between GCG and CCG. The two inputs ".gcg13.depeval" and ".ccg.depeval" are coming from the last outputs of Figure 6.4 and Figure 6.6, respectively. The final output produced is ".gcg13.ccg.depeval.ttestsignif" as shown in Table 6.3. This result shows the dependency evaluation on GCG is significantly better than that of CCG. 143
6.8 McNemar's significance test for dependency evaluation between GCG and CCG. The upper pair of inputs are ".parsed.melconts" and ".melconts", coming from the outputs toward the end of Figure 6.3 to represent the dependency parsing on GCG. The lower pair of inputs are ".ccg.parsed.melconts" and ".ccg.melconts", coming from the outputs toward the end of Figure 6.5 to represent the dependency parsing on CCG. The *step 1* is duplicated in 2 places to make it clear by reducing the number of its inputs and outputs. This *step 1* extracts only the perfect matches in dependency parsing on either GCG or CCG. The ".corr" file name implies taking only the perfectly "correct" dependency parsed sentences. The *step 2* is an R script to compute the McNemar significance test. This result as shown in Table 6.4, one more time, confirms that GCG is significantly more accurate than CCG on dependency parsing. 145
7.1 Filler gap evaluation for GCG. 150
A.1 Branch **N** -> **D A-aN-x**: 'the best' construction. This rule consists of a type changing rule to change an **A-aN** to an **N-aD** and an initial argument attachment rule Aa. 161
A.2 **[VIBLAG]** sentence: branch off initial **E** subject. This rule embedded a type changing rule to change an **E** to an **N** to enable a final argument attachment rule Ae. 162
A.3 Branch off initial modifier **A-aN-x**. If ε=**QP** then its leftmost child in ζ must be of category **$**. This example has α=-aD, β=-bV-bO. This is an initial modifier attachment rule Ma. 163

A.4 Branch off initial modifier **A-aN-x**. If γ=**SBAR[^]*** then its left-most child in δ must be of category **IN** having child not **that**. If ζ=**QP** then its leftmost child in η must be of category **$**. This example has α=-aD, β=-bV-bO. This is an initial modifier attachment rule Ma. 164

A.5 Rebinarize currency unit followed by **QP**. If $\varepsilon \neq \varnothing$ then ε=**(-NONE- *U*)**. This rule embedded a type changing rule to change a **$** to an **N** that will enable a final modifier attachment rule Me. 164

A.6 Branch off initial modifier **A-aN-x**. If ε=**QP** then its leftmost child in ζ must be of category **$**. This example has α=-aD, β=$\varnothing$. This rule embedded a type changing rule to change an **N** to an **A-aN** to enable an initial modifier attachment rule Ma. 165

A.7 Branch off initial modifier **A-aN-x**. Both γ and δ are not $\varnothing$. This example has α=N-aD and β=$\varnothing$. This is an initial modifier attachment rule Ma. 165

A.8 Content question: branch off initial interrogative **R-aN**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fc rules **R-aN-iN** + **Q-g{R-aN}** = **Q-iN** that is followed by a type changing rule to change a **Q-iN** to an **S**. 166

A.9 Embedded question / nom clause: branch off initial interrogative **N** and final modifier **I-aN** with **N** gap. This rule combines an Fc rule **N-iN** + **I-aN-gN** = **I-aN-iN** that is followed by a type changing rule to change an **I-aN** to a **V**. . . 167

A.10 Topicalized sentence: branch off initial topic **S** (possibly quoted). γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules. 168

A.11 Topicalized sentence: branch off initial topic **A-aN**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules. 169

A.12 Topicalized sentence: branch off initial topic **R-aN**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules. 170

A.13 Embedded question / nom clause: branch off initial interrogative **N**. This is an Fe rule. 170

A.14 Embedded question / nom clause / nom clause modifier: branch off initial interrogative **R-aN**. This is an Fe rule 171

A.15 Branch off final **SBAR** as modifier **I-aN-g{R-aN}**. This is an Fa rule **N** + **I-aN-gN** = **N**. 172

A.16 Polar question: unary expand to **Q**. γ is the left-most pre-terminal under $\alpha\beta$. If γ=**VB[A-Z]*** then δ=**[Dd]oes|[Dd]o|[Dd]id|[Ii]s|[Aa]re|[Ww]as|[Ww]ere** or **[Hh]as|[Hh]ave|[Hh]ad**. This example has α=S, β=$\varnothing$, γ=VBD, and ε=NP-SBJ. 173

A.17 Imperative sentence: unary expand to **B-aN**. In this rule, top level nodes in δ must not be any of **VB**, **JJ**, **MD**, or **TO**. This means ε is the first **VP**-head child of γ. This type changing rule changes a **B-aN** to an **S**. 174

A.18 Polar question: allow subject gap without inversion. This type changing rule changes a **V-aN** to a **Q-gN**. This example has α=Q-gN, i=23, and β=$\varnothing$. 174

A.19 Implicit-pronoun relative: delete initial empty interrogative phrase. This type changing rule changes a **V-gN** to a **C-rN**. This example has α=C, β=-rN and i=1. 175

A.20 Implicit-pronoun relative: delete initial empty interrogative phrase as adverbial. This type changing rule changes a **V-g{R-aN}** to a **C-rN**. This example has α=C, β=-rN and i=2. 175

A.21 Branch off final **SBAR** as extraposed modifier **C** relative clause. Subtree ε under δ must not contain **-NONE-** or $what$. This example has i=3. 176

A.22 Branch off final SBAR as extraposed modifier **I-aN**. The part **[^ x]*** in α is to ensure it won't match with intransitive categories which are ending in **-aN-x**. This example has α=V-aN, i=1 and j=2. 177

A.23 Branch off initial parenthetical sentence with extraction. 178

A.24 Inverted declarative sentence: branch off final subject. 178

A.25 Branch off initial modifier **A-aN-x**. if η=**QP** then its leftmost child in θ must be of category **$**. This example has α=β=δ=$\varnothing$ and γ=-iN. 179

A.26 Rebinarize **QP** containing dollar sign followed by ***U***, and continue. If $\zeta \neq \varnothing$ then ζ=**(-NONE- *U*)**. 179

A.27 Branch off currency unit followed by non-final ***U***. 180

Chapter 1

Introduction

Recent years have been experiencing so much excitement and opportunity for computational natural language understanding. These natural language understanding tasks include, but are not limited to, the deployment of widespread commercial systems based on natural language understanding to answer questions about flight departure or arrival time; give directions about geographical information; report on bank account balances or even perform simple financial transactions. The fast growing adoption of mobile devices into society also enable the development of applications that aim to act as a personal assistant communicating to the device owner using natural language like Siri (iOS) and Skyvi (Android). More sophisticated research systems may begin to generate concise summaries of news articles, mine the entire world wide web to look for information of interest, answer fact-based questions, and recognize complex semantic and dialogue structure.

This is not to say that the problem of computational language understanding is solved. Lots of these systems operate on closed domains, recognize only limited sets of rules and/or keywords which are domain specific and carefully hand-crafted, or rely upon some off the shelf parsers that offer little to no way of changing their syntactic analyses to experience different levels of generalities and learnabilities on different specific downstream tasks. It is therefore comes the challenge to research and develop a robust, domain-independent system that can offer a great level of flexibility in choosing any different syntactic analysis for any targeted language construct. To step toward these goals, this hypothetical system must (1) be able to model deep syntactic and at least shallow semantic representation in term of predicate-argument dependencies, and (2) to have a flexible grammar acquirable by mean of automatically reannotating some

available corpus such as the Penn Treebank (Marcus et al., 1993).

This thesis presents the research and development of such a comprehensive system known as "modelblocks" that is publicly available on sourceforge.net. It is organized in 7 chapters following this introduction as outlined below:

- Chapter 2 describes the background or related work of this research. It talks about CCG (Steedman, 2000, Hockenmaier and Steedman, 2007) and HPSG (Pollard and Sag, 1994, Miyao et al., 2004) as two other mainstream research directions toward these goals including grammar formalisms and the reannotation of Penn Treebank (Marcus et al., 1993) into their grammar representations.

- Chapter 3 provides details of a Generalized Categorial Grammar (GCG) formalism. It shows areas of similarities to both CCG and HPSG but highlights the differences that set itself apart and can be considered novel contributions empirically on a number of different downstream tasks. The syntax representation of GCG can be programmatically and deterministically transformed into a shallow semantic representation as structures of predicate-argument dependencies. This is not so straightforward for CCG and HPSG.

- Chapter 4 is about the details of the reannotation process to acquire a corpus in this GCG annotation from the well-known Penn Treebank (PTB). This reannotation is again programmatically and deterministically done using a set of about 175 rules implemented in Perl. This set of rules is modular and easy to change to support experiments using different syntactic analyses. From this perspective, modelblocks is not just a tool that can be taken as is, but a tool to build different tools to suit different purposes or downstream tasks. This is another difference between this system compared to CCG and HPSG.

- Chapter 5 describes the evaluation of our GCG grammar on syntax parsing task. This chapter also shows the same parsing evaluation on the competitive CCG. The two evaluation results show that GCG is more accurate on section 23 of WSJ, but a Student's t-test and a McNemar's test were done to generalize the claim that the reannotated GCG can parse significantly better than CCG.

- Chapter 6 is on the same methodology as Chapter 5 but evaluating the recovery of syntactic dependency relations instead of constituent parsing. The results also show that our GCG grammar is significantly better than the competitive CCG on this task.

- Chapter 7 evaluates GCG on the recovery of unbounded dependency relations, and filler-gap constructions specifically. This evaluation, leveraging results from Rimell et al. (2009) on this same task, shows that GCG outperforms by a good margin on unbounded dependency recovery compared to seven other systems. This result is stronger than what was reported in our paper that earned the Best Paper Award at COLING 2012.

Chapter 2

Background

The work presented in this thesis can be roughly divided into 3 parts: (1) A formalism of a new Generalized Categorial Grammar (GCG), (2) The reannotation process to convert Penn Treebank into this GCG representation, and (3) The various evaluations of the system on constituent parsing, syntactic dependency parsing, and unbounded dependency recovery. This chapter will do a survey of other research approaches to these ideas.

2.1 CCG and HPSG

A question one may ask is why do we need another grammar? There are very many out there already. A grammar is just a way to impose some rules or dependency structures on a language to make it easy to explain, teach, or validate it. For example, linguists come up with sets of syntactic categories as ways to label different types of language constituents in a consistent manner in order to describe language in a formal way. Computational linguists use grammars for parsing or generating language structures to solve a number of downstream tasks such as question answering, information retrieval, etc. One grammar may be proven as a good fit for some tasks but not others. We are interested in finding or developing a grammar that allows a relatively easy way to change some syntactic categories for the purpose of experimenting with different syntactic analyses on some particular downstream task. An example would be to either collapse or separate the two commonly occurring categories for post-nominal modifier phrases and predicative phrases. CCG (Steedman, 2000, Hockenmaier and Steedman, 2007) and other grammars treat these two phenomena as separate categories, but we can show that collapsing

them together yield much better results on the unbounded dependency recovery task. This is the reason we want to build a new grammar, a meta-grammar that can help people build and experiment on their own different grammars to fit their needs.

As a new grammar, our GCG is closely related to CCG (Steedman, 2000, Hockenmaier and Steedman, 2007) and HPSG (Pollard and Sag, 1994, Miyao and Tsujii, 2005). Like CCG or other categorial grammars, it imposes a small, uniform, and easily learnable set of semantic composition operations based on functor-argument relations, but like HPSG, is generalized to limit the number of categories used to those needed to enforce grammatical constraints.

2.2 Reannotation from Penn Treebank to CCG and HPSG

Having a good grammar formalism written out on paper is not so valuable if it's not accompanied by a sufficient coverage corpus to test out its generality and learnability, but annotating a wide coverage corpus by hand is practically impossible. We therefore built a reannotation system to map existing resources based on Government and Binding Theory, like the Penn Treebank, into this categorial representation in much the same way Clark and Curran (2007) and Miyao and Tsujii (2005) did for their CCG and HPSG formalisms respectively. The difference between our reannotation system and these two reannotations lies in its simplicity and capability to adapt to changes to support alternative syntactic analyses. As a system, our reannotation consists of 175 rules and is designed to be applied deterministically in a single top-down pass, pulling arguments and modifiers out of constituents until only a lexical head remains at the bottom. This single-pass architecture allows the rules for reannotating various linguistic phenomena to be relatively modular, so that they can be independently manipulated and evaluated. Both CCG and HPSG reannotation involved multiple phases for head-finding, binarization, and labeling. They both rely up on some heuristic component, so cannot be fully deterministic or automatic.

2.3 System Evaluations

The reannotated GCG grammar is used to define a probabilistic model which is automatically annotated with additional latent variable values (Petrov and Klein, 2007) and used to parse the constituent and syntactic dependencies from input sentences of the Wall Street Journal and from

a minimal but special corpus introduced by (Rimell et al., 2009) that contains only sentences having Object extraction from a relative clause, Object extraction from a reduced relative clause, Subject extraction from a relative clause, Free relatives, Object wh-questions, Right node raising, and Subject extraction from an embedded clause. This corpus was designed specifically to test various parsers on their capability to recover these unbounded dependencies as studied by (Rimell et al., 2009, Nivre et al., 2010).

Chapter 3

Generalized Categorial Grammar

This chapter describes the formalism of our Generalized Categorial Grammar (GCG), which has the transparent predicate-argument dependencies of traditional categorial grammars (based on function application), but is generalized to allow arbitrary sets of type-constructing operators. An extended set of type-constructing operators and a corresponding set of inference rules are then used to group syntactically-interchangeable signs — for example, those with peripheral and non-peripheral gaps or those occurring in post-nominal and predicative contexts — into equivalent categories.

A generalized categorial grammar (Lambek, 1958, Bach, 1981, Oehrle, 1994) is a tuple $\langle U, O, R, X, M \rangle$ of a set U of primitive category types, a set O of type-constructing operators, a set R of inference rules, a set X of vocabulary items, and a mapping M from vocabulary items to complex types. The set of primitive category types U specify various linguistic forms for descriptions of entities or eventualities, corresponding to different clause types,[1] e.g.:

V: finite verbal (*they knew it*)
I: infinitive verbal (*them to know it*)
B: base-form verbal (*them know it*)
L: participial verbal (*them known it*)
A: adjectival/predicative (*them knowing it*)
R: adverbial (*them knowingly*)
G: gerund (*them knowing it*)
N: nominal form (e.g. *their knowledge of it*)
D: determiner (*their knowledge of it's*)
O: genitive (*of their knowledge of it*)
E: embedded infinitive (*for them to know it*)
C: complementized finite (*that they know it*)
Q: interrogative (*did they know it*)
S: complete utterance (*know it*)

[1] This system of categories may be viewed as a simplification of a tectogrammatical type system such as that of Mihalicek and Pollard (2010), weakened to be representable as a context-free grammar.

The set of type-constructing operators O specify various kinds of arguments:[2]

-a: initial argument
-b: final argument
-c: initial conjunct
-d: final conjunct
-g: filler-gap argument
-h: held argument for right node raising
-i: interrogative pronoun argument
-r: relative pronoun argument

Using this set of primitive category types U and type-constructing operators O, a set of complex categories C can be defined such that:

1. every U is in C
2. every $C \times O \times C$ is in C
3. nothing else is in C

Mapping M defines associations from vocabulary items $x \in X$ to meaning functions and associated categories of the form '$(\lambda ...) : u\varphi_1...\varphi_v\psi$', where '$(\lambda ...)$' is a meaning function and '$u\varphi_1...\varphi_v\psi$' is a category consisting of output category $u \in U$, a sequence of argument categories $\varphi_1, ..., \varphi_v \in \{$**-a**, **-b**, **-c**, **-d**$\} \times C$, and an optional non-local argument category $\psi \in (\{$**-r**, **-i**$\} \times C) \cup \{\epsilon\}$. Since this model will be used to generate predicate-argument relations but not scoping relations, these meaning functions are constrained to describe simple existentially-quantified variables over instances of entities or eventualities, connected by a set of numbered argument relations. These meaning functions map instances of entities or eventualities i, j, k to truth values based on whether the described argument relations hold between these referents. These argument relations are defined as numbered functions $(v\, i)$=j from eventuality or predicate instances i to argument instances j identified by the number of the function v. The '0' function identifies j as i's predicate concept (so '0' maps entity or eventuality instances to instances of concepts associated with words in X), the '1' function identifies j as i's first argument (e.g. its subject), the '2' function identifies j as i's second argument (e.g. its direct object), and so on.[3] A graphical representation of the predicate-argument relations generated by this

[2] The **-a** and **-b** operators may be viewed as equivalent to the forward and backward slash, respectively, of Lambek (1958) or Bar-Hillel (1953) categorial grammars, except that they are not used to represent gap arguments or conjuncts. The **-g** operator is similar to the vertical or neutral slash of Kubota and Levine (2012), used to represent gap arguments. The **-c** and **-d** operators for conjuncts, the **-h** operator for rightward raising, and the **-r** and **-i** operators for relative and interrogative pronoun referents are novel extensions to the system.

[3] More sophisticated meaning functions are possible, but are not necessary for evaluating the accuracy of unbounded dependency recovery.

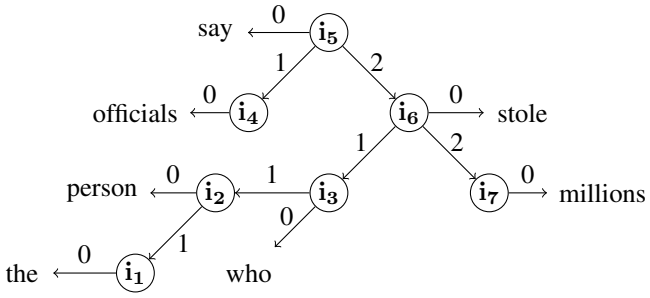


Figure 3.1: Graphical representation of predicate-argument dependencies for the sentence *The person who officials say stole millions.*

system for the sentence *The person who officials say stole millions,* is shown in Figure 3.1. This is similar to the semantic dependency representations of Mel'čuk (1988) and Parsons (1990).

The meaning functions associated with most words specify just the predicate concept (which is here defined to match the word x):

$$x \;\mapsto_M\; (\lambda_i\,(0\,i){=}x) : u\varphi_1 ... \varphi_v \tag{3.1a}$$

Meaning functions for relative pronouns (Equation 3.1b) and interrogative pronouns (Equation 3.1c) introduce additional arguments k, using operators **-r** or **-i** for the referent of the antecedent of a relative or interrogative pronoun respectively:

$$x \;\mapsto_M\; (\lambda_{k\,i}\,(0\,i){=}x \wedge (v\,i){=}k) : u\varphi_1 ... \varphi_{v-1}\textbf{-r}c \tag{3.1b}$$

$$x \;\mapsto_M\; (\lambda_{k\,i}\,(0\,i){=}x \wedge (v\,i){=}k) : u\varphi_1 ... \varphi_{v-1}\textbf{-i}c \tag{3.1c}$$

Inference rules are defined in terms of *composition functions* for arguments, modifiers, and conjuncts. These composition functions each take a meaning function g for an initial (left) child sign and a meaning function h for a final (right) child sign (each defining a set of entity or eventuality instances) and return a meaning function for the parent, which is itself a function from entity or eventuality instances i to truth values:

- Composition functions for arguments $f_{u\varphi_1 ... \varphi_v}$ connect the referent j of an initial (left) child function g as an argument of referent i of a final (right) child function h, or vice

versa:

$$f_{u\varphi_1 \ldots \varphi_{v-1}\textbf{-a}c} \stackrel{\text{def}}{=} \lambda_{g\,h\,i}\, \exists_j\, (v\,i){=}j \wedge (g\,j) \wedge (h\,i) \tag{3.2a}$$

$$f_{u\varphi_1 \ldots \varphi_{v-1}\textbf{-b}c} \stackrel{\text{def}}{=} \lambda_{g\,h\,i}\, \exists_j\, (v\,i){=}j \wedge (g\,i) \wedge (h\,j) \tag{3.2b}$$

- Composition functions for initial and final modifiers (f_{IM} and f_{FM}) are category-independent and return the referent of the argument (j) rather than of the predicate (i):

$$f_{\text{IM}} \stackrel{\text{def}}{=} \lambda_{g\,h\,j}\, \exists_i\, (1\,i){=}j \wedge (g\,i) \wedge (h\,j) \tag{3.3a}$$

$$f_{\text{FM}} \stackrel{\text{def}}{=} \lambda_{g\,h\,j}\, \exists_i\, (1\,i){=}j \wedge (g\,j) \wedge (h\,i) \tag{3.3b}$$

- Composition functions for conjuncts are similar to composition functions for arguments, except that they only count conjunct arguments, for $c, d \in C$:[4]

$$f_{c\textbf{-c}d} \stackrel{\text{def}}{=} \lambda_{g\,h\,i}\, \exists_j\, (1\,i){=}j \wedge (g\,j) \wedge (h\,i) \tag{3.4a}$$

$$f_{c\textbf{-d}d} \stackrel{\text{def}}{=} \lambda_{g\,h\,i}\, \exists_j\, (2\,i){=}j \wedge (g\,i) \wedge (h\,j) \tag{3.4b}$$

$$f_{\&} \stackrel{\text{def}}{=} \lambda_{g\,h\,i}\, \exists_{j\,k}\, j{=}(2\,i) \wedge (0\,i){=}(0\,j) \wedge (1\,j){=}k \wedge (g\,k) \wedge (h\,j) \tag{3.4c}$$

The set of *inference rules* R in the categorial grammar then apply these composition functions to compose and categorize super-lexical signs. These inference rules will use variables f, g, h over meaning functions, variables k over referents for possible values of gaps, variables $u \in U$ over primitive categories, variables $c, d, e \in U \times (\{\textbf{-a}, \textbf{-b}, \textbf{-c}, \textbf{-d}\} \times C)^*$ over categories with local arguments, and variables $\psi \in \{\textbf{-g}, \textbf{-h}, \textbf{-i}, \textbf{-r}\} \times C$ over non-local operators and argument categories:[5]

1. Inference rules for argument attachment apply functors of category $c\textbf{-a}d$ or $c\textbf{-b}d$ to initial or final arguments of category d. Non-local arguments k, using non-local operator and argument category ψ, are then propagated to the consequent from all possible combinations

[4] The last of these (3.4c) introduces lexical and compositional relations for elided conjunctions in sequences of three or more conjuncts (e.g. between *creditors* and *investors* in the conjunction *creditors, investors, and employees*).

[5] A deductive system consists of inference rules of the form $\frac{P}{Q} R$, meaning premises or antecedents P entail conclusion or consequent Q according to rule or side condition R (Shieber et al., 1995). Additionally, this notation assumes adjacent premises arise from adjacent and similarly ordered sequences of observations.

of antecedents, skipping over the composition function:

$$\frac{g{:}d \;\; h{:}\, c\textbf{-a}d}{(f_{c\textbf{-a}d}\, g\, h){:}\, c} \qquad \frac{g{:}d\psi \;\; h{:}\, c\textbf{-a}d}{\lambda_k\, (f_{c\textbf{-a}d}\, (g\, k)\, h){:}\, c\psi} \qquad \frac{g{:}d \;\; h{:}\, c\textbf{-a}d\psi}{\lambda_k\, (f_{c\textbf{-a}d}\, g\, (h\, k)){:}\, c\psi} \qquad \frac{g{:}d\psi \;\; h{:}\, c\textbf{-a}d\psi}{\lambda_k\, (f_{c\textbf{-a}d}\, (g\, k)\, (h\, k)){:}\, c\psi} \quad \text{(Aa–d)}$$

$$\frac{g{:}\, c\textbf{-b}d \;\; h{:}d}{(f_{c\textbf{-b}d}\, g\, h){:}\, c} \qquad \frac{g{:}\, c\textbf{-b}d\psi \;\; h{:}d}{\lambda_k\, (f_{c\textbf{-b}d}\, (g\, k)\, h){:}\, c\psi} \qquad \frac{g{:}\, c\textbf{-b}d \;\; h{:}d\psi}{\lambda_k\, (f_{c\textbf{-b}d}\, g\, (h\, k)){:}\, c\psi} \qquad \frac{g{:}\, c\textbf{-b}d\psi \;\; h{:}d\psi}{\lambda_k\, (f_{c\textbf{-b}d}\, (g\, k)\, (h\, k)){:}\, c\psi} \quad \text{(Ae–h)}$$

For example, to attach a verb to a direct object with or without a gap:

$$\dfrac{\dfrac{\text{read} \quad \text{a book about cars}}{\textbf{V-aN-bN} \qquad \textbf{N}}}{\textbf{V-aN}}\ \text{Ae} \qquad \dfrac{\dfrac{\text{read} \quad \text{a book about}}{\textbf{V-aN-bN} \qquad \textbf{N-gN}}}{\textbf{V-aN-gN}}\ \text{Ag}$$

2. Inference rules for modifier attachment apply initial or final modifiers of category $u\textbf{-a}d$ to modificands of category c (again propagating non-local arguments ψ to the consequent from all combinations of antecedents, so as to skip over the composition function):

$$\frac{g{:}\, u\textbf{-a}d \;\; h{:}c}{(f_{\mathrm{IM}}\, g\, h){:}c} \qquad \frac{g{:}\, u\textbf{-a}d\psi \;\; h{:}c}{\lambda_k\, (f_{\mathrm{IM}}\, (g\, k)\, h){:}c\psi} \qquad \frac{g{:}\, u\textbf{-a}d \;\; h{:}c\psi}{\lambda_k\, (f_{\mathrm{IM}}\, g\, (h\, k)){:}c\psi} \qquad \frac{g{:}\, u\textbf{-a}d\psi \;\; h{:}c\psi}{\lambda_k\, (f_{\mathrm{IM}}\, (g\, k)\, (h\, k)){:}c\psi} \quad \text{(Ma–d)}$$

$$\frac{g{:}c \;\; h{:}\, u\textbf{-a}d}{(f_{\mathrm{FM}}\, g\, h){:}c} \qquad \frac{g{:}c\psi \;\; h{:}\, u\textbf{-a}d}{\lambda_k\, (f_{\mathrm{FM}}\, (g\, k)\, h){:}c\psi} \qquad \frac{g{:}c \;\; h{:}\, u\textbf{-a}d\psi}{\lambda_k\, (f_{\mathrm{FM}}\, g\, (h\, k)){:}c\psi} \qquad \frac{g{:}c\psi \;\; h{:}\, u\textbf{-a}d\psi}{\lambda_k\, (f_{\mathrm{FM}}\, (g\, k)\, (h\, k)){:}c\psi} \quad \text{(Me–h)}$$

For example, to attach an adverbial modifier with or without a gap:

$$\dfrac{\dfrac{\text{sleep} \quad \text{in Aix}}{\textbf{V-aN} \qquad \textbf{R-aN}}}{\textbf{V-aN}}\ \text{Me} \qquad \dfrac{\dfrac{\text{sleep} \quad \text{in}}{\textbf{V-aN} \qquad \textbf{R-aN-gN}}}{\textbf{V-aN-gN}}\ \text{Mg}$$

3. Inference rules for conjunct attachment apply conjunctions of category $c\textbf{-c}d$ or $c\textbf{-d}d$ to conjuncts of category d (including repeated initial conjuncts):

$$\frac{g{:}d \;\; h{:}\, c\textbf{-c}d}{(f_{c\textbf{-c}d}\, g\, h){:}\, c} \qquad \frac{g{:}d \;\; h{:}\, c\textbf{-c}d}{(f_{\&}\, g\, h){:}\, c\textbf{-c}d} \qquad \frac{g{:}\, c\textbf{-d}d \;\; h{:}d}{(f_{c\textbf{-d}d}\, g\, h){:}\, c} \quad \text{(Ca–c)}$$

$$\frac{g{:}d\psi \;\; h{:}\, c\textbf{-c}d\psi}{\lambda_k(f_{c\textbf{-c}d}\, (g\, k)\, (h\, k)){:}\, c} \qquad \frac{g{:}d\psi \;\; h{:}\, c\textbf{-c}d\psi}{\lambda_k(f_{\&}\, (g\, k)\, (h\, k)){:}\, c\textbf{-c}d\psi} \qquad \frac{g{:}\, c\textbf{-d}d\psi \;\; h{:}d\psi}{\lambda_k(f_{c\textbf{-d}d}\, g\, (h\, k)){:}\, c} \quad \text{(Cd–f)}$$

For example, to combine three noun phrase conjuncts:

$$\dfrac{\dfrac{\text{creditors} \qquad \dfrac{\text{investors} \qquad \dfrac{\text{and} \qquad \text{employees}}{\textbf{N-cN-dN} \qquad \textbf{N}}}{\textbf{N} \qquad \textbf{N-cN}}\ \text{Cb}}{\textbf{N} \qquad \textbf{N-cN}}\ \text{Ca}}{\textbf{N}}\ \text{Cc}$$

4. Inference rules for gap attachment hypothesize gaps as initial arguments, final arguments, or modifiers:[6]

$$\frac{g{:}\ c\text{-}\mathbf{a}d}{\lambda_k\,(f_{c\text{-}\mathbf{a}d}\,\{k\}\;g){:}\ c\text{-}\mathbf{g}d} \qquad \frac{g{:}\ c\text{-}\mathbf{b}d}{\lambda_k\,(f_{c\text{-}\mathbf{b}d}\,g\,\{k\}){:}\ c\text{-}\mathbf{g}d} \qquad \frac{g{:}c}{\lambda_k\,(f_{\mathrm{IM}}\,\{k\}\;g){:}c\text{-}\mathbf{g}d} \qquad \text{(Ga–c)}$$

For example:

$$\frac{\underline{\text{is sleeping}}}{\dfrac{\mathbf{V\text{-}aN}}{\mathbf{V\text{-}gN}}\ \text{Ga}} \qquad \frac{\underline{\text{we}}\quad\dfrac{\underline{\text{drove}}}{\mathbf{V\text{-}aN\text{-}bN}}}{\dfrac{\mathbf{N}\qquad\mathbf{V\text{-}aN\text{-}gN}}{\mathbf{V\text{-}gN}}\ \begin{matrix}\text{Gb}\\ \text{Ac}\end{matrix}} \qquad \frac{\underline{\text{is sleeping}}}{\dfrac{\mathbf{V\text{-}aN}}{\mathbf{V\text{-}aN\text{-}g\{R\text{-}aN\}}}\ \text{Gc}}$$

5. Inference rules for filler attachment apply gapped clauses to modificands or relative or interrogative phrases as fillers:

$$\frac{g{:}e\;\;h{:}\,c\text{-}\mathbf{g}d}{\lambda_i\,\exists_j\,(g\,i)\wedge(h\,i\,j){:}e} \qquad \frac{g{:}d\text{-}\mathbf{r}e\;\;h{:}\,c\text{-}\mathbf{g}d}{\lambda_{k\,j}\,\exists_i\,(g\,k\,i)\wedge(h\,i\,j){:}\,c\text{-}\mathbf{r}e} \qquad \frac{g{:}d\text{-}\mathbf{i}e\;\;h{:}\,c\text{-}\mathbf{g}d}{\lambda_{k\,j}\,\exists_i\,(g\,k\,i)\wedge(h\,i\,j){:}\,c\text{-}\mathbf{i}e} \qquad \text{(Fa–c)}$$

$$\frac{g{:}e\;\;h{:}\,c\text{-}\mathbf{g}e}{\lambda_j\,\exists_i\,(g\,i)\wedge(h\,i\,j){:}d} \qquad \frac{g{:}d\text{-}\mathbf{i}e\;\;h{:}\,c\text{-}\mathbf{g}d}{\lambda_j\,\exists_{i\,k}\,(g\,k\,i)\wedge(h\,i\,j){:}\,e} \qquad \text{(Fd–e)}$$

For example:

$$\frac{\underline{\text{the car}}\quad\underline{\text{we drove}}}{\mathbf{N}\qquad\mathbf{V\text{-}gN}\ \text{Fa}} \qquad \frac{\underline{\text{which}}\quad\underline{\text{we drove}}}{\mathbf{N\text{-}rN}\qquad\mathbf{V\text{-}gN}\ \text{Fb}} \qquad \frac{\underline{\text{what}}\quad\underline{\text{do we drive}}}{\mathbf{N\text{-}iN}\qquad\mathbf{Q\text{-}gN}\ \text{Fc}}$$

$$\frac{\underline{\text{such a flight}}\quad\underline{\text{I see}}}{\mathbf{N}\qquad\mathbf{V\text{-}gN}\ \text{Fd}} \qquad \frac{\underline{\text{what}}\quad\underline{\text{we saw}}}{\mathbf{N\text{-}iN}\qquad\mathbf{V\text{-}gN}\ \text{Fe}}$$

6. Inference rules for relative pronoun attachment apply pronominal relative clauses of category c-**r**d to modificands of category e:

$$\frac{g{:}e\;\;h{:}c\text{-}\mathbf{r}d}{\lambda_i\,\exists_j\,(g\,i)\wedge(h\,i\,j){:}e} \qquad \frac{g{:}c\text{-}\mathbf{r}d\;\;h{:}e}{\lambda_j\,\exists_i\,(g\,j\,i)\wedge(h\,j){:}e} \qquad \text{(Ra–b)}$$

For example:

$$\frac{\underline{\text{the car}}\quad\underline{\text{which we drove}}}{\mathbf{N}\qquad\mathbf{V\text{-}rN}\ \text{Ra}} \qquad \frac{\underline{\text{when we left}}\quad\underline{\text{it snowed}}}{\mathbf{V\text{-}rN}\qquad\mathbf{V\text{-}aN}\ \text{Rb}}$$

[6] Here, set notation is used in order to save space: $\{k\} = (\lambda_i\, i{=}k)$.

7. Inference rules for argument elision (including determiners of plural nouns) simply leave these arguments unspecified in the resulting meaning function:

$$\frac{g\colon c\textbf{-a}d}{g\colon c} \qquad \frac{g\colon c\textbf{-b}d}{g\colon c} \qquad \frac{g\colon c\textbf{-a}d\psi}{g\colon c\psi} \qquad \frac{g\colon c\textbf{-b}d\psi}{g\colon c\psi} \tag{Ea–d}$$

For example, to elide determiners of plural nouns or optional direct objects:

$$\frac{\dfrac{\text{cars}}{\textbf{N-aD}}}{\textbf{N}}\ \text{Ea} \qquad \frac{\dfrac{\text{drive}}{\textbf{V-aN-bN}}}{\textbf{V-aN}}\ \text{Eb}$$

8. Inference rules for right node raising introduction and attachment treat right-node raising as a type of non-local argument using operator **-h**:

$$\frac{g\colon c\textbf{-h}d \;\; h\colon d}{\lambda_i\, \exists_j\, (g\, j\, i) \wedge (h\, j)\colon c} \qquad \frac{g\colon c\textbf{-b}d}{\lambda_k\, (f_{c\textbf{-b}d}\, g\, \{k\})\colon c\textbf{-h}d} \tag{Ha–b}$$

For example:

$$\dfrac{\dfrac{\dfrac{\text{peel}}{\textbf{V-aN-bN}}}{\textbf{V-aN-hN}}\ \text{Hb} \quad \dfrac{\text{and} \qquad \dfrac{\text{eat}}{\textbf{V-aN-bN}}}{\textbf{(V-aN-hN)-c(V-aN-hN)-d(V-aN-hN)} \quad \textbf{V-aN-hN}}\ \text{Hb}}{\textbf{(V-aN-hN)-c(V-aN-hN)}}\ \text{Cc} \quad \dfrac{\text{shrimp}}{\textbf{N}}}{\textbf{V-aN-hN}}\ \text{Ca}}{\textbf{V-aN}}\ \text{Ha}$$

9. Inference rule to change category keeping the meaning function as is.

$$\frac{g\colon c\psi}{g\colon c\chi} \tag{T}$$

An example derivation of the noun phrase *the person who officials say stole millions*, exemplifying F, G, and R rules, is shown in Figure 4.1; and an example derivation of the noun phrase *creditors, investors and employees of the company*, exemplifying C, E, and H rules, is shown in Figure 3.3. After all lambda expressions are applied to arguments in a derivation, each word is associated with the variable of an existential quantifier. These existentially quantified variables can then be uniquely identified using numerical indices of words, and the numbered functions in lambda expressions $(v\, i) = j$ are interpreted as dependency relations assigning the v^{th} argument of i to be j.

This system has the attractive property that the same syntactic constraints can be assigned the same category in every context. This property is not shared by most categorial grammars: e.g. post-nominal and post-copular prepositional phrases often have different categories.

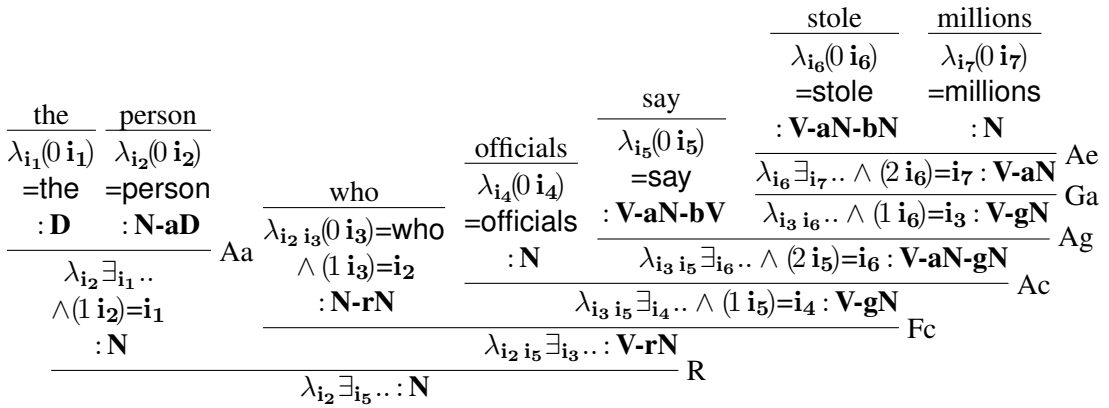


Figure 3.2: Example categorization of the noun phrase *the person who officials say stole millions.* This derivation yields the following lexical relations: $(0\,\mathbf{i_1})$=the, $(0\,\mathbf{i_2})$= person, $(0\,\mathbf{i_3})$=who, $(0\,\mathbf{i_4})$=officials, $(0\,\mathbf{i_5})$=say, $(0\,\mathbf{i_6})$=stole, $(0\,\mathbf{i_7})$=millions, and the following argument relations: $(1\,\mathbf{i_2})$=$\mathbf{i_1}$, $(1\,\mathbf{i_3})$=$\mathbf{i_2}$, $(1\,\mathbf{i_5})$=$\mathbf{i_4}$, $(2\,\mathbf{i_5})$=$\mathbf{i_6}$, $(1\,\mathbf{i_6})$=$\mathbf{i_3}$, $(2\,\mathbf{i_6})$=$\mathbf{i_7}$. The semantic dependency relations for this sentence are represented graphically in Figure 3.1.

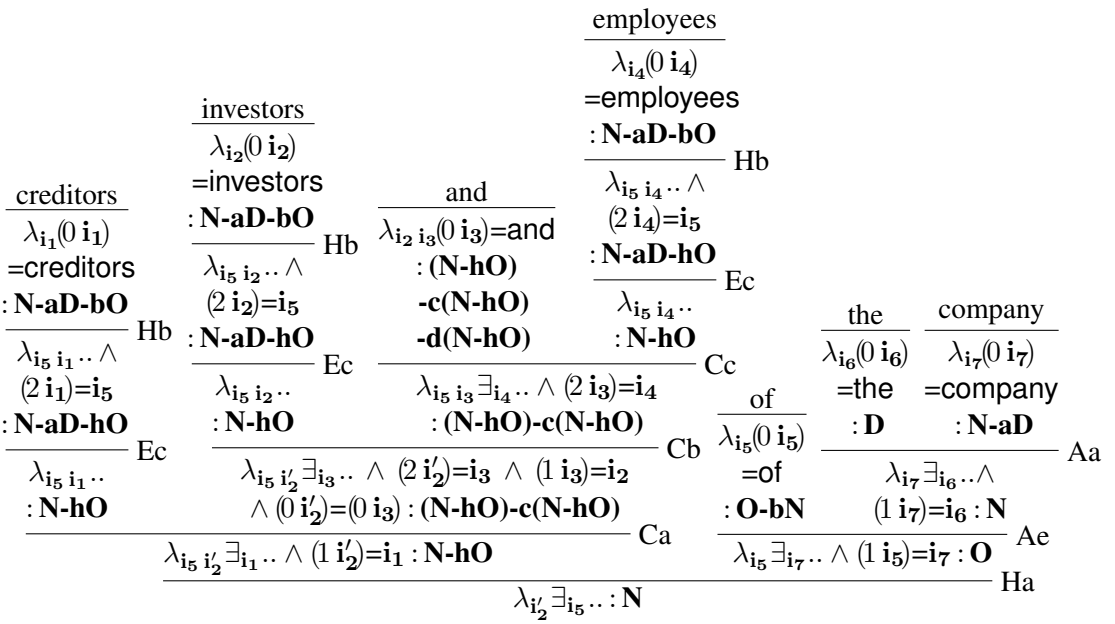


Figure 3.3: Example categorization of the noun phrase *creditors investors and employees of the company.* This derivation yields the following lexical relations: $(0\,\mathbf{i_1})$= creditors, $(0\,\mathbf{i_2})$=investors, $(0\,\mathbf{i'_2})$=$(0\,\mathbf{i_3})$=and, $(0\,\mathbf{i_4})$=employees, $(0\,\mathbf{i_5})$=of, $(0\,\mathbf{i_6})$ =the, $(0\,\mathbf{i_7})$=company, and the following argument relations: $(2\,\mathbf{i_1})$=$\mathbf{i_5}$, $(2\,\mathbf{i_2})$=$\mathbf{i_5}$, $(2\,\mathbf{i_4})$=$\mathbf{i_5}$, $(1\,\mathbf{i'_2})$=$\mathbf{i_1}$, $(2\,\mathbf{i'_2})$=$\mathbf{i_3}$, $(1\,\mathbf{i_3})$=$\mathbf{i_2}$, $(2\,\mathbf{i_3})$=$\mathbf{i_4}$, $(1\,\mathbf{i_5})$=$\mathbf{i_7}$, $(1\,\mathbf{i_7})$=$\mathbf{i_6}$.

Chapter 4

Automatically Reannotating TreeBank

This chapter describes the reannotation process to convert Penn Treebank (PTB) into our GCG annotation. It will be organized to show not only a complete reannotation system but also the linguistic motivation behind each construct of widely accepted grammatical phenomena such as noun phrase, verb phrase, relative clause, interrogative clause, topicalizations, right node raising, etc.

TreeBank is one of the most well-known English annotated corpora for research in natural language processing since early 1990's. However, the annotation scheme is not easy for a PCFG to learn as it is:

- Non-local: It has a variety of useful "null" elements and "trace" information to represent non-local dependencies, specifically, to signal where the filler and the gap parts are, but there is no connecting means to go from one to the other, so it is not suitable for a Probabilistic Context Free Grammar (PCFG) learner that usually relies on local connecting information between a node and its immediate children.

- Too flat: The flatness in PTB can be seen at the sentence level, as well as various constituent types of **VP**s and **NP**s. For verb phrases, a modal or auxiliary, if present, introduces a new **VP** level; within that the possibly complex modifiers and arguments appear at the same level as sisters of the main verb. For noun phrases, any complex noun modifiers appear at the same level of the **NP** they modify. Compound nouns are lacking any internal structure. This flatness is a disadvantage for a typical probabilistic learner as evidence for large flat rules will be very sparse.

The reannotation system described in this research defines its target grammar in terms of a set of reannotation rules. These reannotation rules work within a script that traverses each bracketed sentence in a corpus by selecting each pair of matching brackets from the top of the tree to the bottom, and from left to right, then running a sed-like pattern substitution rule on each selection (see Figure 4.1). Such rules can implement local syntactic transformations, as well as certain non-local transformations like adding gap arguments to constituents containing a particular trace marker, for example. Reannotation of the categorial grammar evaluated in this paper requires about 175 such rules. These rules are modular and can be reused or modified to experiment with different syntactic analyses.

In order to make the trace and function labels in the Treebank accessible to a PCFG-based parser and latent variable annotator, they must be incorporated into the syntactic categories of each tree node and propagated from filler to gap constituents, creating a categorial grammar with local associations between parents and immediate children. First all trace annotations for interrogatives, relative clauses, and topicalizations are transformed into gap arguments: **-gN** (for noun phrase gaps), **-g{R-aN}** (for adverbial phrase gaps), and **-gS** (for e.g. topicalized sentential gaps), which follow the category label for each constituent. Similar transformations localize it-clefts (*it seems that...*), tough constructions (*tough to cut*), parentheticals (*he/she said*), and certain types of inversion (*'it rained,' she said*), also using the gap operator **-g**. This is similar to the treatment of filler-gap constructions used in HPSG (Pollard and Sag, 1994).

Then, specifiers of head projections are annotated as initial arguments, e.g. **-aN** for nominal subjects, and complements of head projections are annotated as final arguments: e.g. **-bN** for transitive verbs, prepositions, and certain adjectives, **-bV** for sentential complements, **-bN-bN** for ditransitive verbs, etc. The **-h** operator is then used to propagate directional dependencies in right node raising (*they peeled and ate shrimp*). These are similar to the 'subcat' feature in HPSG, or to the left and right slash in categorial grammar accounts of specification, complementation and right node raising.

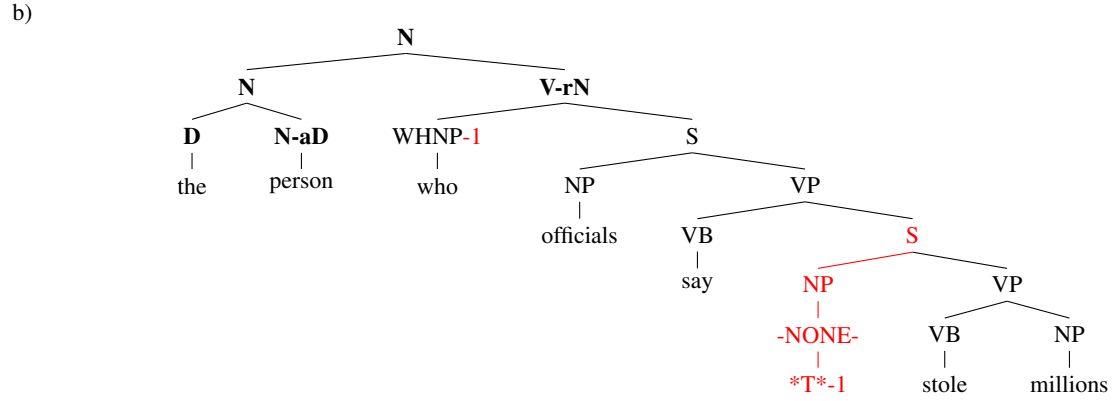
Then, interrogative and relative pronouns (e.g. *what* and *which*) are distinguished with **-iN** and **-rN** arguments, respectively, in order to regularize typical contexts for filler-gap traces.

Conjuncts are assigned **-c** and **-d** arguments to distinguish composition functions for conjuncts from composition functions for ordinary arguments. This distinction makes it possible for ordinary arguments to be shared among conjuncts.

The transform rules are defined as recursive rewrites that progress down the Treebank trees,

a) s/\^(V-rN) <(WHNP)(-[0-9]+)([^ >]*)> (.*[-NONE- *T*\3].*)\^/\^\1 <N-rN\4> <V-gN\3 \5>\^/;

b)



c)

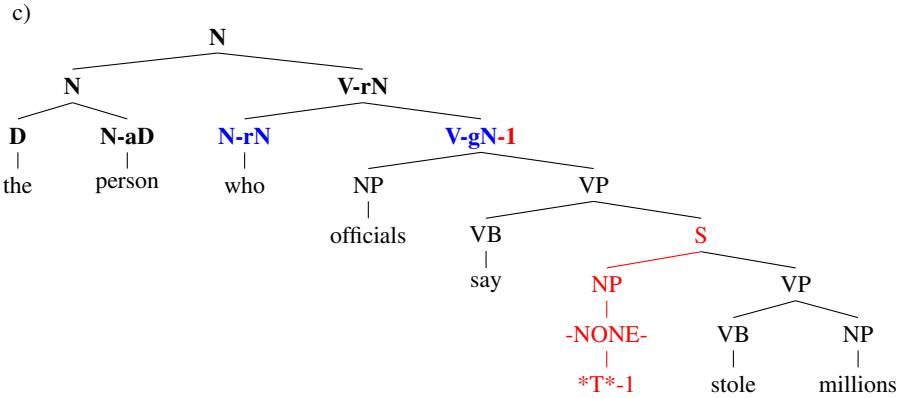


Figure 4.1: Sample sed-like reannotation rule introducing a gap tag at the top of a relative clause (a), and an application of this rule to the movement-based notation in the Penn Treebank (b) to produce a binary-branching categorial grammar derivation using gap arguments (c). Rules are applied to every constituent from the top of the tree down, using parentheses to delimit constituents above the current constituent, carets to delimit the current constituent, angle brackets to delimit child constituents, and square brackets to delimit constituents below children. Delimiters are then updated at every iteration.

propagating **-a**, **-b**, **-c**, **-d**, **-g**, **-h**, **-i**, and **-r** arguments as they go. Figure 4.1 shows one step in a reannotation of the noun phrase *the person who officials say stole millions.* This set of recursive rules obtains about 94% coverage of the training set, consisting of sections 2–21 of the Penn Treebank. Trees not completely transformed by these rules are excluded from training.

In many cases the conversion to this categorial grammar replaces Treebank category labels with more general specifications. For example, in most contexts, adjective phrases, prepositional phrases, and progressive and passive verb phrases are replaced with a predicative category **A-aN**. This allows an HPSG-, GPSG- or CCG-like treatment of conjunction of any of these kinds of phrases with any other, following Sag et al. (1985), Gazdar et al. (1985), and Komagata (2002). This generalized predicative category is also used for noun phrase modifiers, and for noun phrases following the copular *be*. This annotation is compatible with additional specification of categories for adjective phrases (following words like *become*), or gerund phrases (following words like *start*), but these contexts cannot be reliably identified by the current reannotation.

In other cases, the conversion to categorial grammar distinguishes categories that are conflated in the Treebank. For example, the Treebank 'SBAR' category is distinguished into adjectival relative clauses **V-rN**, embedded questions **V-iN**, embedded inflected sentences **C**, embedded infinitival sentences **E**, nominal clauses **N**, adjectival modifier phrases **A-aN**, and adverbial modifier phrases **R-aN** (e.g. *because* ...). Treebank 'S' categories are similarly distinguished into inflected sentences **V**, infinitival sentences **I**, base form sentences **B**, adjectival sentences (small clauses) **A**, and participial sentences **L**.[1]

Also, like other categorial grammars, this transform leaves trees in binary branching (Chomsky normal) form. For matching criteria, occasionally, some of these rules may look beyond the current constituent to the right siblings, or down the subtree to the lexical nodes, but normally they only rely on the category of the parent of the constituent (GCG category) and the categories of the immediate children (PTB categories). Section 4.10 shows preprocessing steps done on the PTB trees to convert the parent from a PTB category to a GCG category as the initial step before progressing down the tree to applying these rules to transform the entire tree into GCG format.

The following sections in this chapter will describe in more details all the reannotation rules

[1] It is important to note that these rules allow the word *that* to be treated as a relativizer rather than a relative pronoun, as it is in other categorial grammars, such as CCG.

of the system. To aim for completeness in explaining these rules, they are associated with the inference rules introduced in Chapter 3. These inference rules were classified into 9 groups and labeled by A (argument attachment), M (modifier attachment), C (conjunction attachment), G (gap attachment), F (filler attachment), R (relative pronoun attachment), E (argument elision), H (right node raising), and T (type changing). A small letter can be used in conjunction with these capital letters to label individual rules if the group has more than one rule. As a mirror to this setup, the reannotation rules will be classified and described accordingly in the following sections. The reannotation rules that do not belong to any of these groups will be described in the miscellaneous section. For the most part, a reannotation rule is a reversed process of its inference rule counterpart. For example, if inference rules Aa–d were about applying functors of category c**-a**d to initial arguments of category d then their reannotation rule counterpart will show how to inspect some node of category c to see if it can be rewritten to have a left child of category d and a right child of category c**-a**d. Another way to think about this reversed process is that the reannotation goes top down to rewrite the corpus into the desire format that can be learnt and parsed in a bottom up fashion (e.g. by a CKY parser).

4.1 Reannotation rules for initial and final argument attachment (-a/-b)

The reannotation rules for initial and final arguments together account for about one-third of the entire reannotation rule set. The wild card term at the end of the parent category, usually denoted as **[^]*** or a regular expression containing a non-empty string, is the possible non-local attachment ψ that can tag along with the category and carry on to the left or the right child in the re-write. These reannotation rules mainly match against the GCG category of the parent node, together with the PTB categories of the immediate children nodes, to decide the appropriate new GCG categories for the left and the right child of the parent. Some rules occasionally look at the categories of the siblings of the parent or descendant nodes beyond the immediate children all the way to the lexical item to determine whether they contain helpful information that is not available at the parent or immediate children. This process continue to push the boundary of GCG/PTB categories downward to the leaf nodes when the entire tree was reannotated into GCG format.

Following are the two sub-sections to describe the rules for initial argument attachment

(Aa–d) and the ones for final argument attachment (Ae–h). Since English is mostly a head-initial language, we are more likely see final arguments than initial arguments. As a matter of fact, the number of final argument rules is about five times more than that of initial arguments.

4.1.1 Reannotation rules for initial argument attachment (-a)

We have about a dozen rules for initial argument attachment. Common cases for the initial argument rules are the one used to split the initial noun phrase (the possessor) of a possessive *'s* construct; split the initial specifier of measurement from an adjectival or adverbial phrase; split the subject noun phrase as initial argument of the main verb phrase; or split the initial *wh-* and *non-wh* determiner as an initial argument to the rest of a noun phrase.

Figure 4.2 shows an example of a rule to reannotate the possessive construct. If the parent category has been determined to be a determiner **D** or a noun phrase **N** having no initial argument, and the rightmost child has a **POS**, the PTB annotation for a possessive, then this rule will create a new intermediate left child **N** as an initial argument for the possessor which is the head and receive a new category **D-aN**. The example in this figure has **D** as the category of the parent, but other examples like *the lab 's* can have an **N** category at the parent. The idea of this rule is to specify the possessor as the head of a phrase, and whatever comes before it as its initial argument.

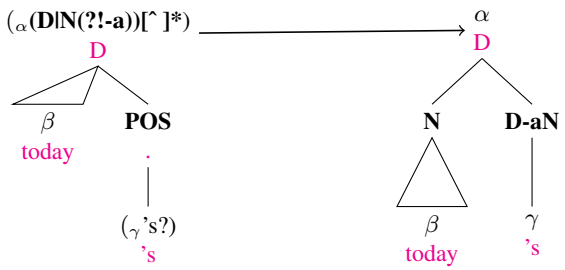


Figure 4.2: Branch off final possessive *'s*. This example has α=D and γ='s.

Figure 4.3 is for a rule to break up the initial specifier for measurement from an adjectival or adverbial phrase. We use **-x** to denote intransitive and **e** for the expletive. The **-[cp]** is about coordination conjunction and punctuation that will be explained in Section 4.3. This rule is designed to eliminate any of those scenarios. If the parent is an adverbial or adjectival phrase that has its left-most child annotated by the PTB's **NP**, then that **NP** will be re-written into the

GCG category **N** which is an initial argument of a new right child node created to group all the rest of the children. This right child node takes the category α of the parent appended by a **-aN**.

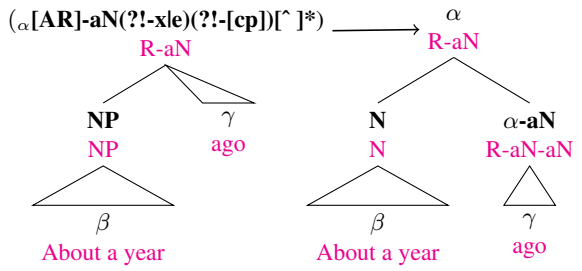


Figure 4.3: Branch off initial specifier **N** measure. This example has α=R-aN.

Figure 4.4 and Figure 4.5 are the two rules for reannotating the subject of a verb phrase into an initial argument. If the category of the parent was determined to be a verbal (**V**: finite verbal, **I**: infinitive verbal, **B**: base-form verbal, **L**: participial verbal, **A**: adjectival/predicative, **G**: gerund) and the left-most immediate child is a subjective nominal phrase as denoted in PTB's **NP(-SBJ)?** (Figure 4.4) or **[^]*-(NOM|SBJ)[^]*** (Figure 4.5), then this subjective nominal phrase will be re-written into a GCG category **N** which is an initial argument for a newly created right child node that captures the rest of the children. This newly created right child node will get the verbal part of the category of the parent α, followed by a **-aN** to denote its initial argument, and the last part β of the parent as a mean to tag along non-local argument ψ.

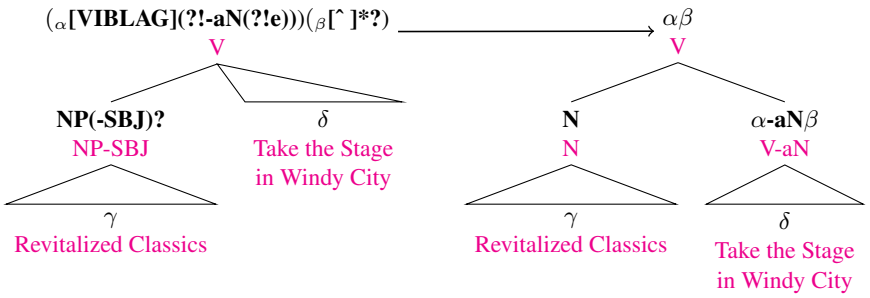


Figure 4.4: **[VIBLAG]** sentence: branch off initial **N** subject. This example has α=V and β=$\varnothing$.

The next topic for initial argument attachment is about making the determiner an initial argument for a noun phrase. These rules are illustrated in Figure 4.6, Figure 4.7, and Figure 4.8.

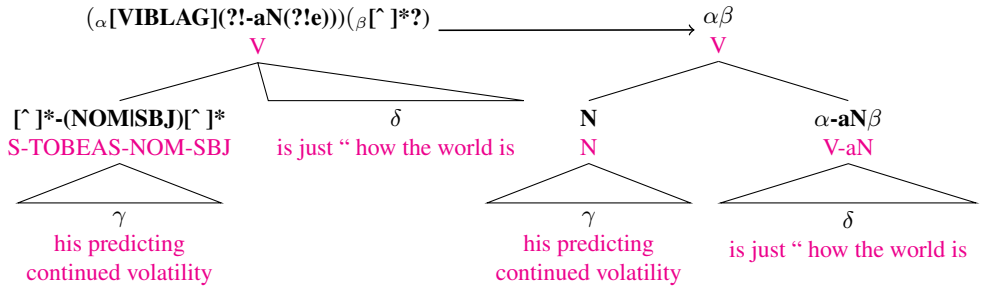


Figure 4.5: **[VIBLAG]** sentence: branch off initial **N** subject. This example has α=V and β=$\varnothing$.

The things to note about these rules are:

- The parent node must be recognized as a noun phrase having no determiner as an initial argument yet. This is denoted by **N(?!-aD)**.
- If the determiner has a PTB category of **DT** or something that covers a **POS**, but not a wh-category, then any non-local attachment β recognized on the parent node is passed on to the right child which is the head of the noun phrase. This is shown in Figure 4.6 and Figure 4.7.

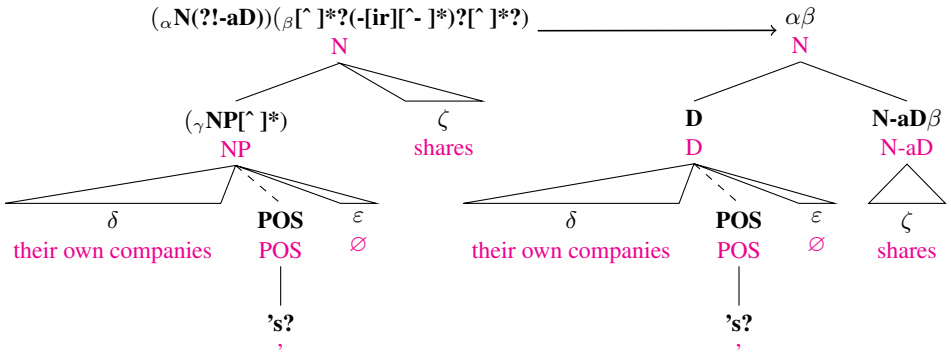


Figure 4.6: Branch off initial determiner (non-wh). This example has α=N and β=$\varnothing$.

- If the determiner has a wh-category which is the category of the English wh-words like *what, which, whose, who, whom, where, when, how, why*, then the non-local attachment recognized on the parent has to be split up when passing down to the left and the right

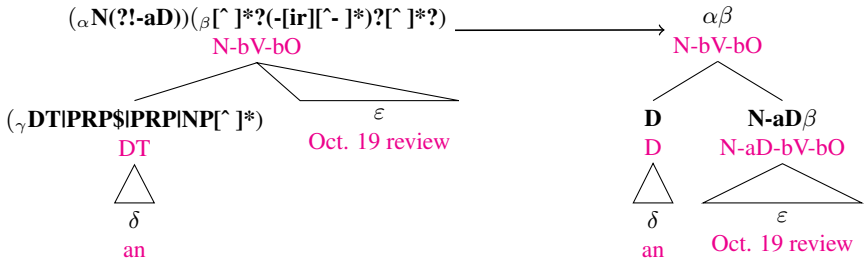


Figure 4.7: Branch off initial determiner (non-wh). If γ=**NP[^]*** then it must have a descendant **(POS 's)** or **(POS ')** in δ. This example has α=N and β=-bV-bO.

child. The left (determiner) will get **-i** or **-r** as it's more likely to be an interrogative or relative. The right (head noun phrase) will get other non-local attachments such as **-g** or **-h**. This is shown in Figure 4.8.

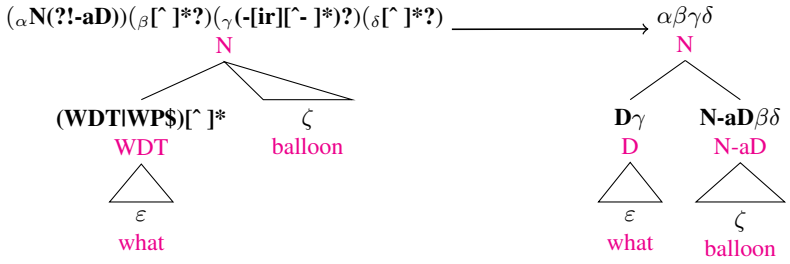


Figure 4.8: Branch off initial determiner (wh). This example has α=N, β=γ=δ=$\varnothing$.

4.1.2 Reannotation rules for final argument attachment (-b)

We need about 50 rules to reannotate the final argument attachments of this corpus. Common cases for the final argument rules are the one used to split the object part as a final argument of the main verb of the object verb phrase in one of the most popular types of SVO sentences (subject-verb-object). Also common final argument attachment can be found in questions where the verb ordering is swapped; a number of embedded and complementizer constructions where the last verb phrase is a final argument of the complementizer; or various types of modifiers having a preposition to be the head and the rest of the phrase is a final argument. One thing to note about the final argument attachment as apposed to the initial argument attachment is that

it is more common to see a head with multiple final arguments attached to it (e.g. **-bN-bN** for ditransitive verbs). Below is an account of all the rules we need to reannotate the final argument attachments. These rules are classified based on the grammatical type of the parent node that are: verbal phrase, modifier phrase, interrogative phrase, complementized phrase, embedded phrase, genitive phrase, and noun phrase.

Final argument attachment for verbal phrase

We need about two dozen rules to reannotate the final argument attachment for verbal phrases. The parent verbal phrase is annotated with a **[VIBLARG]** or any subset of those verbal categories. The **A** and the **R** here are classified as verbal because they usually have a PTB's **VP** as their right-most child. This is different from the **[AR]** parent that would be classified as a modifier as shown in the next section. The main idea of these rules is to spin off the right-most child (usually of PTB's category **VP**) into one of the GCG's verbal categories depending on the context of the other children of the parent node. If the right-most child is not of PTB's **VP**, then depend on what it is and what their children are, these rules will try to re-write it into the appropriate GCG's category. Other usage of these rules is to reannotate the particle as a final argument of the verb in a particle verb phrase (e.g. *carry on*, *make up*, etc.) Based on this idea, these rules can be further classified into what GCG category they decided for the final argument they generate. The possibilities are: base form verbal phrase **B**, participial verbal phrase **L**, adjectival/predicative phrase **A**, sentential phrase **S**, genitive phrase **O**, gerund phrase **G**, noun phrase **N**, infinitive verbal phrase **I**, finite verbal phrase **V**, embedded verbal phrase **E**, complementized verbal phrase **C**, or any specific phrase type depending on the category of the right-most child of the parent node. More specifically:

- Final argument categorized as base form verbal phrase **B**: If the right-most child has the PTB category of **VP**, and some other child on the left is either of category **TO** (for the word *to*), **MD** (for modals), or any other verbal category (e.g. for auxiliary verbs), then the right-most child should be reannotated as a base form **B** final argument of a newly created child node that covers the rest of the children. This is illustrated in the rule at Figure 4.9. The three examples in this figure show three different types of *to*, modal, and auxiliary verb of the left child node.

- Final argument categorized as participial verbal phrase **L**: Similar to the rules for base

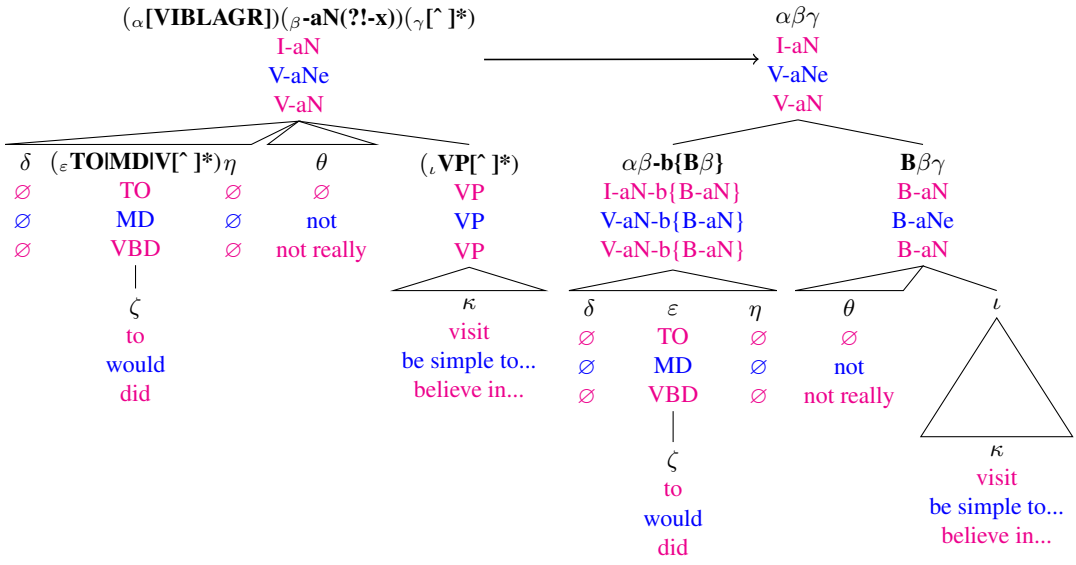


Figure 4.9: Branch off final **VP** as argument **B-aN**. If ε=**V[^]*** then ζ must be either *do*, *does*, or *did*. If $\theta \neq \varnothing$ then its top-left-most branch must be a **(RB .*)** e.g. **(RB not)**. The three examples show different types of possible categories of the left child.

form verbal phrase above, if some of the sibling of the right-most child is an auxiliary *have* then it is a good signal that this is a past participial phrase. This means the right-most verbal child will be reannotated as a GCG's category of **L** which is a final argument for a newly created child that encapsulates the rest of the children. This rule is illustrated in Figure 4.10. A more specific version of this rule if the one shown in Figure 4.11 when there is a noun phrase sitting between the auxiliary verb and the main verb. In this case, the newly created left child covers only the auxiliary and the rest of the children (including the noun phrase and the participial phrase) will be grouped under another newly created right child labeled with an **L**.

Another common case to use **L** is in the passive voice as seen in Figure 4.12. In this rule, the participial is the left-most child of the right-most child of the parent. The parent must also have another child annotated by a PTB's category **VBZ** covering an apostrophe-s *'s*. This rule is more specific on certain participial to minimize the risk of it not being a passive voice if *'s* is in fact a *has*, not *is*.

- Final argument is a particle categorized as **P** followed by a word. If the right-most child

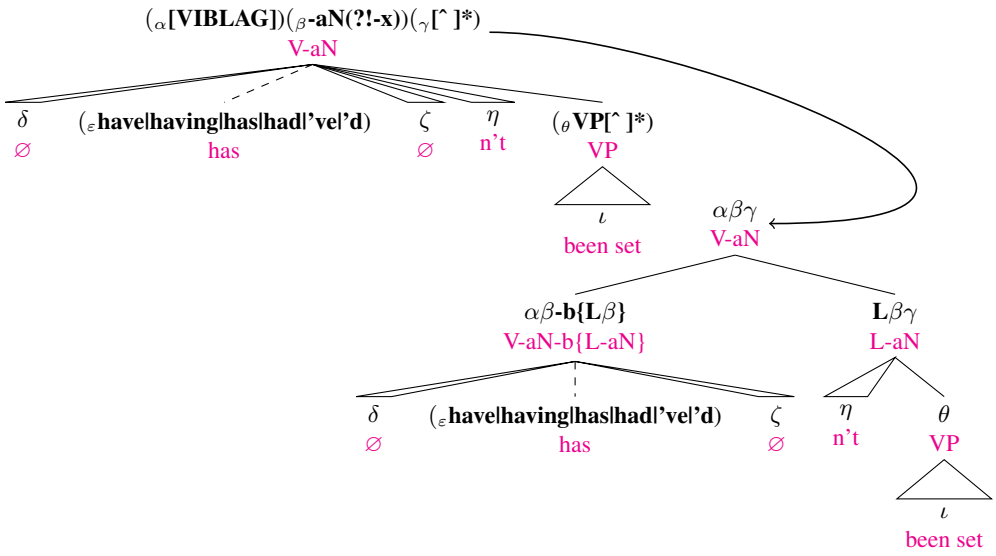


Figure 4.10: Branch off final VP as argument **L-aN** (w. special cases because *'s* is ambiguous between *has* and *is*). If ε=**'d** then its parent must be a **VBD**. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be a **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. This example has α=V, β=-aN, γ=∅.

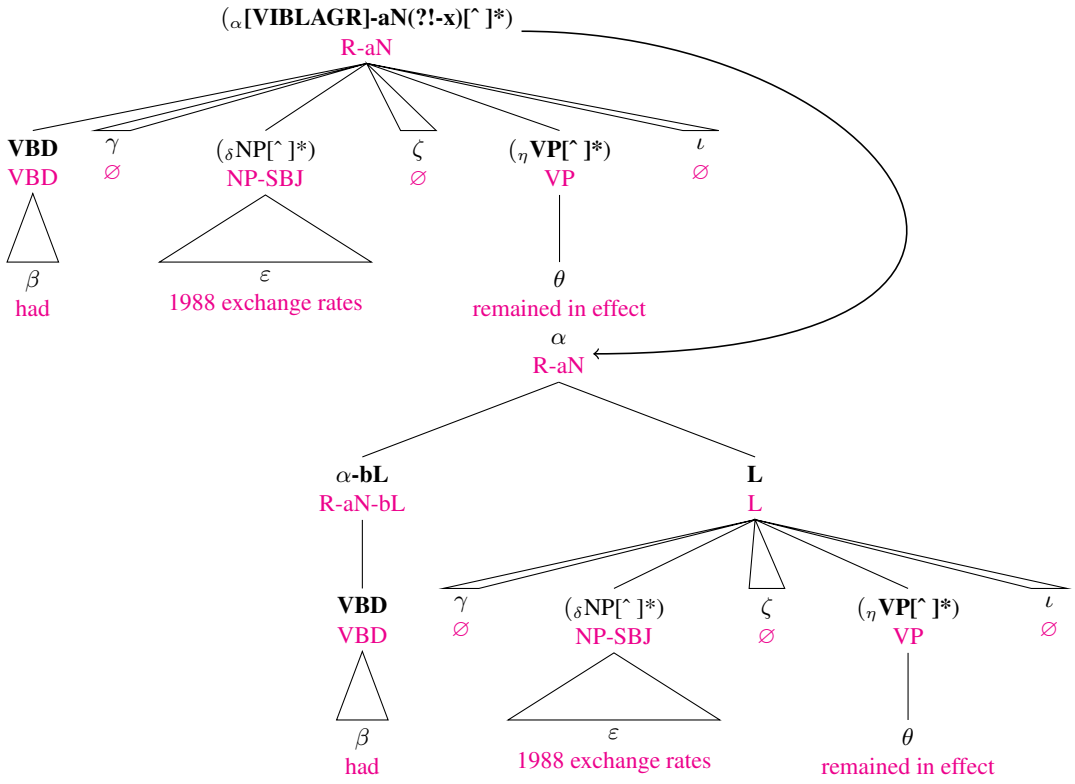


Figure 4.11: Branch off final argument **L** (*had I known* construction).

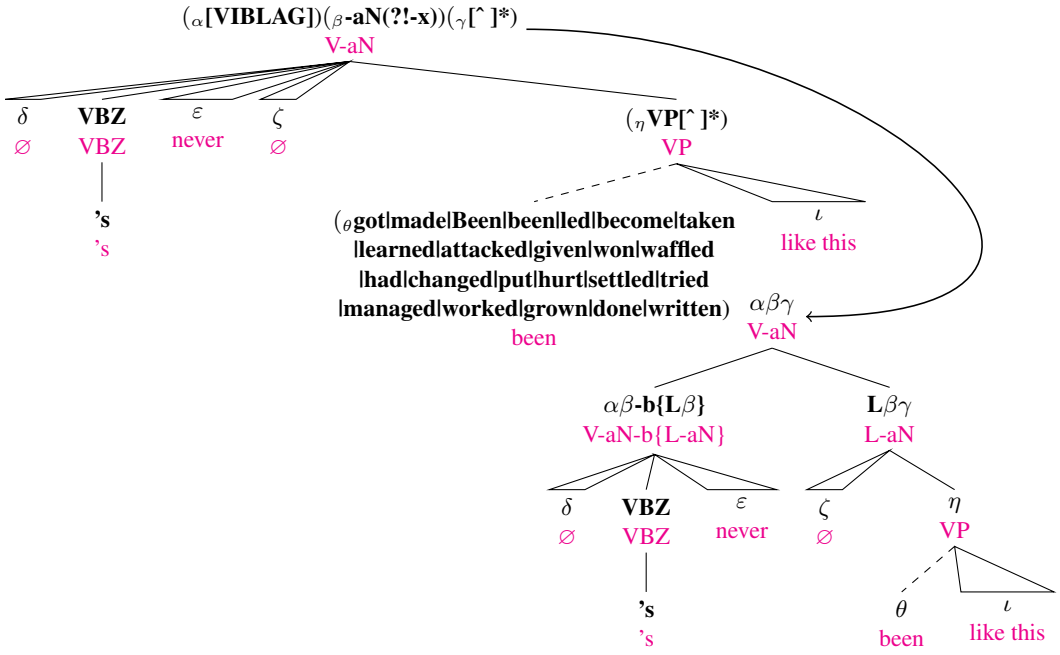


Figure 4.12: Branch off final VP as argument **L-aN**. This example has α=V, β=-aN, γ=$\varnothing$.

was annotated by a PTB's **PRT** (for particle) that has only one child of category **RP** (for adverb), then it is a good indication of a particle verb phrase, e.g. *make up*, *carry out*, *stand by*, *look after*, etc. The **PRT**-**RP** will be collapsed to just one node and reannotated with a **P** category followed by the particle word, e.g. **Pout** for *carry out* as in the example at Figure 4.13

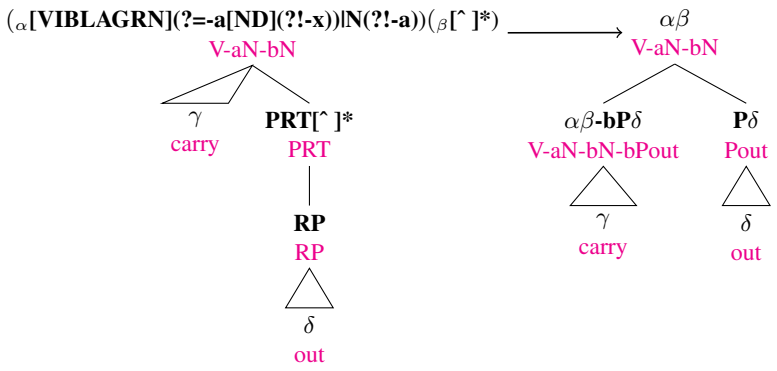


Figure 4.13: Branch off final **PRT** as argument **Pword** particle. This example has α=V and β=-aN-bN.

- Final argument categorized as adjectival/predicative phrase **A**. These rules try to detect and spin off the predicative phrase that usually occurs after the copular *be* as shown in Figure 4.14 and Figure 4.15. If there is enough evidence that the spin-off predicative is also a type of noun phrase then the rule will further makes a unary transform to an **N** category as depicted at Figure 4.15.

  There are adjectival and adverbial predicative phrases following verbs other than the copular, e.g. *ready* in the phrase *remain ready*. This grammatical recognition is shown in the rule at Figure 4.16.

- Final argument categorized as sentential phrase **S**. PTB used **SQ**, **SINV**, and **SBARQ** to denote embedded questions as shown in Figure 4.17 and Figure 4.18. In GCG analysis, these type of embedded sentential phrases would be reannotated into the category **S** as a final argument to the newly created head left child that covers the rest of the children of the parent node.

- Final argument categorized as genitive phrase **O**. Genitive is about the *of* construction.

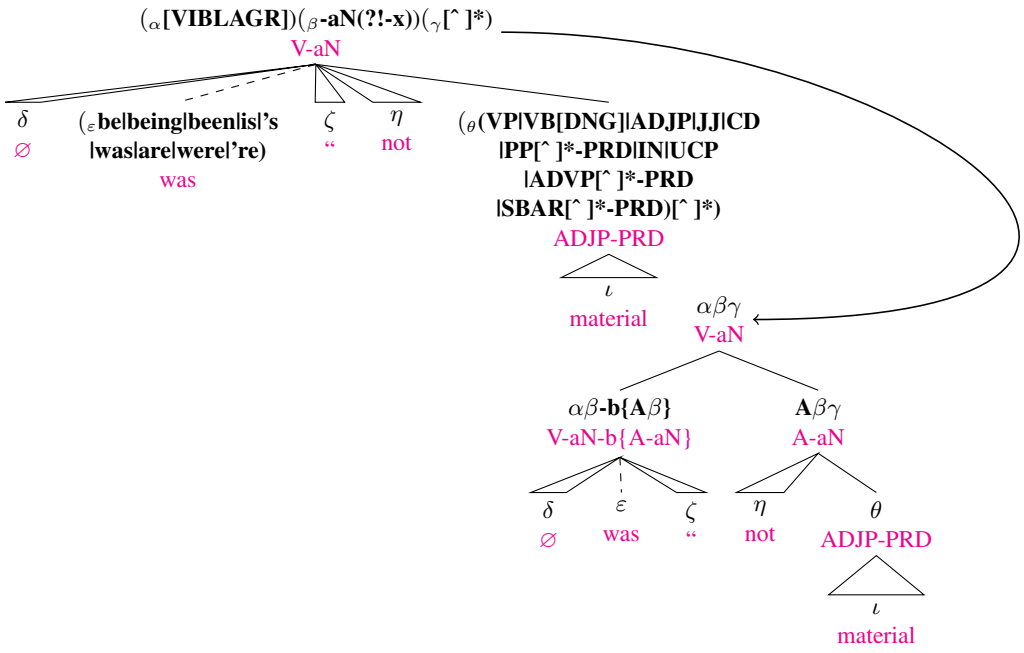


Figure 4.14: Branch off final verbal or predicative adjectival phrase as argument **A-aN** if it occurs after a copular *be*. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If ε**='s** then its immediate parent must be a **VBZ**. If θ=**SBAR[^]*-PRD** then its left-most child must not be a **WH[^]*** or **(IN that)**. This example has α=V, β=-aN, γ=∅.

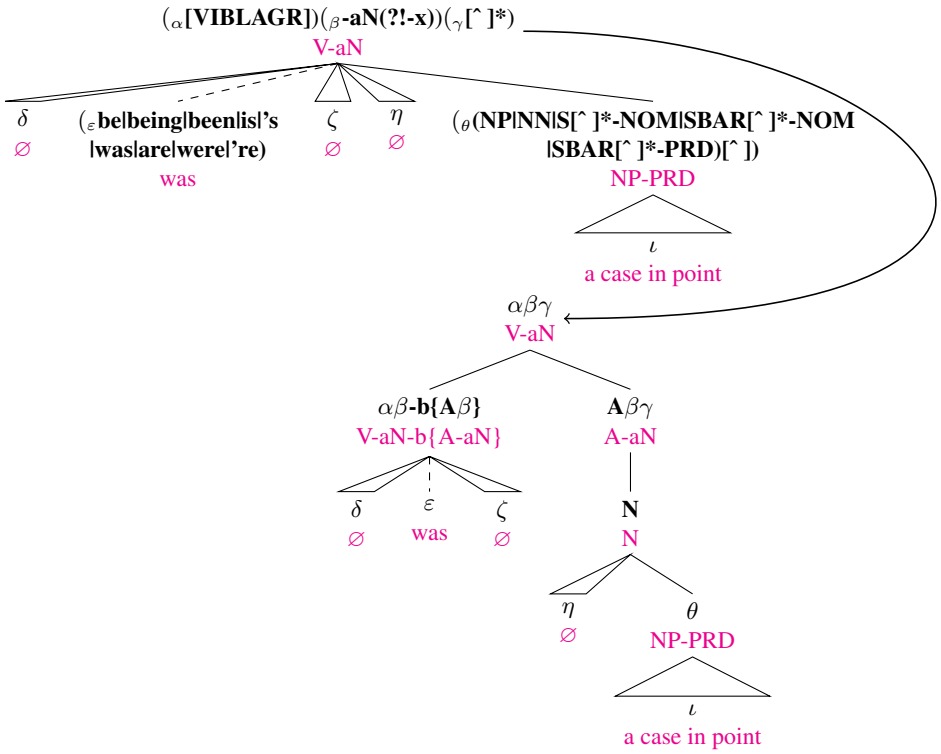


Figure 4.15: Branch off final predicative noun phrase following the copular *be* as argument **A-aN** and then unary transforming to **N**. Top-right-most node in ζ must not be an **RB**. Subtree η, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If ε=**'s** then its immediate parent must be a **VBZ**. This example has α=V, β=-aN, γ=∅.

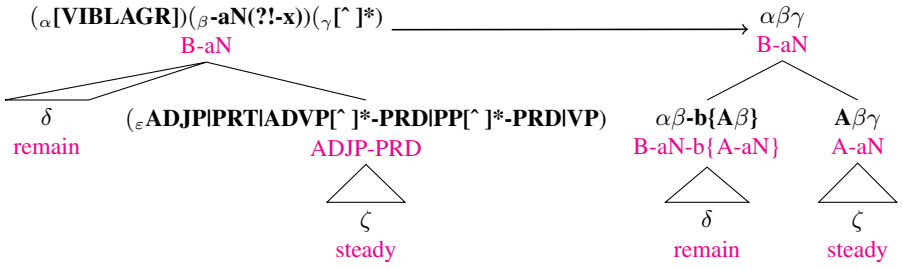


Figure 4.16: Branch off final adjectival or adverbial predicative as argument **A-aN**. If ε=**VP** then its first **VP**-head must be a **VB[NG]**. This means there's no child of category **VB|JJ|MD|TO** between the left-most child and the **VB[NG]**-head. This example has α=B, β=-aN and γ=∅.

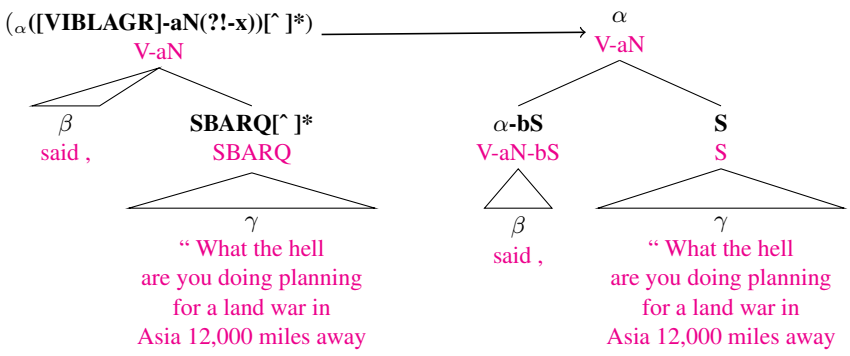


Figure 4.17: Branch off final argument embedded question **S** with quotations.

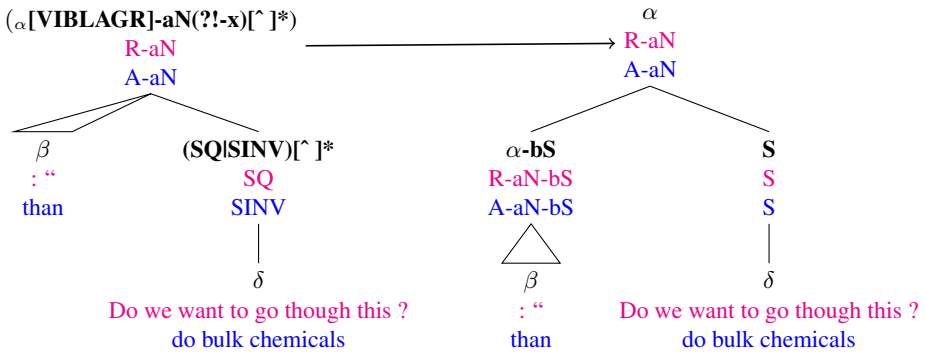


Figure 4.18: Branch off final **SQ|SINV** as argument **S**.

PTB uses category **IN** for the preposition *of*. We detect this **IN** category to spin off a genitive phrase of category **O**. While most of the time PTB grouped the genitive phrase under a **PP** or a **PP-CLR** (closely related) that can be reannotated as shown in Figure 4.20, it sometimes flatten out all the components of the genitive phrase and can be re-written by a rule shown in Figure 4.21. This rule however found only matching when the parent node is a noun phrase **N**.

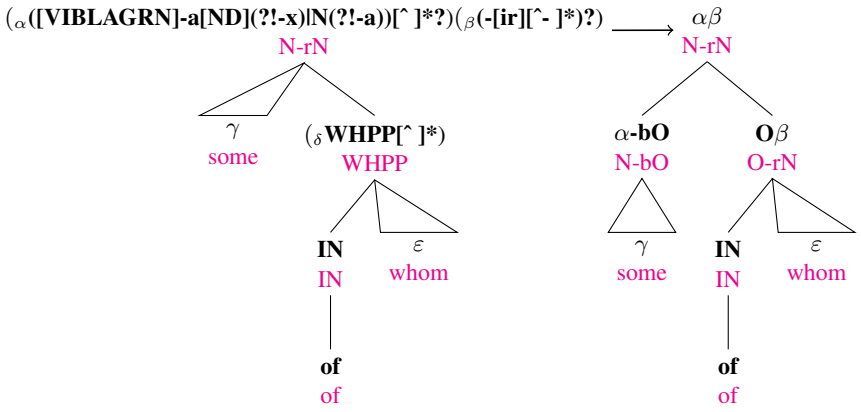


Figure 4.19: Branch off final argument **O**. This example has α=N and β=-rN.

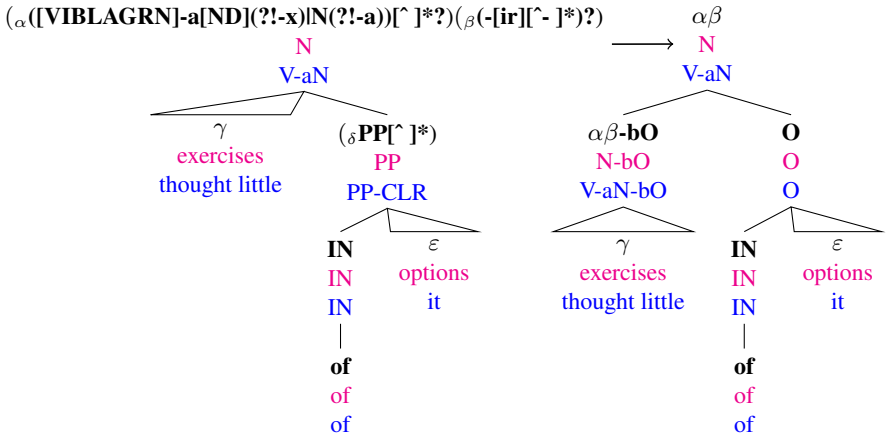


Figure 4.20: Branch off final argument **O**. The magenta example is for parent node of noun phrase category **N**. It has α=N and β=$\varnothing$. The blue example is for parent node of verbal categories. It has α=V-aN and β=$\varnothing$.

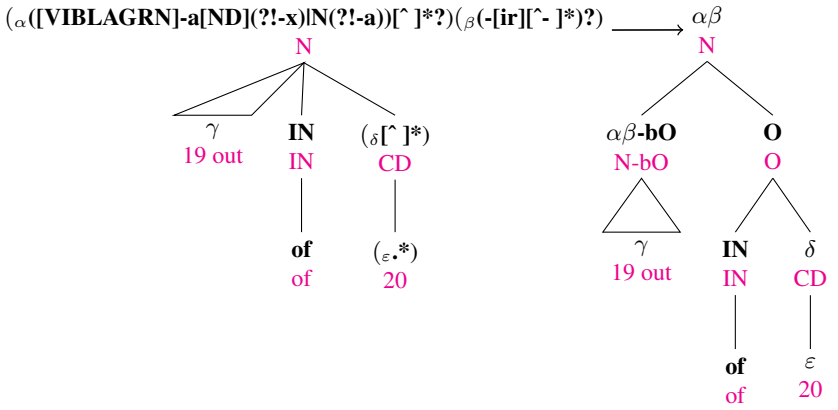


Figure 4.21: Branch off final argument **O**. This example has α=N and β=$\varnothing$.

- Final argument categorized as gerund phrase **G**. While gerunds are a variety of verbal phrase, they often function like a noun phrase. This is why PTB used **S-NOM** to annotate gerunds (extension **-NOM** is for nominal). We reannotate this type of node with a **G-aN** (Figure 4.22) or simply a **G** (Figure 4.23) depending on whether the left-most child of the **S-NOM** node was annotated with an **NP-SBJ** that cover a null element. This **NP-SBJ** covering a null element usually signals a trace back to some subject happening before this node, hence we use **G-aN** instead of just a **G**. Since we use **G-aN** to encode this subject information, we no longer need the **NP-SBJ** and the null element on the resulting tree.

- Final argument categorized as nominal phrase **N**. Spinning off the last noun phrase, the PTB category **NP**, as a final argument of a verb head is the most common case for final argument attachment rules (Figure 4.24). For example, a sentence with a transitive verb normally undergoes the first initial attachment rule **S** $\rightarrow$ **N V-aN** and then is followed by this final argument attachment rule to spin off the last noun phrase as a nominal final argument of the main verb **V-aN** $\rightarrow$ **V-aN-bN N**. Figure 4.25 and Figure 4.26 are two other variations of these nominal final argument attachment rules. Specifically note that the parent node of all these rules has to have a **-aN**. These variations are needed due to the fact that PTB may use annotations other than **NP**, e.g. **-NOM**, to mark functional noun phrase.

- Final argument categorized as **I-aN**. Figure 4.27 shows a very complicated structure to

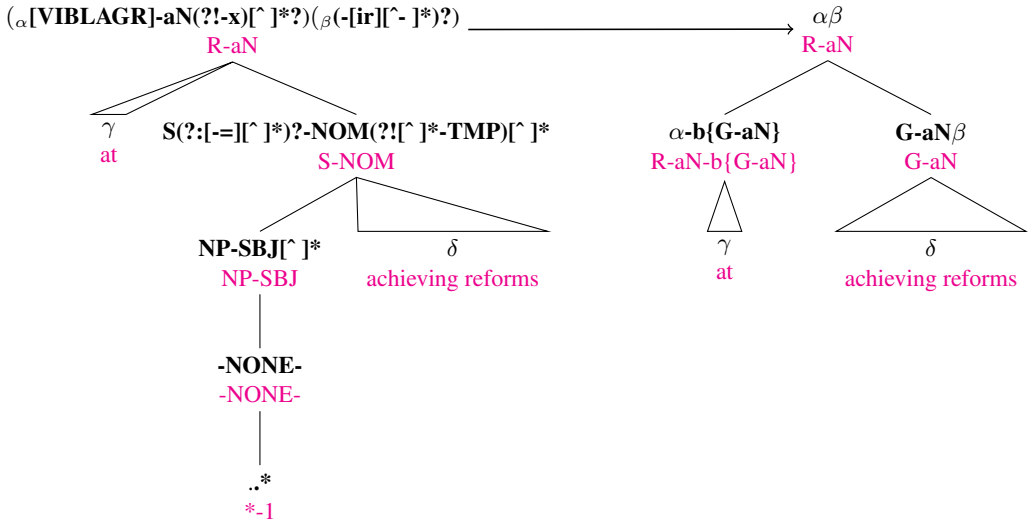


Figure 4.22: Branch off final argument **G-aN**. This example has α=R-aN and β=$\varnothing$.

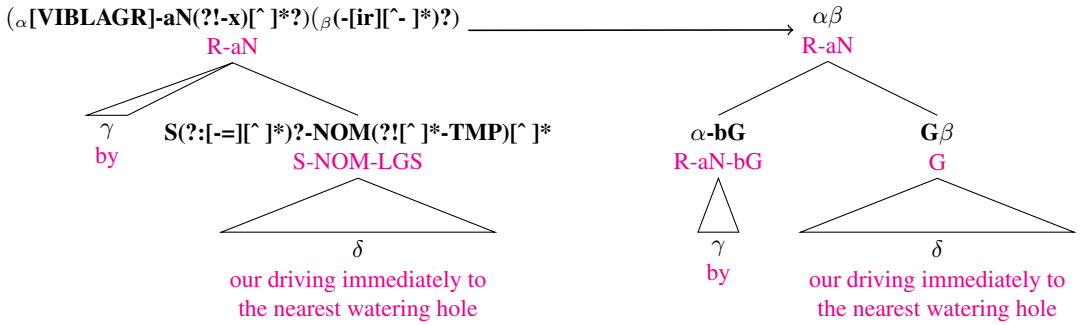


Figure 4.23: Branch off final argument **G**. This example has α=R-aN and β=$\varnothing$.

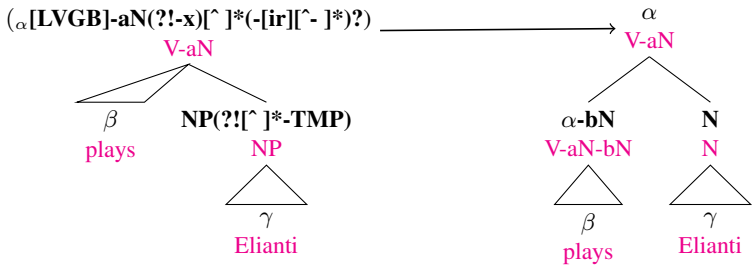


Figure 4.24: Branch off final argument **N**. This example has α=V-aN.

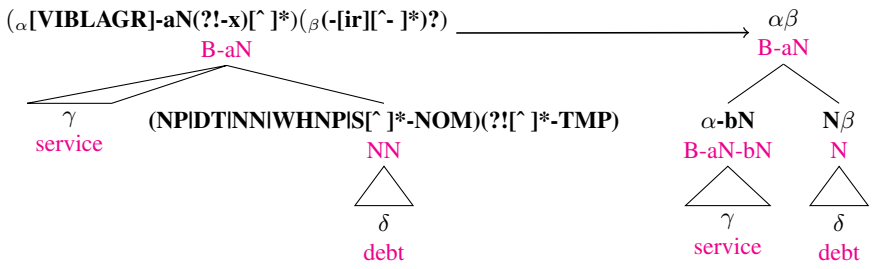


Figure 4.25: Branch off final argument **N**. This example has α=B-aN and β=$\varnothing$.

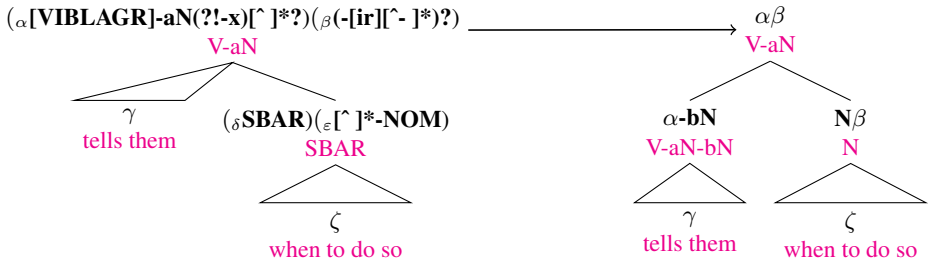


Figure 4.26: Branch off final **SBAR** as argument **N** (nom clause): If ε=$\varnothing$ then the left-most leaf on ζ must be **(when|whether)** as depicted in this example. No node in ζ can be of category ending in **-TMP**. This example has α=V-aN and β=$\varnothing$.

realize a PTB category **SBAR** into an **I-aN**. The matching condition of this rule goes a couple levels down beyond the parent and immediate children. It looks for the keyword *to* of category **TO** and a co-indexation of a trace to a null element. The end result is to remove the trace and the null element, promote the child **S[^]*** of **SBAR** up and give it the category **I-aN**.

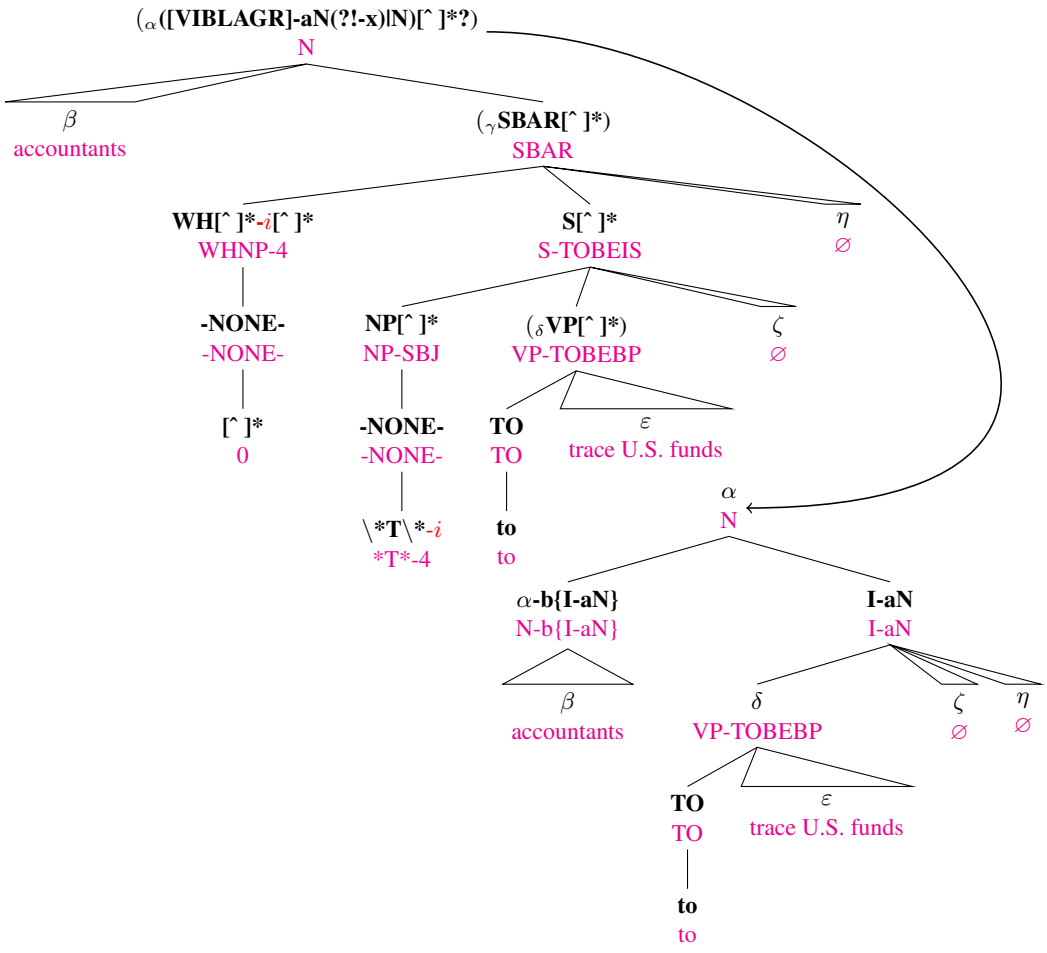


Figure 4.27: Branch off final **SBAR** as argument **I-aN**:

- Final argument categorized as **V-iN**. The PTB **SBAR** used on constructs such as *how to* ... or *whether/if* ... will be reannotated into a **V-iN** as shown in the rule at Figure 4.28. Note one of the two examples of this rule shows its application to the it-cleft construct

where the parent note was annotated with a **-aNe**.

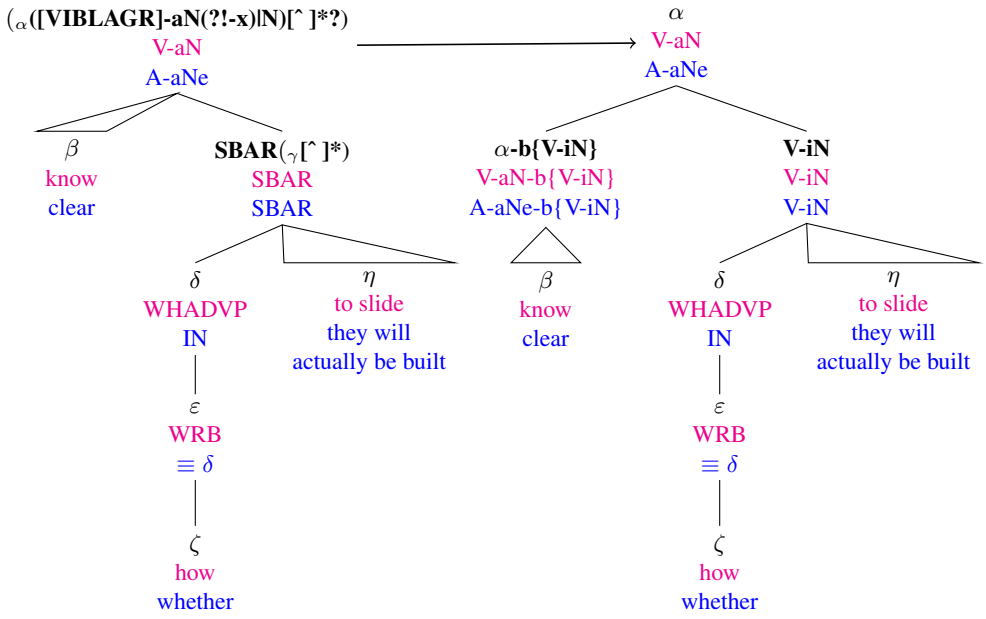


Figure 4.28: Branch off final **SBAR** as argument **V-iN**. This rule matches when either (1) $\gamma \neq$**(-ADV|-TMP)** and ζ=**(whether|if)** or (2) δ=**WH[^]*** and neither ζ=**that** nor ε=**-NONE-**.

- Final argument categorized as an embedded phrase **E**. If *for* is the first word of an **SBAR** phrase, it is more likely that this **SBAR** is an embedded phrase and will be reannotated into an **E** as shown on the rule at Figure 4.29.

- Final argument categorized as a complementized finite phrase **C**. For some **SBAR** that does not match with any rule above, or for the rightmost child of category **INTJ**, they look more like a complementized phrase and we re-write them into a category **C** as shown in the rule at Figure 4.30 and Figure 4.31.

- Final argument categorized as a **V**, **I**, **B**, or **A** depending on the category of the rightmost child. Please refer to the preprocessing rules to see how we propagate the verbal phrasal types from the parent of the tree downward to mark nodes as **-TOBE[VIBA][SP]**. These marks, especially the **[VIBA]** part as denoted by variable γ on Figure 4.32, Figure 4.33, and Figure 4.34 serve as the key criteria to determine the category of the right child final

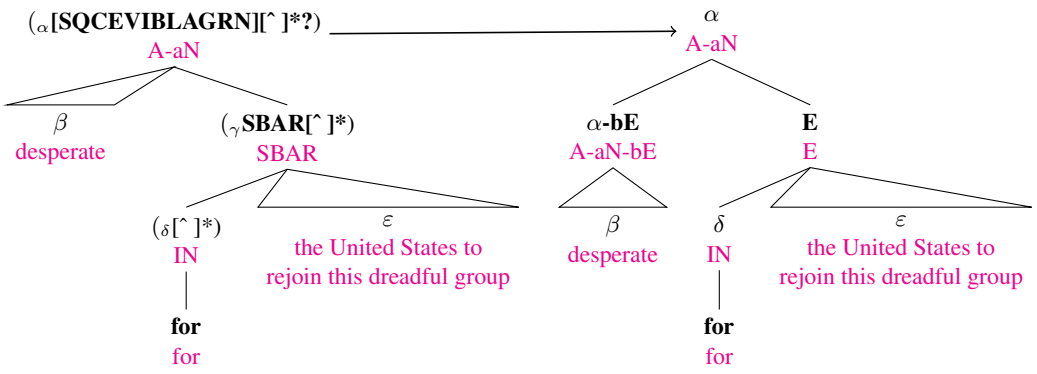


Figure 4.29: Branch off final **SBAR** as embedded argument **E**. The top-left node in β must not be a **[^]*-ADV** and its sibling, if β has more than one top-level node, must not be a **([^]* :)**.

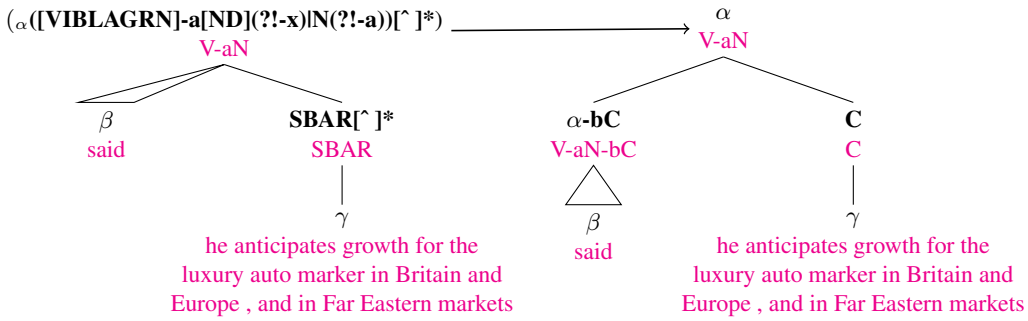


Figure 4.30: Branch off final **SBAR** as complementized argument **C**.

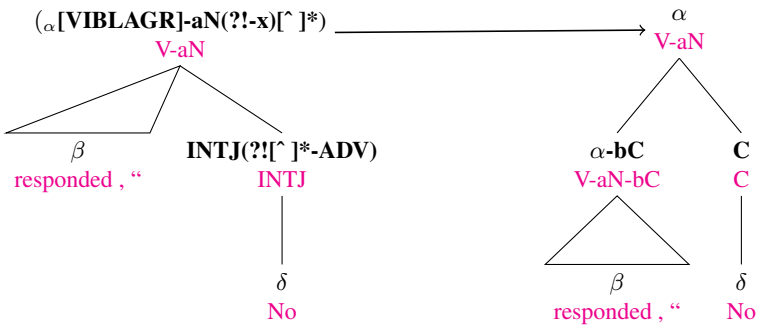


Figure 4.31: Branch off final **INTJ** as argument **C**.

argument. These rules also create a new left child as the head that encapsulates all the rest of the children.

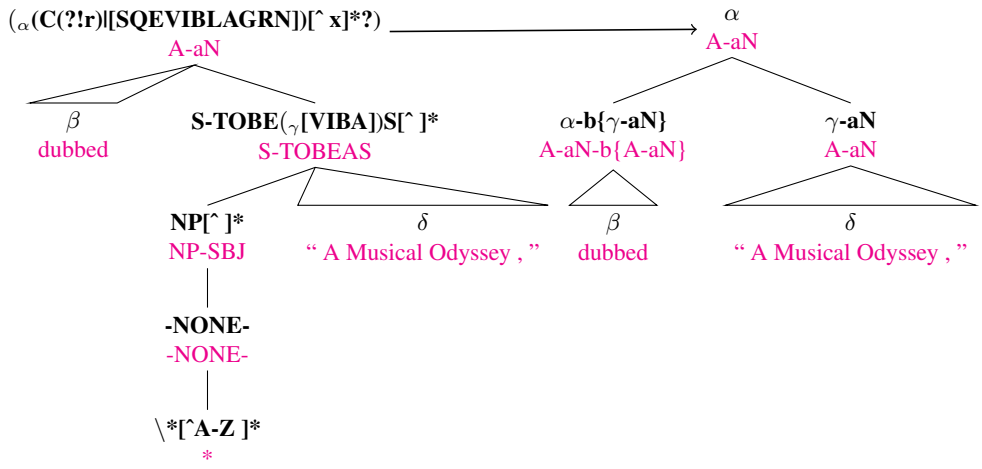


Figure 4.32: Branch off final **S** with empty subject as argument **[VIBA]-aN**.

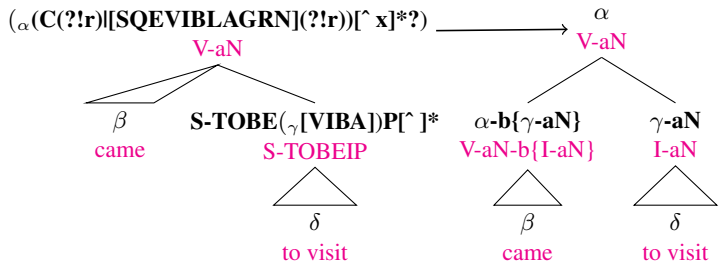


Figure 4.33: Branch off final **S** with empty subject as argument **[VIBA]-aN**. This example has γ=I.

Final argument attachment for modifier phrase

The parent node of rules falling under this classification have to be reannotated as modifiers, e.g. having the category **[AR]-aN** or sometimes an **I-aN** which is an infinitive lacking an initial argument playing the role of a modifier (e.g. the second example in Figure 4.38). The most common category of the final argument of a modifier phrase is **N**, but they could be other categories like **A-aN**, **E-gN**, or **I-aN-gN**. Below is an account of all possible rules to annotate

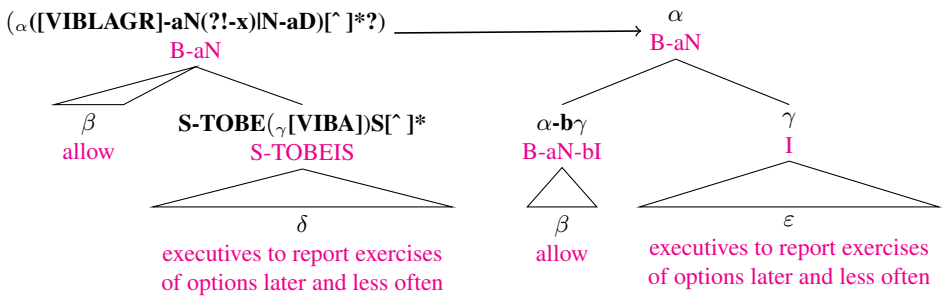


Figure 4.34: Branch off final **S** as modifier **[VIBA]**. This example has γ=I.

final argument attachment for modifier phrase based on the category of the final argument they generate.

- Final argument categorized as a noun phrase **N**. One of the most common kind of modifiers is the prepositional phrase. Our GCG annotates the preposition as a head and the noun phrase following a final argument. Figure 4.35 shows the reannotation for *for now* where it detects the specific keyword *for* followed by either an **RB** or an **ADVP**. Other scenarios to show a noun phrase being a final argument for a modifier can be seen in Figure 4.36 as the way to annotate the *no matter* construction, or Figure 4.37, Figure 4.38 and Figure 4.39 where the last child was annotated by PTB as some variations of a noun phrase.

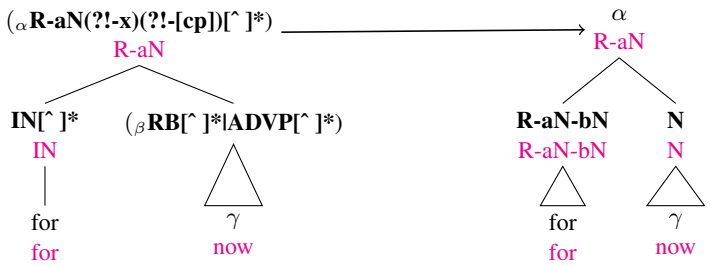


Figure 4.35: If β=**ADVP[^]*** then its leftmost child must be an **R-aN**.

- Final argument categorized as an **A-aN**. Figure 4.40 shows the reannotation for a modifier having **IN** and **JJ** as its left and right child. The adjective **JJ** is rewritten into an adjectival

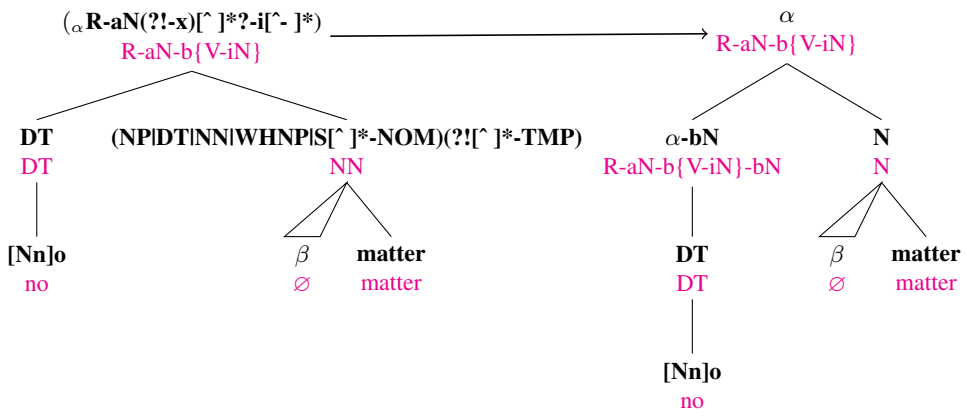


Figure 4.36: Branch off final argument **N**, special handling for "no matter".

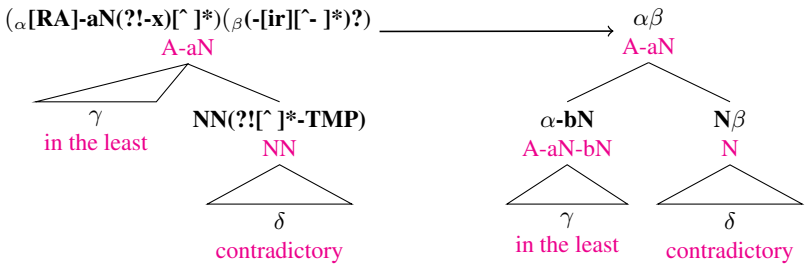


Figure 4.37: Branch off final argument **N**. This example has α=A-aN and β=$\varnothing$.

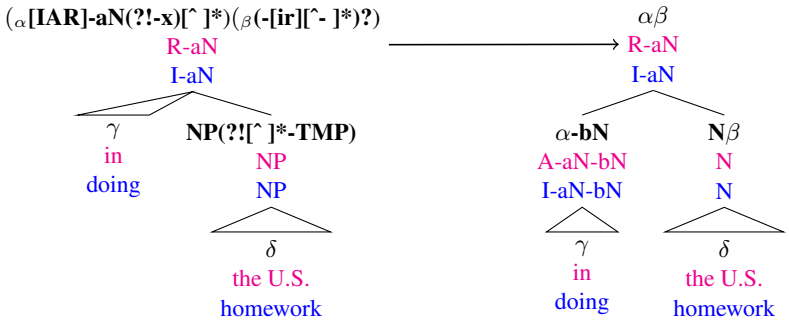


Figure 4.38: Branch off final argument **N**. This example has α=R-aN and β=$\varnothing$.

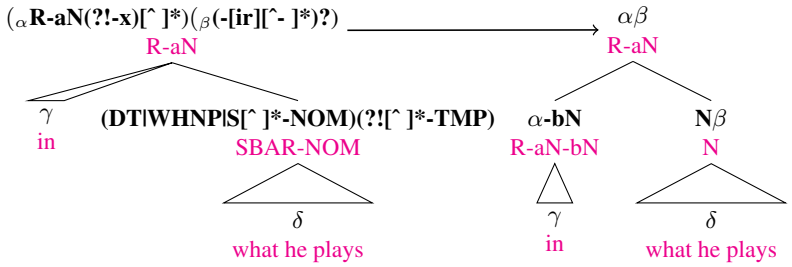


Figure 4.39: Branch off final argument **N**. This example has α=R-aN and β=$\varnothing$.

A-aN. The example on that figure is for *at least* where *least* is an adjectival final argument to preposition *at*.

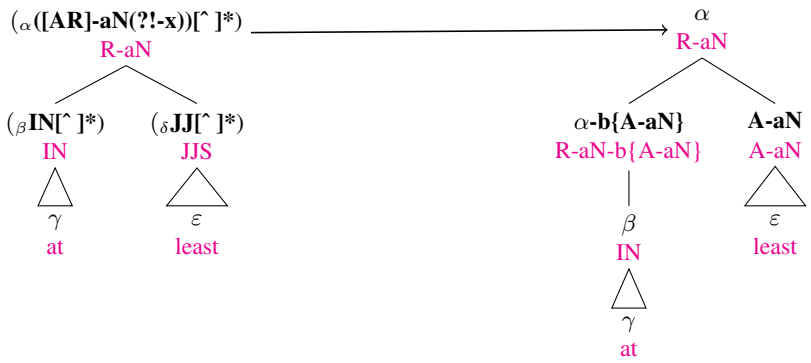


Figure 4.40: Branch off final **VP|ADJP** as argument **A-aN**. This example has α=R-aN.

- Final argument categorized as an **E-gN**. Figure 4.41 shows a rule for a *tough* construction like *tough for X to Y*. This reannotation re-writes an **SBAR** into an **E-gN**. It got an **E** due to the keyword *for* there to serve as a good signal of an embedded. It got a **-gN** because of the co-indexation of a **WHNP** to a null **NP** that gives a good indication of a gap.

- Final argument categorized as an **I-aN-gN**. Figure 4.42 is for another variation of a *tough* construction, e.g. *tough to Y*. This is one of the most specific and complicated rule.

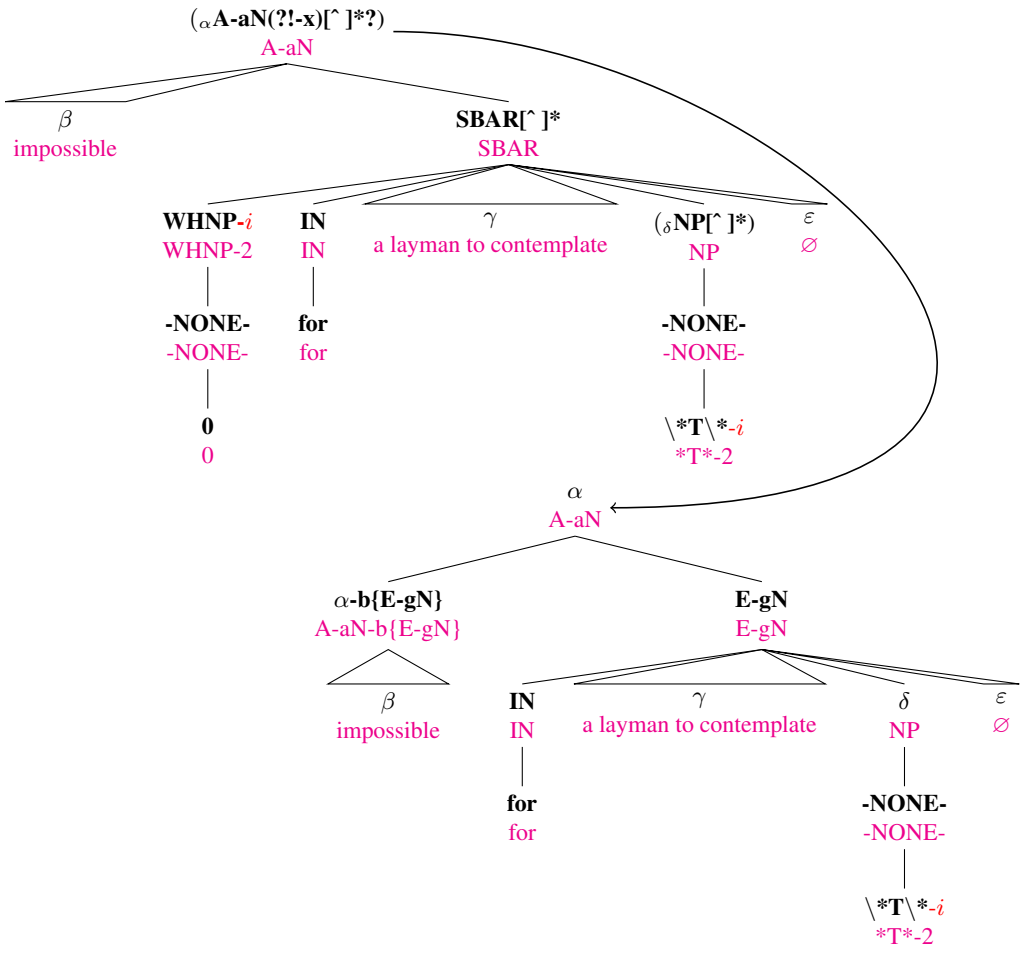


Figure 4.41: Branch off final **SBAR** as argument **E-gN** (*tough for X to Y* construction):

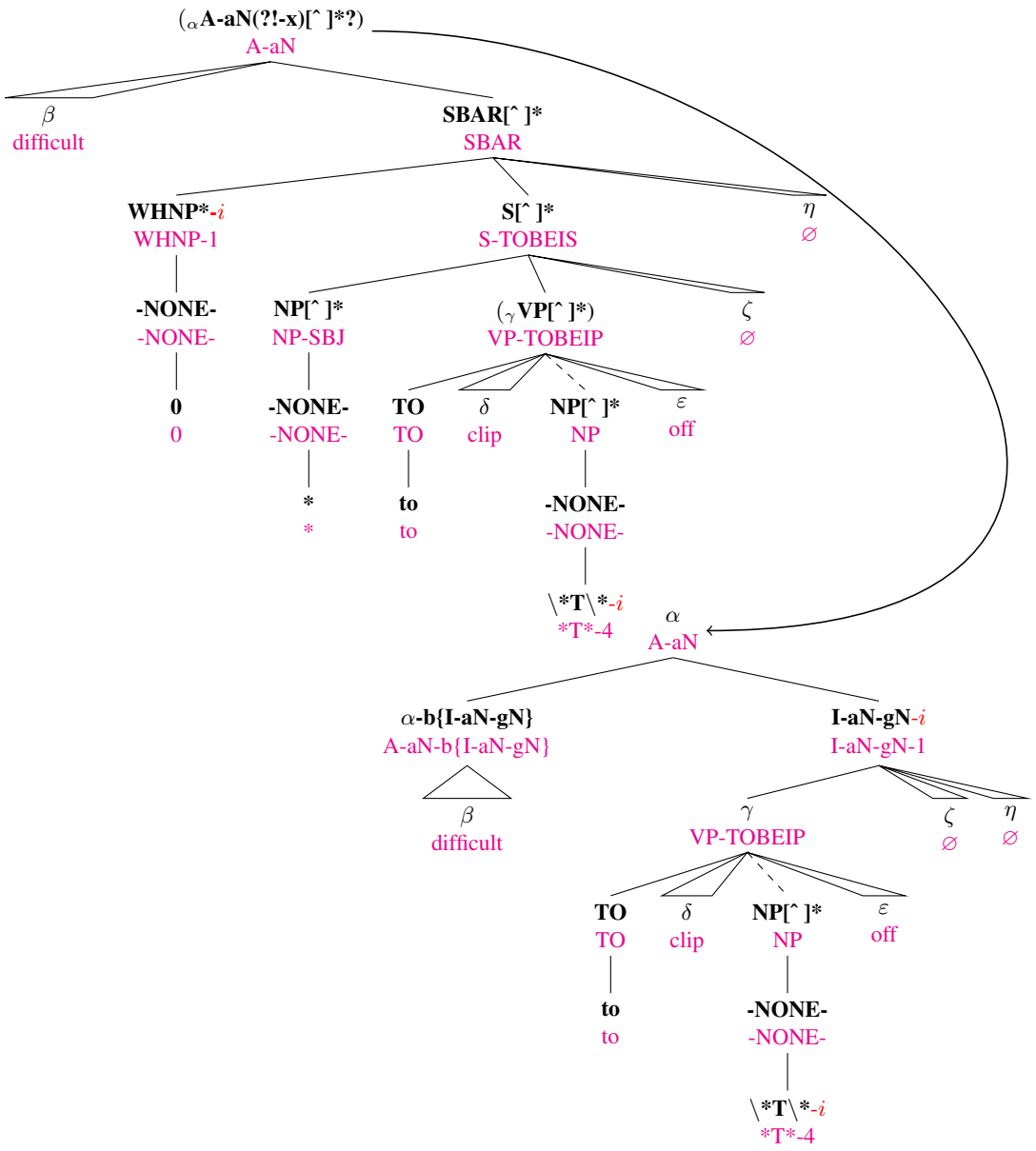


Figure 4.42: Branch off final **SBAR** as argument **I-aN-gN** (*tough to Y* construction):

Final argument attachment for interrogative phrase

In English, a common way to turn an affirmative statement into a question is to prefix the statement with an auxiliary verb such as *do*, *have*, and *be*. This way makes the original statement become a final argument to the auxiliary verb. For example, the affirmative statement is *the milk expired* and one of the questions that can be constructed from this statement is *has the milk expired ?*. In this case, the phrase *the milk expired* is analyzed as a final argument to *has* and is given the category **L** because this is a participial phrase as shown in Figure 4.46. Similarly, the rule at Figure 4.44 deals with *be* questions and the rule at Figure 4.43 is for the *do* questions or the ones started with a modal. One difference to note between *be* and *do* questions is the category of the final argument. The *be* questions generate an **N** argument while the *do* questions usually go with a base-form phrase **B**.

Another common constructing of questions is to have the content of the question followed by *, he asked* or something similar. In this construct, the added phrase after the question content is analyzed as a final predicative argument to the question content. This is handled by the rule shown in Figure 4.45.

Final argument attachment for complementized phrase

Complementized phrases are usually composed of a *that* followed by a verb phrase. In GCG analysis, *that* is the head and the verb phrase is its final argument. The category of this final argument is a finite verbal phrase **V**. PTB use **IN** as the category for *that*. We used this category as a matching condition to re-write the final argument attachment for complemented phrase as shown in Figure 4.47. For the complementized phrase that does not start with a *that*, we can still recognize it by the parent category which was hypothesized as a **C** through the hit of previous rules and the rightmost child is of category **S-TOBEVS** because complementized phrase must be ending with a verb phrase. This rule is illustrated in Figure 4.48. If the parent node is categorized as **C**, but the rightmost child is an **SBAR** phrase starting with a *for* then it's more likely to be an embedded final argument as shown in Figure 4.29. If the parent is a **C** but the rightmost child is not of a sentential category, i.e. **S-TOBE[VIBA]P** then the final argument attachment will get a **-aN** on its category to denote that it is sententially incomplete as shown in Figure 4.33.

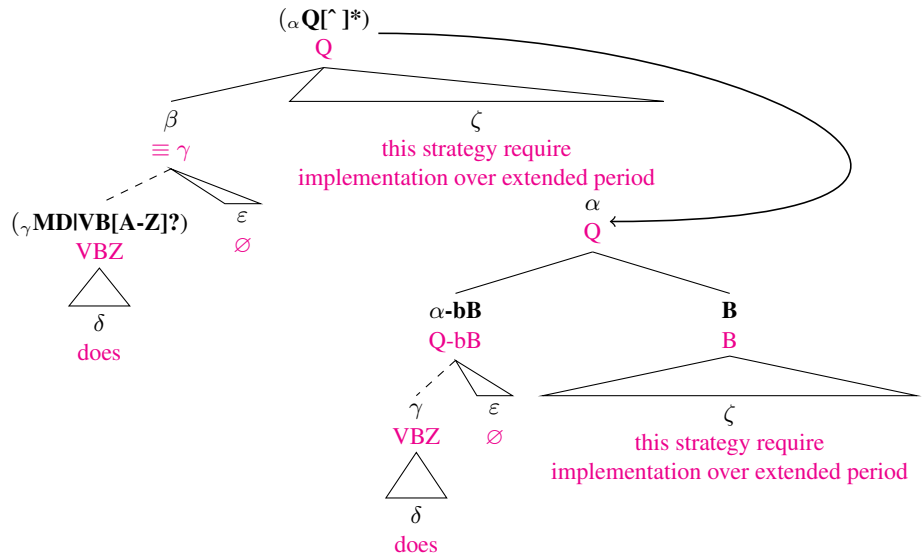


Figure 4.43: Polar question: branch off initial **B-aN**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ = VBZ, but β and γ could be different. If γ=**VB[A-Z]*** then δ=(**[Dd]oes|[Dd]o|[Dd]id|'d**).

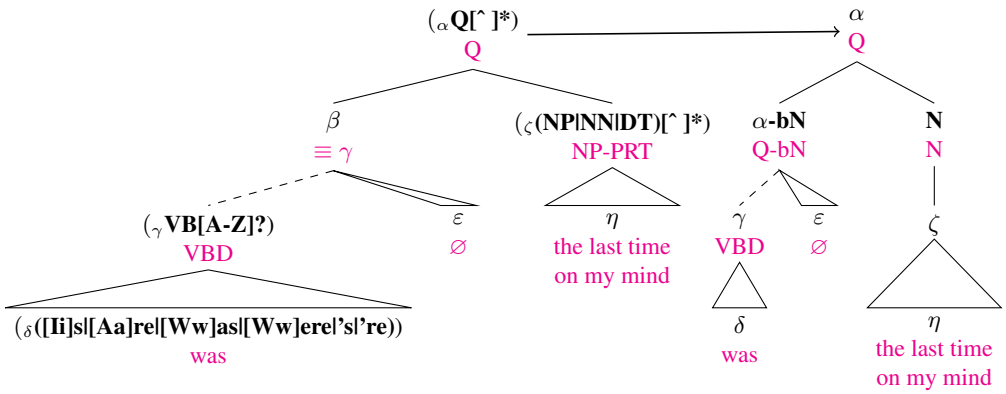


Figure 4.44: Polar question: branch off initial **N**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ as it is the case almost all the time in the corpus.

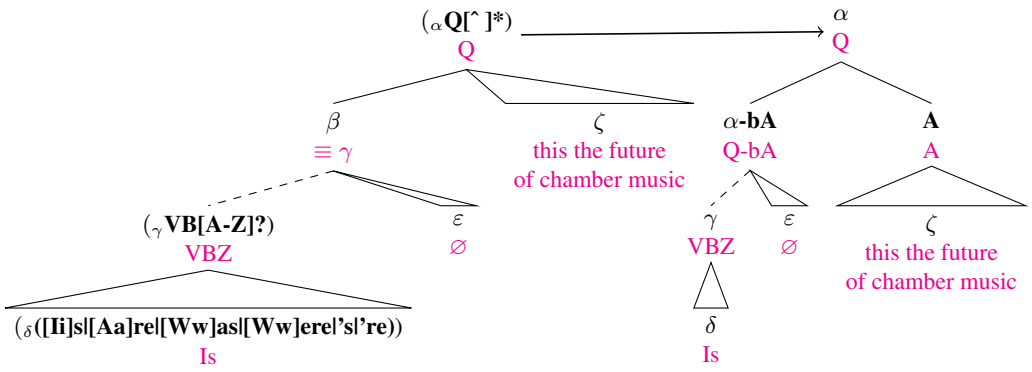


Figure 4.45: Polar question: branch off initial **Q-bA**-taking auxiliary. γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ = VBZ but β and γ may be different.

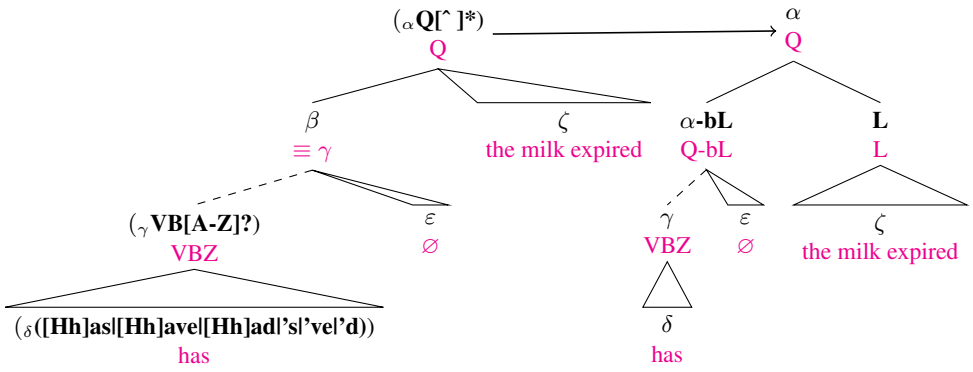


Figure 4.46: Polar question: branch off initial **L-aN**-taking auxiliary. The γ is the left-most pre-terminal under β and β is the left-most child of α, so γ is the left-most pre-terminal under α. This example has $\beta \equiv \gamma$ as it is the case almost all the time in the corpus.

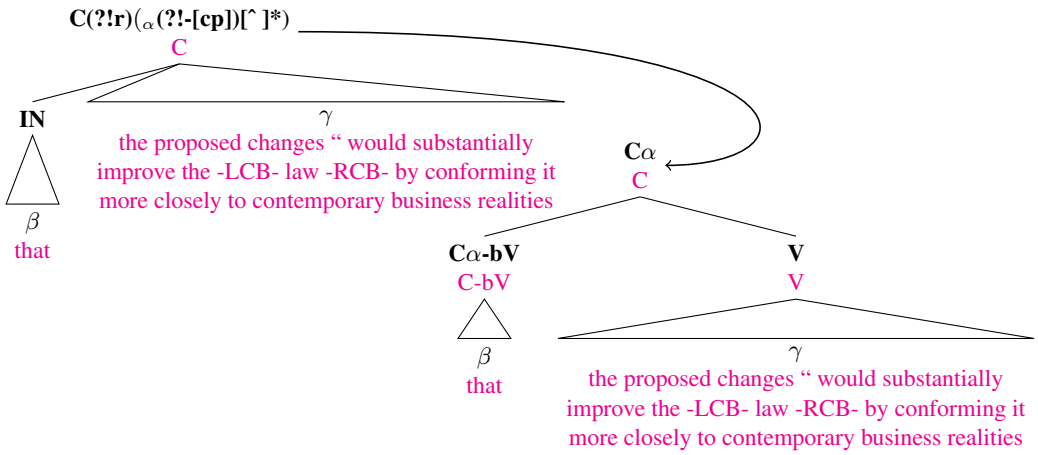


Figure 4.47: Embedded sentence: Branch off initial complementizer. This example has α=$\varnothing$.

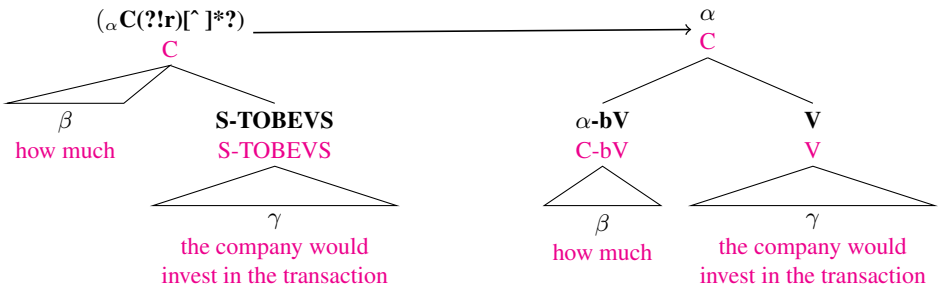


Figure 4.48: Branch off final **S** as argument **V**.

Final argument attachment for embedded phrase

An embedded phrase usually starts with a *for*, so the rule to recognize a final argument attachment for it is very straightforward. If the parent node is known to be an embedded, i.e. annotated with an **E** and the leftmost child is of PTB category **IN** that categorize a *for*, then create a new right child node to cover the rest of the children. This right child node is the final argument and will be categorized with an **I** as shown in Figure 4.49. This is inline with the rule **E** $\rightarrow$ **E-bI I**.

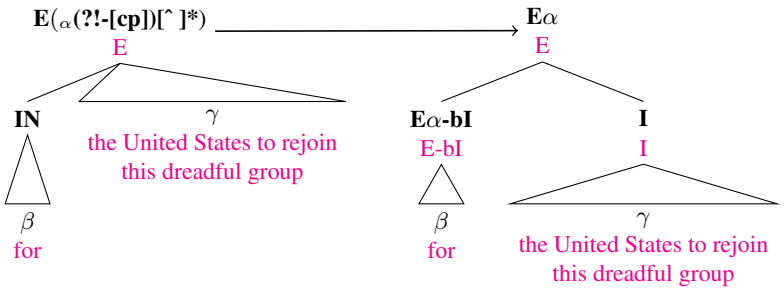


Figure 4.49: Embedded sentence: Branch off initial complementizer. This example has α=$\varnothing$.

Final argument attachment for genitive phrase

A genitive phrase always starts with *of* followed by a noun phrase, so the rule to recognize a final argument for this phrase is to check for the parent of category **O** and the leftmost child has PTB's category **IN** that categorize a lexical *of*. This rule will create a new right child node of category **N** to cover the rest of the children and it is the final argument as shown in Figure 4.50.

Final argument attachment for noun phrase

Figure 4.51 and Figure 4.52 show the reannotation for a specific type of noun phrase that composed of a copular *be* followed by a predicative phrase. If this pattern is recognized, these rules will generate a final argument of category **A-aN** for the predicative phrase. The difference between these two rules is that the predicative **A-aN** can transform unarily into a noun phrase **N** if it was annotated as a noun (**NN**), a noun phrase (**NP**), some form of nominal phrase (**-NOM**), or predicative (**-PRD**) on PTB.

Another specific type of noun phrase is the one composed of *having* followed by a verb

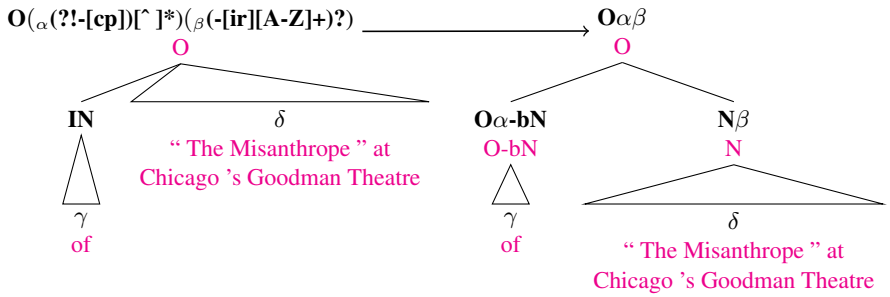


Figure 4.50: Embedded sentence: Branch off initial complementizer. This example has α=β=$\varnothing$.

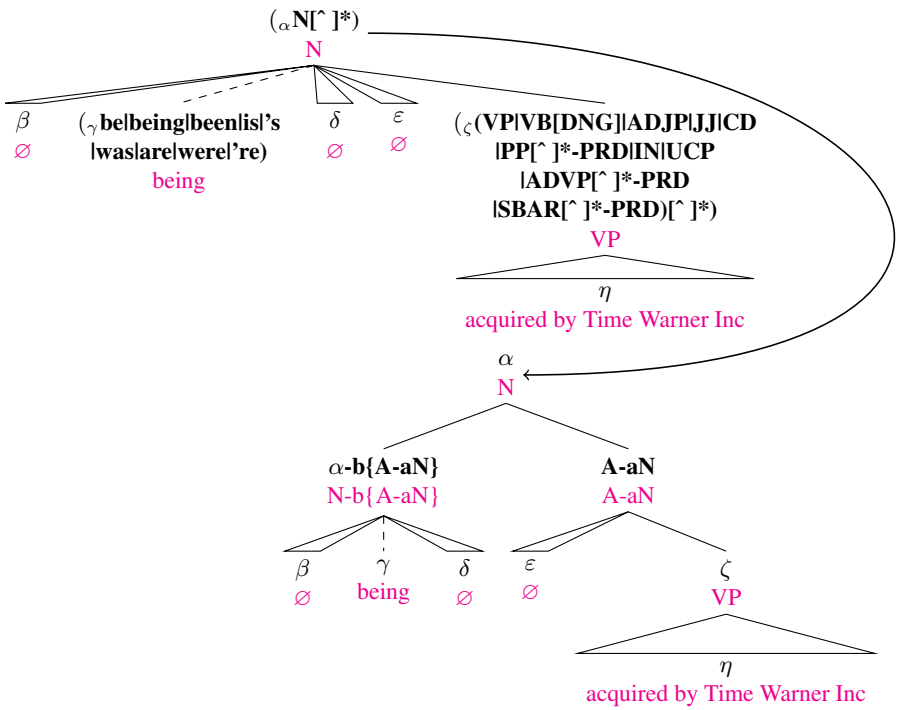


Figure 4.51: Branch off final predicative phrase after the copular *be* as argument **A-aN**. Top-right-most node in δ must not be an **RB**. Subtree ε, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If γ=**'s** then its immediate parent must be a **VBZ**. If ζ=**SBAR[^]*-PRD** then its left-most child must not be a **WH[^]*** or **(IN that)**. This example has α=N.

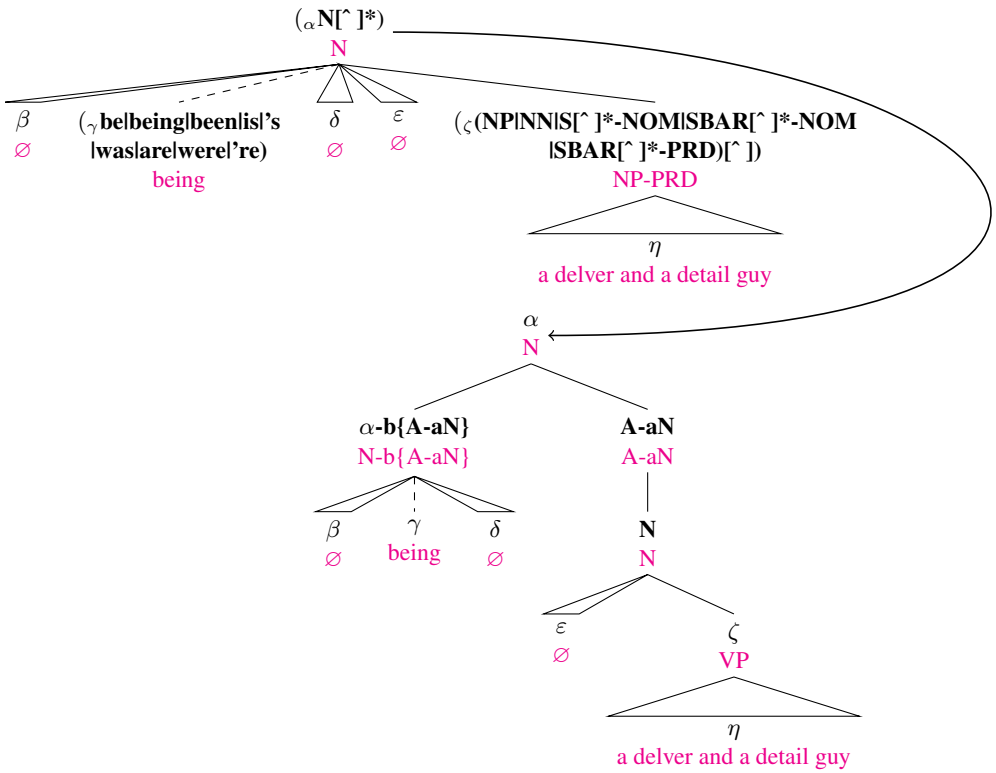


Figure 4.52: Branch off final post nominal phrase as argument **A-aN**. Top-right-most node in δ must not be an **RB**. Subtree ε, if not empty, must be an **(RB .*)** such as **(RB then)**, **(RB n't)** or **(RB not)**. If γ=**'s** then its immediate parent must be a **VBZ**. This example has α=N.

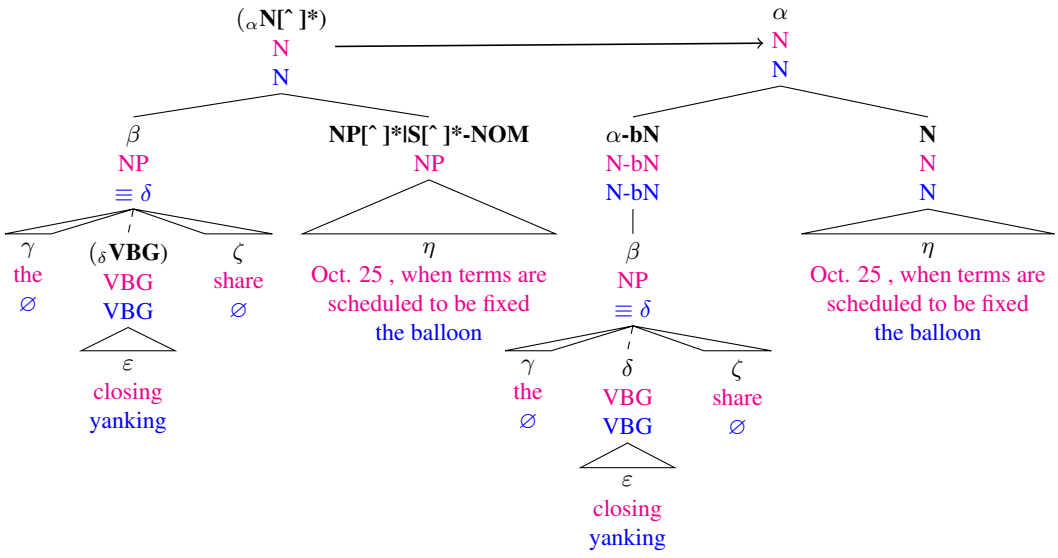


Figure 4.53: Gerund: branch off final argument **N**. All but three sentences in the corpus have the need to have $\beta \not\equiv \delta$ as shown by the magenta example. This requires that δ not have any category in **VB|JJ|MD|TO|NN**. The rest of the matches of this rule are when $\beta \equiv \delta$ as depicted by the blue example.

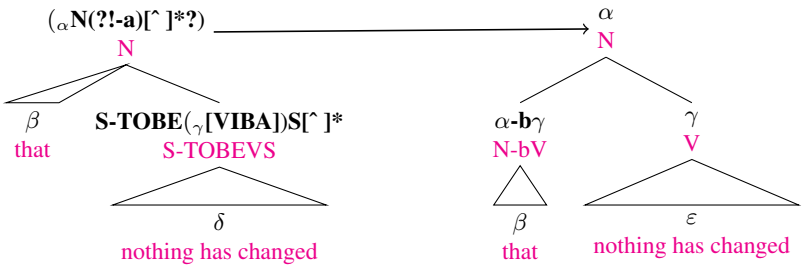


Figure 4.54: Branch off final **S** as modifier **[VIBA]**. This example has γ=V.

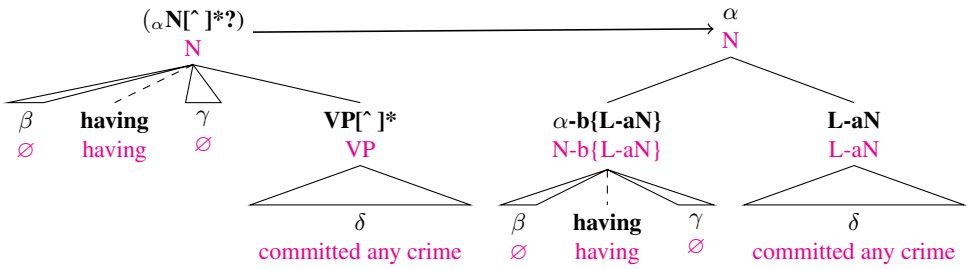


Figure 4.55: Branch off final argument **L-aN**.

phrase, e.g. *having committed any crime* as shown in Figure 4.55. The noun phrase following *having* is a participial verb phrase, hence receiving a category **L-aN** and it is a final argument. A noun phrase can also be a gerund, as annotated by **VBG** in PTB, followed by a noun phrase **NP** or some form of nominal phrase **-NOM**. When this pattern is realized, the noun/nominal phrase will be reannotated to an **N** and served as a final argument to the newly generated head covering the rest of the children as depicted in Figure 4.53. The last scenario of a noun phrase as covered by this reannotation is the one ending with an **S-TOBE[VIBA]S** that will be branched off as a final argument of a new head covering the rest of the children as shown in Figure 4.54.

4.2 Reannotation rules for initial and final modifier attachment

Initial and final modifiers are also called pre-modifiers and post-modifiers in literature. An initial modifier is a modifier that goes before the constituent that it modifies. Likewise, a final modifier is a modifier that goes after the thing it modifies. For example, *a white ball on the shelf* has *white* as an initial modifier and *on the shelf* as a final modifier. By definition, modifiers are adjuncts to the sentence, e.g. the core meaning of the sentence is preserved even with the modifiers stripped off. For example, it is still *a ball* when removing both initial modifier *white* and final modifier *on the shelf*. However, both initial and final modifiers play a vital part in the semantic interpretation of the sentence. Recognizing and annotating initial and final modifiers correctly can be a crucial help for semantically related downstream tasks as mentioned in a couple of different evaluations in Chapter 6 and Chapter 7.

There are two types of modifiers: adjectival modifiers modifying noun phrases and adverbial modifiers modifying verbal, adverbial, adjectival phrases or clauses. Adjectival modifiers are

annotated with **A-aN** and adverbial modifiers get an **R-aN**. In this GCG analysis, the category **A** is overloaded to represent both adjectival and predicative, so only if it is an adjectival phrase (e.g. an attribute) can it be a modifier. Due to the many different types of constituent it modifies, the adverbial modifier attachment rules outnumber the ones for adjectival modifier attachment.

This section describes the reannotation process to identify and branch off the modifiers from a larger constituent. The three larger constituents that often come with some initial and/or final modifiers that we focus on are (1) verbal or sentential phrases, (2) nominal phrases, and (3) adverbial or adjectival/predicative phrases. The following two subsections will describe in details how an initial and a final modifier argument attachment is done to re-write that type of structure from PTB.

4.2.1 Reannotation rules for initial modifier attachment

We can classify the rules to do initial modifier attachments into three different groups based on the grammatical type of the parent node: the one with a verbal or sentential category, the one with the nominal category and the one with the adverbial or adjectival/predicative category.

Initial modifier attachment for verbal or sentential phrase

The first modifier attachment rule for a verbal or sentential phrase at Figure 4.56 shows the recognition of a lexical colon separating an initial modifier (before the colon) from the head or the modified constituent (after the colon). Note that not everything before a colon can be a modifier. They have to be some variation of an adverbial, prepositional, or the often conflated **SBAR**. This rule also further branches off the colon from the initial modifier on the next tree level.

Figure 4.57 shows the next rule to deal with initial modifier attachment for a verbal phrase or sentence where the modifier part is an infinitive, e.g. *[to get a good result] , he studied day and night.* In PTB, this type of infinitive initial modifier is often annotated as a sentence with a null subject. The lexical word *to* is also used as one of the key to recognize this structure. The initial modifier **R-aN** re-written is also unarily transformed to an **I-aN** to reflect that it is an infinitive before further unarily transforming to the verbal category of the PTB. The intermediate **I-aN** node inserted here will help the next reannotation step down the tree to treat this constituent as an infinitive.

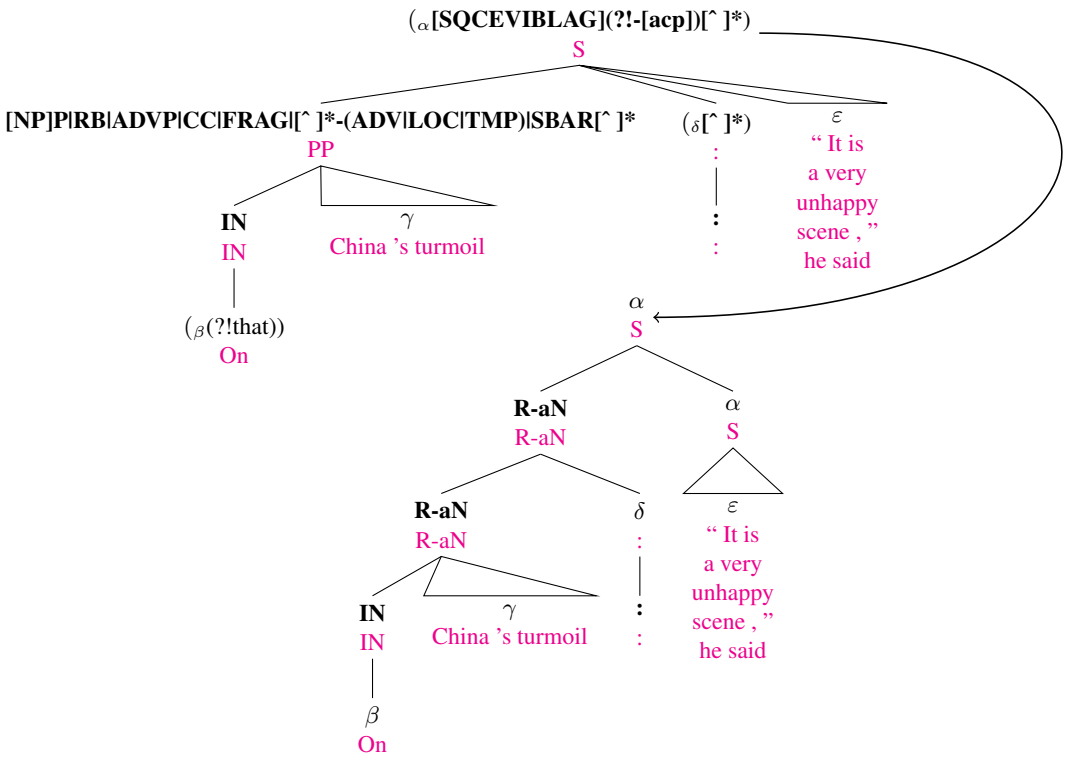


Figure 4.56: Branch off initial modifier **R-aN** with colon. This example has α=$\varnothing$.

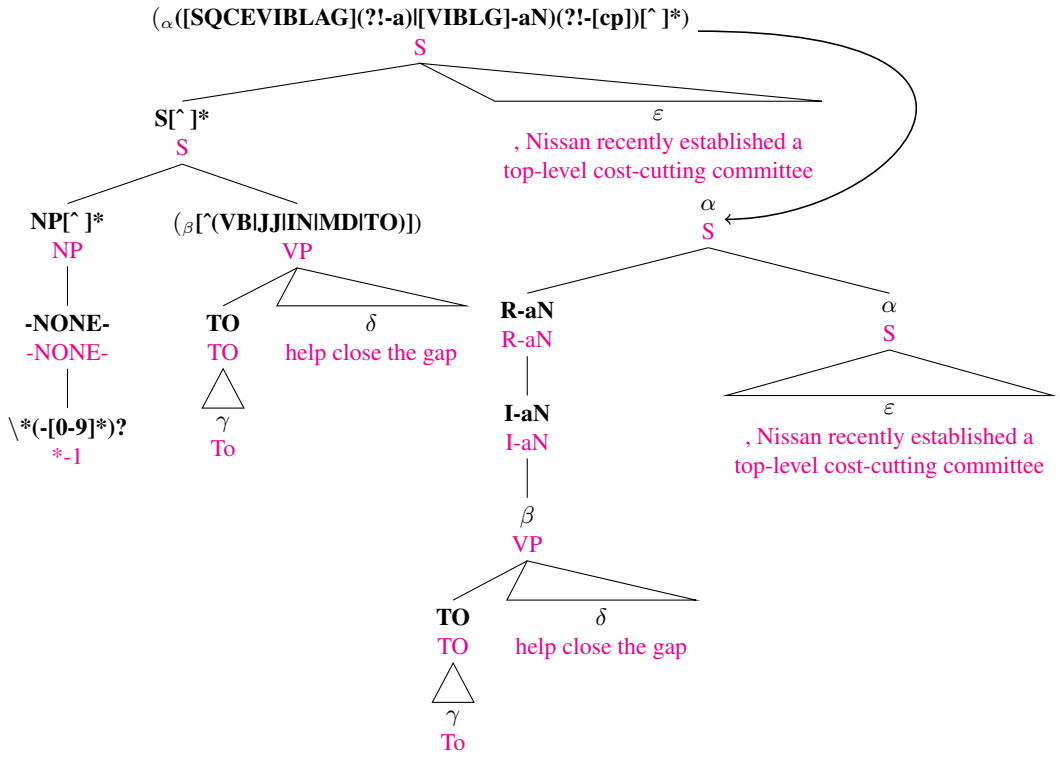


Figure 4.57: Branch off initial modifier **R-aN** and **I-aN**. This example has α=S.

Beside the infinitival, participial sentential initial modifiers are another form of sentential initial modifier as shown in Figure 4.58. This construct usually starts with a verb in participle form. Again, PTB annotated this construct with an **S** that has an empty element as a null subject. We used that as matching conditions to re-write this construct into an initial modifier **R-aN** removing the null element.

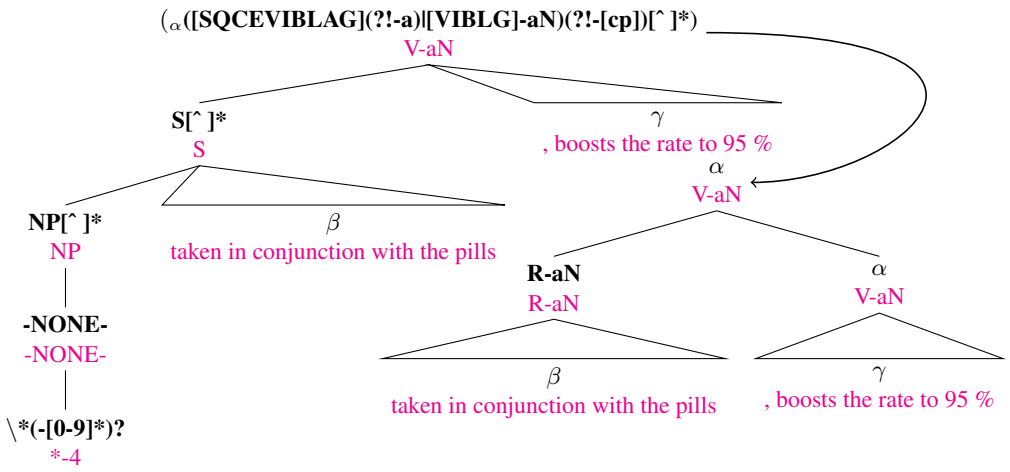


Figure 4.58: Branch off initial modifier **R-aN** and **A-aN**. This example has α=V-aN.

The last type of sentential initial modifier we deal with is the conditional initial modifier, e.g. *[if their jobs are terminated] , receive cash from the fund* as shown in Figure 4.59. PTB annotated this structure with an **SBAR** as a subordinating conjunction. To be considered an initial modifier, we restrict the matching condition to only the **SBAR** not having **-SBJ** and the left-most lexical cannot be a *that, for, where, when* as those could be some form of complementized, embedded or interrogative constructs.

The rule shown in Figure 4.60 groups a preposition followed by an adjective into some commonly known modifier, e.g. *at least*, *at most.* PTB is flat in the sense that it does not have a common node to represent this construct. We recognize them together as a single construct which is a modifier of category **R-aN**. Note that for the same lexical like *at least* could be annotated on PTB as an **ADVP** if it is under a nominal phrase as shown in Figure 4.64. This inconsistency makes parsing a harder task. This problem is corrected in this GCG analysis as they will be annotated as **R-aN** in all different contexts.

Another common initial modifier for a verbal or sentential phrase is concerned with timing,

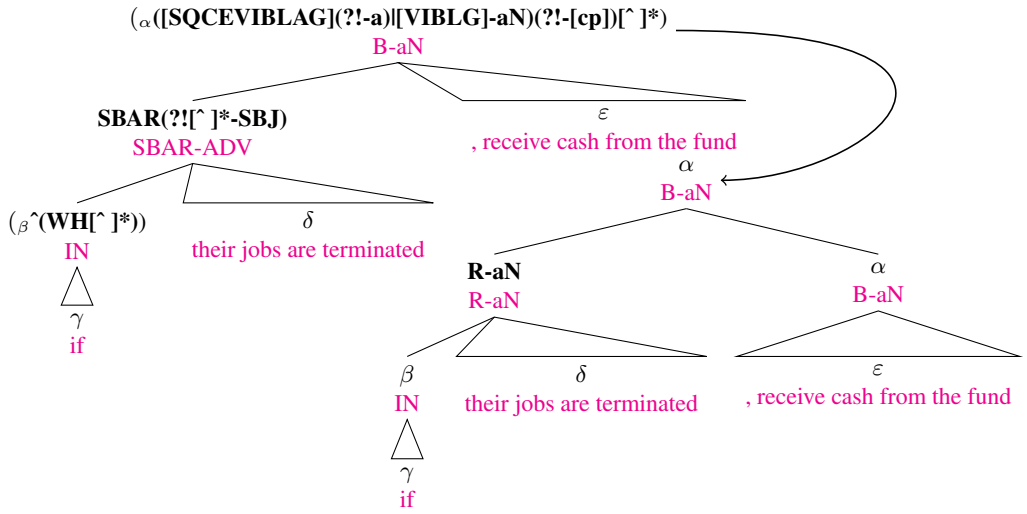


Figure 4.59: Branch off initial modifier **R-aN** from **SBAR**. If β=**IN** then γ must not be either $that$, for, $where$ or $when$.

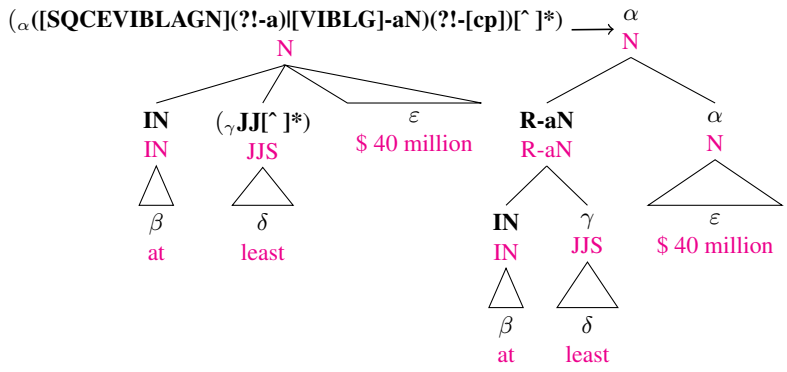


Figure 4.60: Branch off initial **RB** and **JJS** as modifier **R-aN** (e.g. "at least/-most/strongest/weakest"). The left-most branch of ε that started from α must not be **CC**.

e.g. *[Some nights] he slept under his desk*, as shown in Figuref 4.61. PTB considered this construct a noun phrase **NP** and used the category extension **-TMP** to denote the timing. We re-write it into an initial modifier attachment with the category **R-aN**.

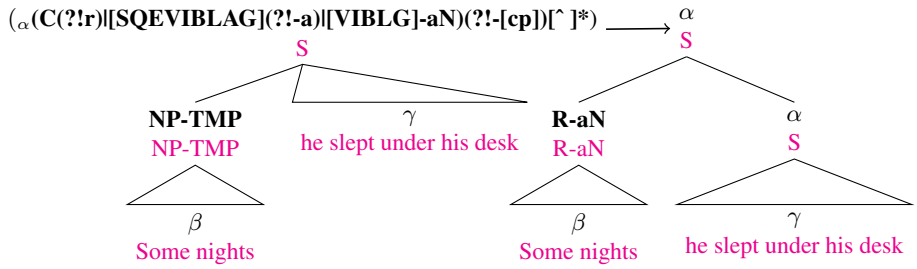


Figure 4.61: Branch off initial modifier **R-aN**. The top-left node of γ must not be a **CC**. This example has α=S.

The last initial modifier attachment rule for verbal or sentential phrase is shown at Figure 4.62. This is a catch all rule for any type of determiner, prepositional or adverbial phrase before the head of the verbal phrase or sentence allowing them to be an initial modifier attachment, given the category **R-aN**.

Initial modifier attachment for nominal phrase

If the parent node has category **N**, then this is a nominal phrase that we are trying to detect initial modifier attachment for. Our analysis shows there are only two kinds of left-most child that could be initial modifiers. They are either (1) a conjunction **CC** shown in Figure 4.63 or (2) an adverb or prepositional phrase shown in Figure 4.64. While the later is reannotated to an **R-aN** as usual, the former is chosen to be an **R-aN-x** to denote its lexicality. More about the difference between an **[AR]-aN** and an **[AR]-aN-x** will come on the next sub-section about initial modifier attachment for adverbial or adjectival/predicative phrases. In both cases, the initial modifier is branched off to modify a new nominal head that covers all the rest of the children.

Initial modifier attachment for adverbial and adjectival/predicative phrase

When the parent node is of category **[AR]-aN(?!-x)**, i.e. it does not have **-x** yet, then it can be further re-written as a modifier and a new head. Figure 4.65 and Figure 4.66 show two

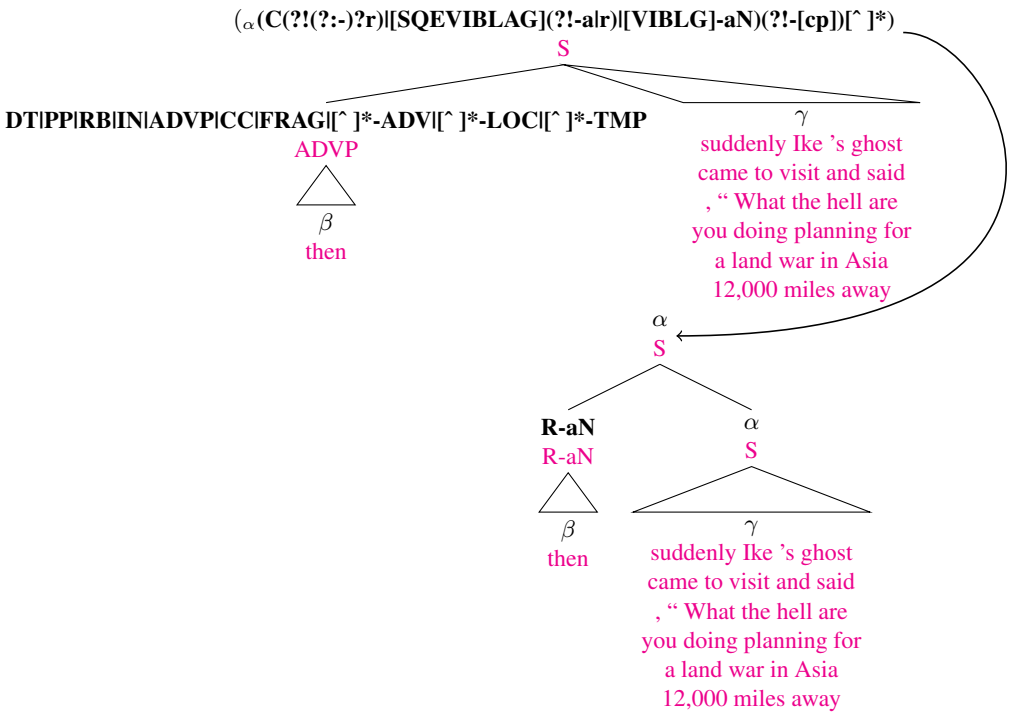


Figure 4.62: Branch off initial modifier **R-aN** (including determiner, e.g. *both in A and B*. The top-left node of γ must not be a **CC**. This example has α=S.

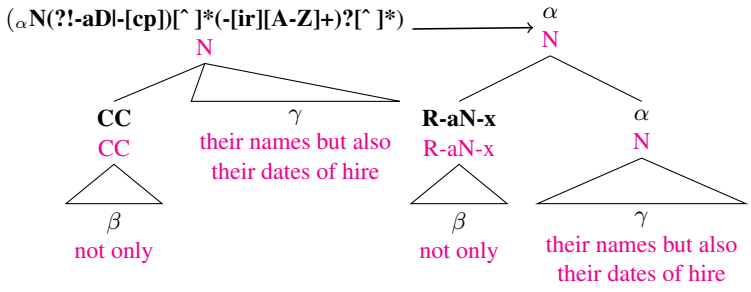


Figure 4.63: The left-most branch of γ started from α must not be a **PP** or **WHPP**. This example has α=N.

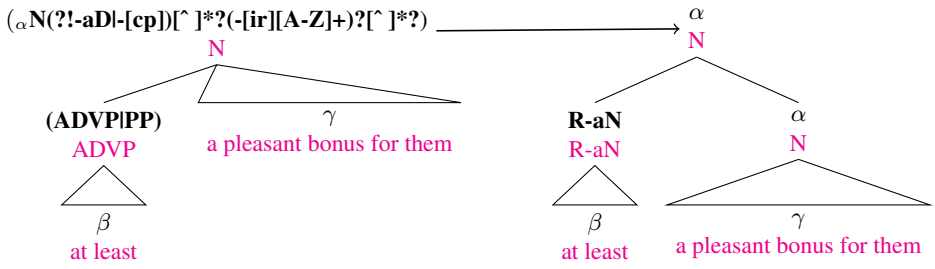


Figure 4.64: Branch off initial modifier **R-aN** (including determiner, e.g. *both in A and B*. The top-left node of γ must not be a **CC**. This example has α=N.

variations of this rule to spin off an initial modifier of category **R-aN-x** and a new head of the same category as the parent node to cover the rest of the children. While the matching condition of Figure 4.66 is so specific that the parent must have only two children of those specific categories, the one on Figure 4.65 is more relaxed in term of the number of children because a **WRB** (meaning Wh-adverb in PTB) could be a modifier of the following adjective or predicative phrase.

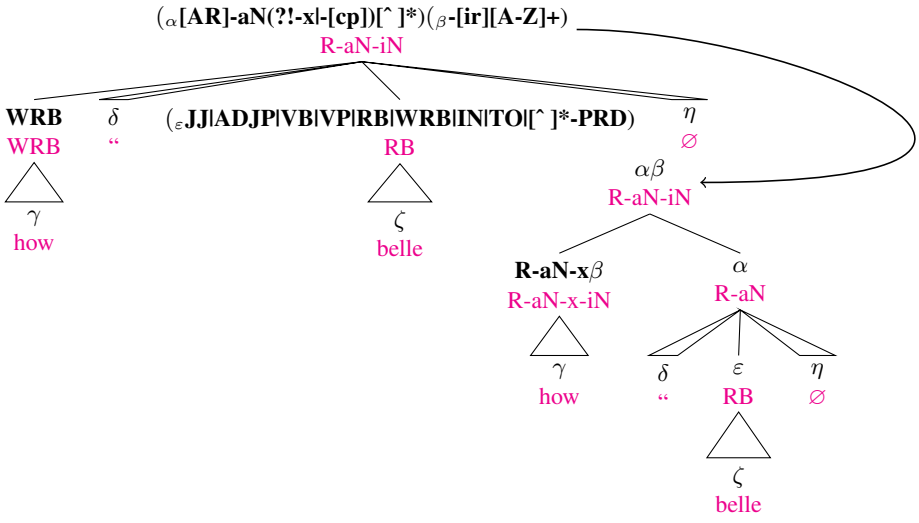


Figure 4.65: Branch off initial modifier **R-aN-x-iN/R** of **A-aN/R-aN**. The δ in this rule must not have any PTB category that contains a vertical bar such as **PRT|ADVP**. This example has α=R-aN and β=-iN.

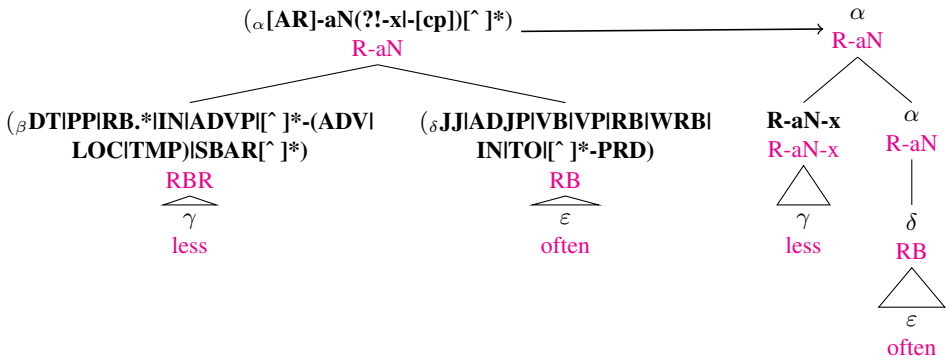


Figure 4.66: Branch off initial modifier **R-aN** of **A-aN** or **R-aN**. If β=**IN** then δ must also be of category **IN**. If β=**SBAR[^]*** then its left-most branch must be of category **IN** and the child of this child must not be $that$. This example has α=R-aN.

The rest of the rules to deal with initial modifier attachment for adverbial and adjectival/predicated phrases work on parent nodes already annotated with a **-x**. The two rules at Figure 4.67 and Figure 4.68 show the analysis of an adverbial modifier modifying an adjectival phrase. Note that both children, the adverbial and the adjectival phrase, got **-x** on their categories. Similarly, the analysis of an adverbial modifier modifying another adverbial phrase is done in Figure 4.69 and Figure 4.70.

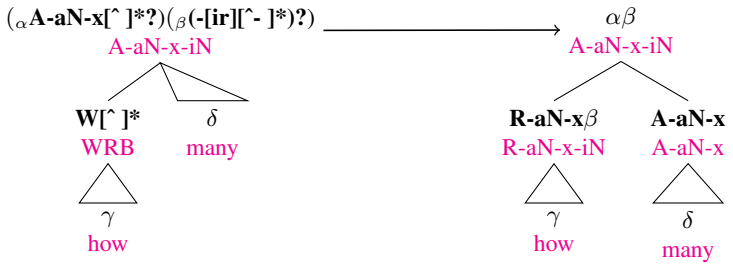


Figure 4.67: Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=A-aN-x and β=-iN.

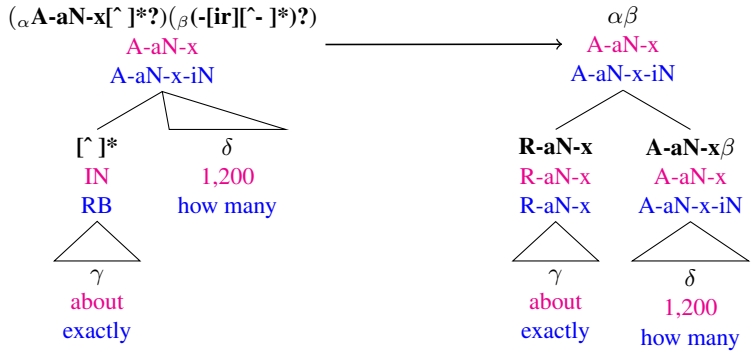


Figure 4.68: Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. The magenta example has α=A-aN-x and β=$\varnothing$. The blue example has α=A-aN-x and β=-iN.

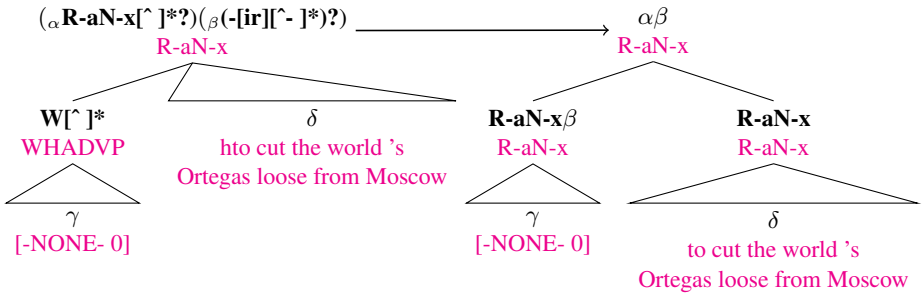


Figure 4.69: Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=R-aN-x and β=$\varnothing$.

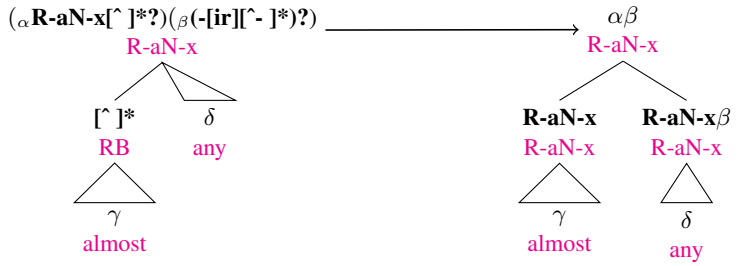


Figure 4.70: Branch off initial modifier **R-aN-x**. Both γ and δ are not $\varnothing$. This example has α=R-aN-x and β=$\varnothing$.

4.2.2 Reannotation rules for final modifier attachment

We focus our analysis on only final modifier attachment for (1) nominal phrase and (2) verbal or sentential phrase because these two types of phrase are most commonly found having final modifier attachment. One crucial difference between the modifiers of these two phrase types is that the modifier of a nominal phrase is an adjectival phrase hence will get category **A-aN** and the modifier of a verbal or sentential phrase is an adverbial phrase and will get the category **R-aN**.

Final modifier attachment for nominal phrase

The first rule to deal with final modifier attachment for nominal phrases detects the special construct having a nominal phrase followed by a colon then another nominal phrase as shown in Figure 4.71. The phrase after the colon, including the colon, is considered a final modifier and will receive a category **A-aN**.

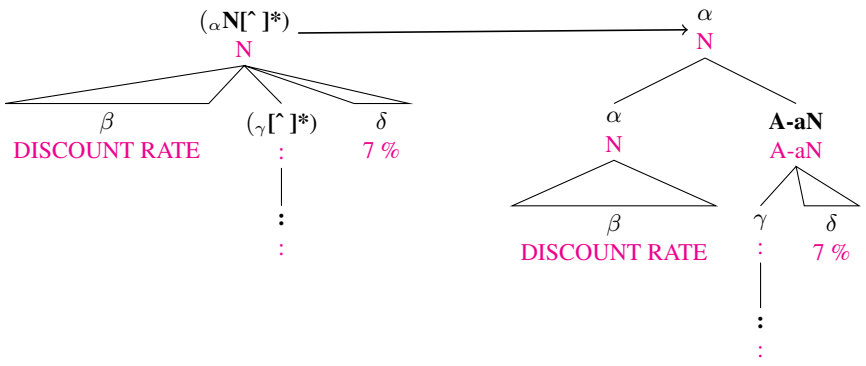


Figure 4.71: Branch off middle modifier **A-aN** colon.

The second rule for final modifiers of nominal phrases targets temporal (time) and location (place) specifiers that come after a nominal phrase. In PTB, **-TMP** and **-LOC** were used as extensions to denote the time and location. We used these as matching conditions to spin off final modifiers for nominal phrases as shown in Figure 4.72.

Next about final modifier attachment of nominal phrases, if a nominal phrase is composed of two smaller nominal phrases without any sign of them being in a coordination conjunction then the second nominal phrase is annotated as a modifier of the first one. Using PTB's category

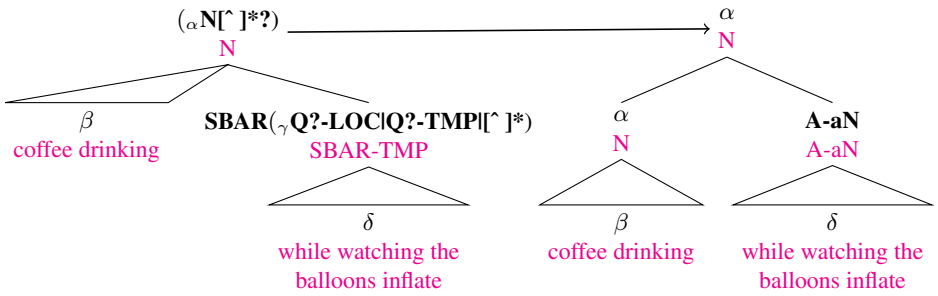


Figure 4.72: Branch off final **SBAR** as modifier **A-aN**. If γ is not ending with **(-LOC|-TMP)** then the top-left node of δ must be an **IN** and its child must not be a **that**.

extension **-NOM** as a marker of nominal phrase, this rule is shown in Figure 4.73.

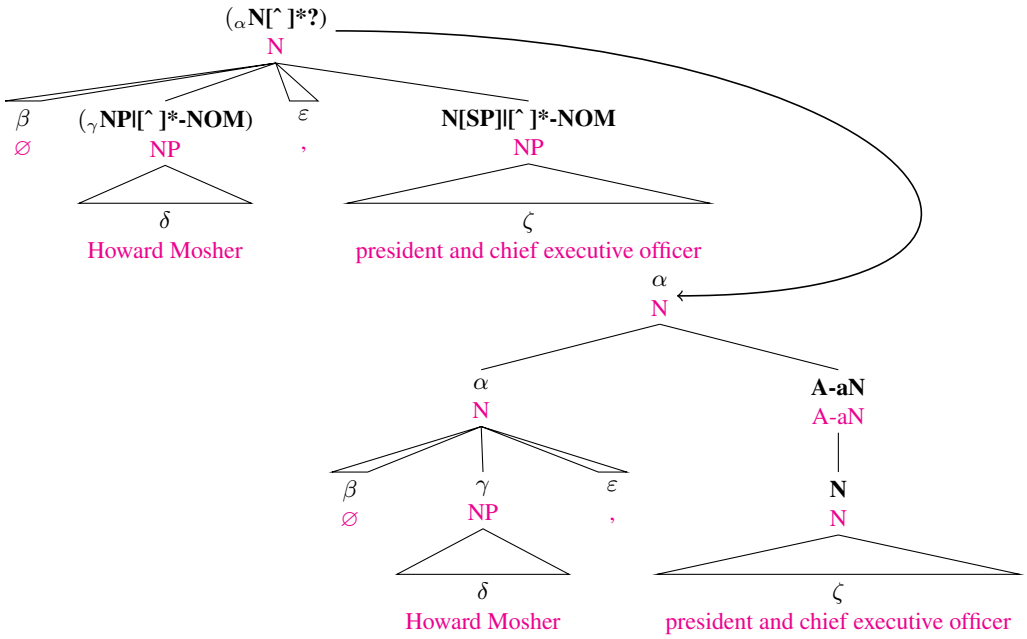


Figure 4.73: Branch off final modifier **A-aN** appositive **N**. In this rule, α must not have a child of category **CC**.

The next two rules dealing with final modifiers of nominal phrases look at nominal phrases ending with a verbal phrase. For example, *final guidelines [to be published in early November]* in Figure 4.74 or *Bell [based in Los Angeles]* in Figure 4.76. The verbal phrase in these cases

are adjectival final modifiers and will get the category **A-aN**. The difference between these two rules is that if the modifier is an infinitival phrase then it will be unarily transformed into an **I-aN**. In either cases, a new head is formed to group the rest of the children and get the same category with the parent node.

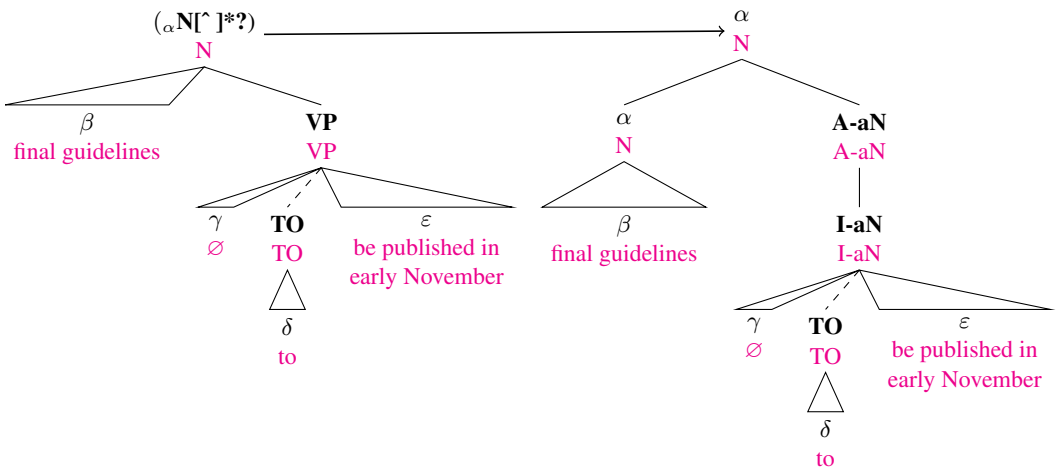


Figure 4.74: Branch off final modifier **A-aN** infinitive phrase (with **TO** before any **VB**). In this rule, γ must not contain any **VB**.

The last catch-all rule for final modifier of nominal phrase is to look at the last child of the nominal phrase. If the last child is a reduced relative clause, a prepositional phrase, a temporal, a locative, or some form of adjectival/adverbial phrase then they will be considered final modifier and will be annotated with an **A-aN**. This is shown in Figure 4.75.

Final modifier attachment for verbal or sentential phrase

All final modifiers for verbal or sentential phrases are adverbial phrases and will get category **R-aN**. The first analysis to find a final modifier attachment for a verbal or sentential phrase is to look for the rightmost child of a verbal category that is co-indexed with an ***ICH*** node which is a descendant of some other child. The ***ICH*** means "insert constituent here" in PTB, i.e. the co-indexation signifies that the rightmost child could syntactically be inserted at the position of the ***ICH*** node. If the leftmost child is a **VB[^]***, i.e. a finite verb in any form, then the rightmost child could be a final modifier of a verb phrase headed by the verb on the leftmost child. This is shown in Figure 4.77.

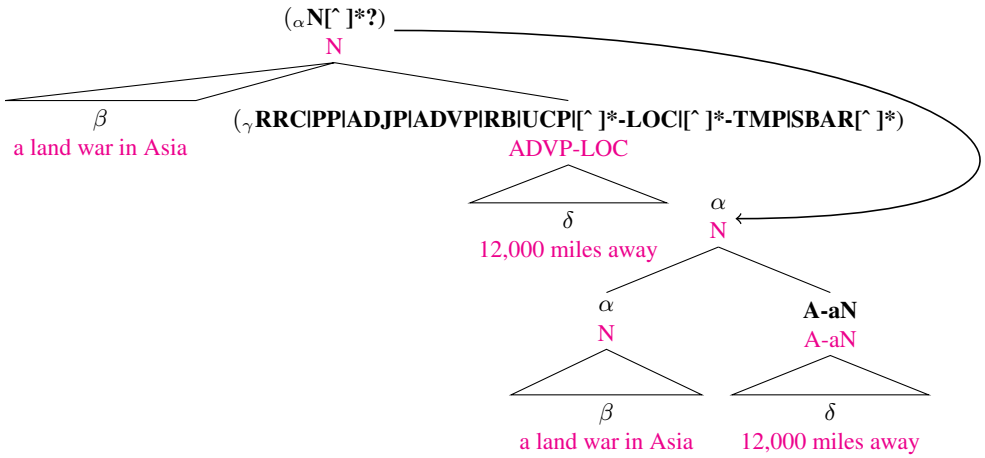


Figure 4.75: Branch off final modifier **A-aN**. If γ=**SBAR[^]*** then its left-most child must be **IN** covering something not a $that$. This example has α=β=γ=$\varnothing$.

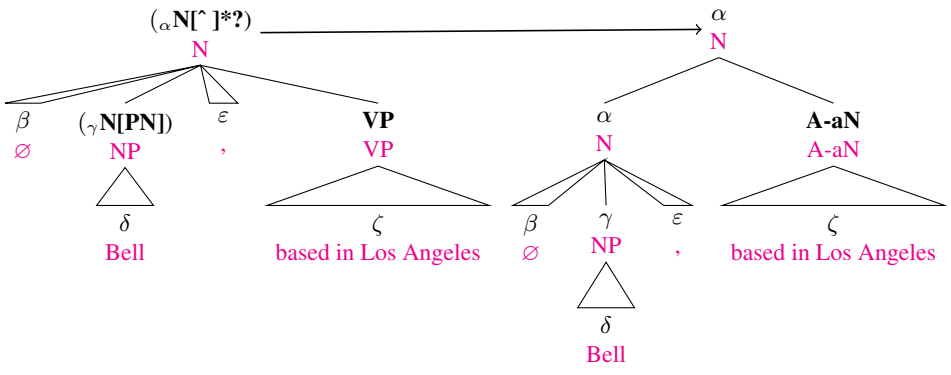


Figure 4.76: Branch off final modifier **AP** infinitive phrase (with **TO** before any **VB**).

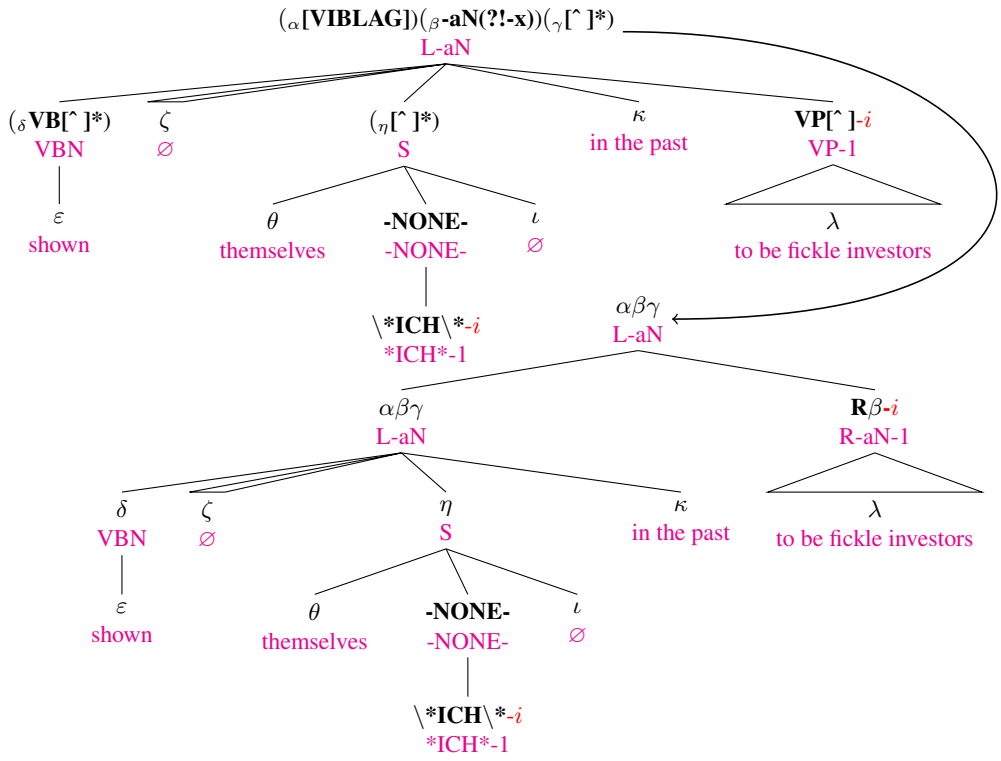


Figure 4.77: Branch off final modifier **R-aN** (extraposed from argument) This example has α=L, β=-aN, γ=$\varnothing$.

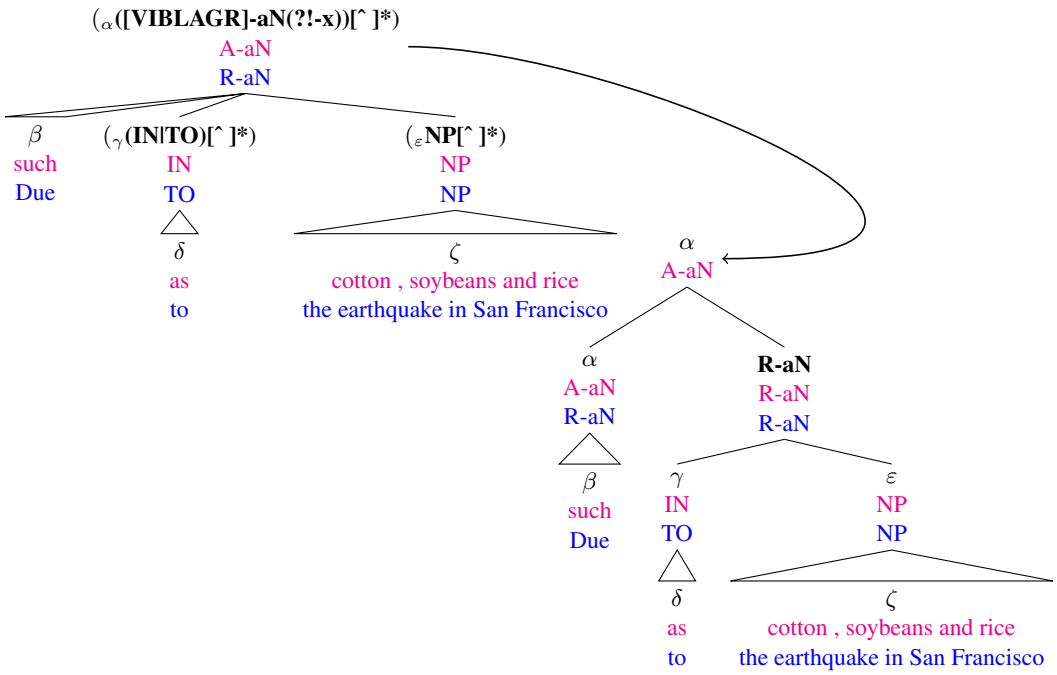


Figure 4.78: Branch off final **(IN|TO) + NP** as modifier **R-aN**.

The next five rules at Figure 4.79, 4.80, 4.81, 4.82, and 4.83 attempt to turn the rightmost child of category **S-TOBE(AS|IP|AP|VS|IS)** into a final modifier. This should be noted that PTB does not have **-TOBExx**. They are coming from the percolation process done as a pre-processing step before the reannotation takes place. While the first three of these four rules reannotate only the rightmost child of category **S-TOBExx** into a modifier on its own, the last rule at Figure 4.82 expects a lexical *so* to detect modifiers of the form *so + sentential phrase*. It also combines the second rightmost child (covering the lexical *so*) with the rightmost one into a final modifier.

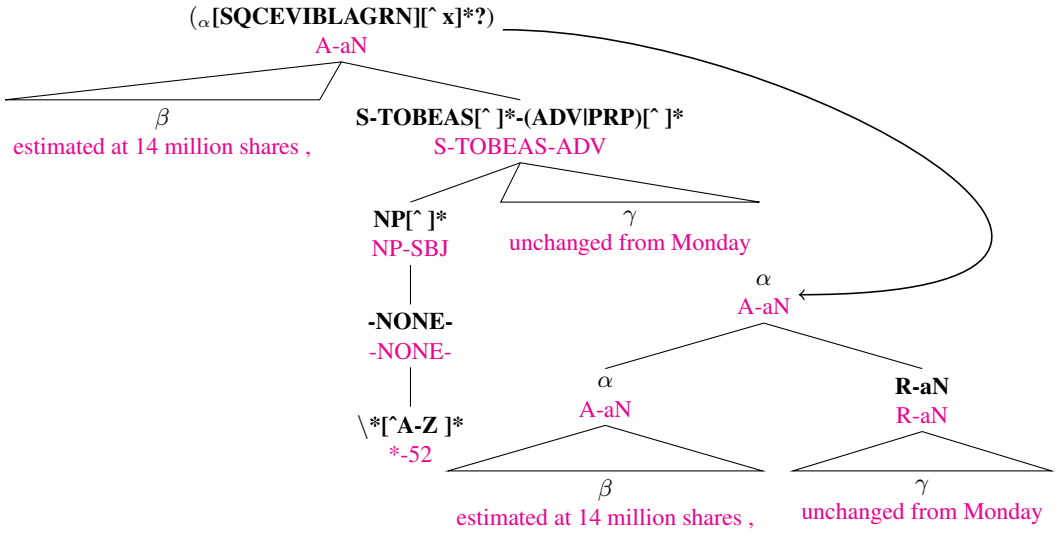


Figure 4.79: Branch off final **S-ADV** with empty subject as modifier **R-aN**.

The next construct of interest to spinning off final modifier for a verbal or sentential phrase is the detection of usage of colon and/or **SBAR** on the rightmost child. Figure 4.84 shows the usage of both a colon and the rightmost child of category **SBAR**, but Figure 4.86 and Figure 4.85 show when only one of the two conditions is satisfied. In any case, the colon is considered part of the final modifier.

Next are a couple of special rules (1) to turn the second to last child of category **IN** or **TO**, together with the last child of category **NP**, into a final modifier **R-aN** as shown in Figure 4.78; (2) to turn the last child of category **NP-TMP** into a final modifier **R-aN** as in Figure 4.87.

The last rule to deal with final modifier attachment for verbal or sentential phrase is one

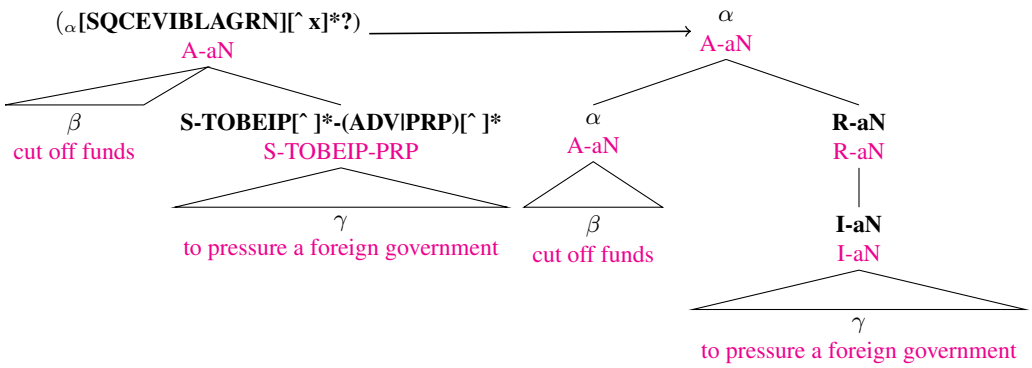


Figure 4.80: Branch off final **S-ADV** with empty subject as modifier **R-aN**.

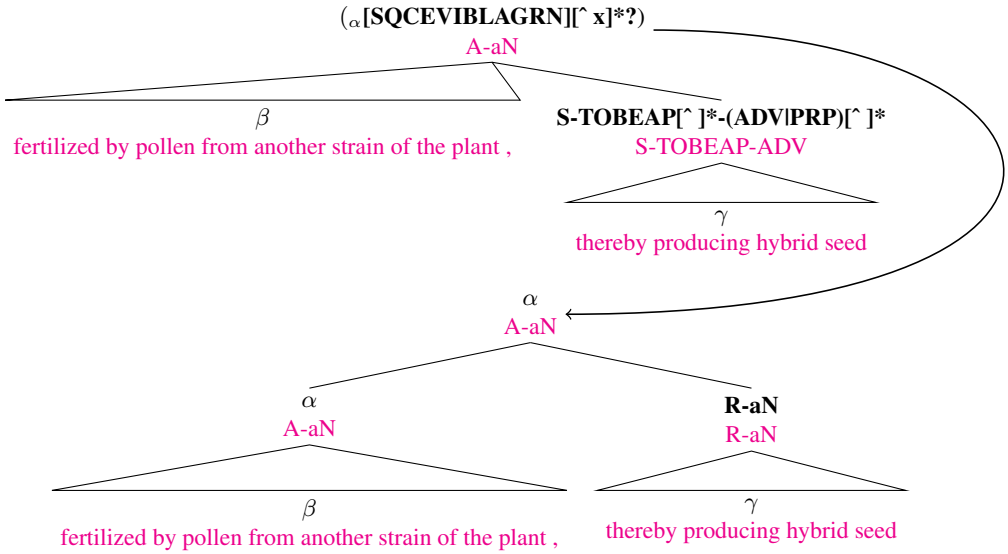


Figure 4.81: Branch off final **S-ADV** with empty subject as modifier **R-aN**.

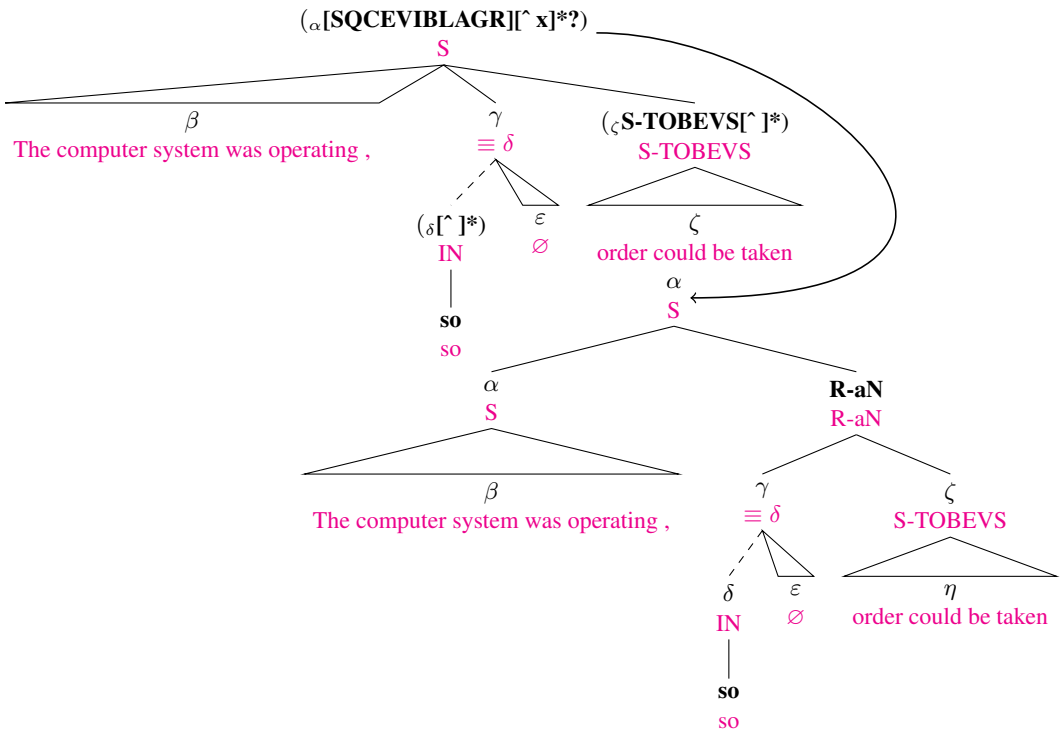


Figure 4.82: Branch off final 'so' + **S** as modifier **R-aN**. This rule either has $\delta \equiv \gamma$ or δ is the left-most leaf branch of γ.

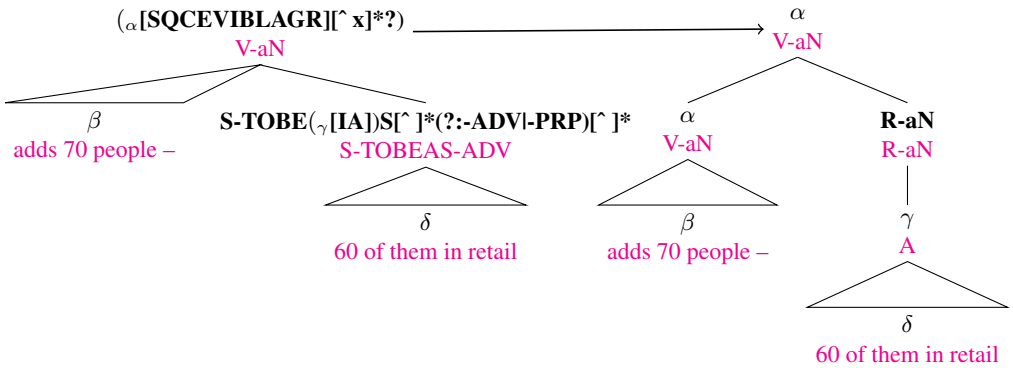


Figure 4.83: Branch off final **S-ADV** or **S-PRP** as modifier **R-aN**. This example has γ=A.

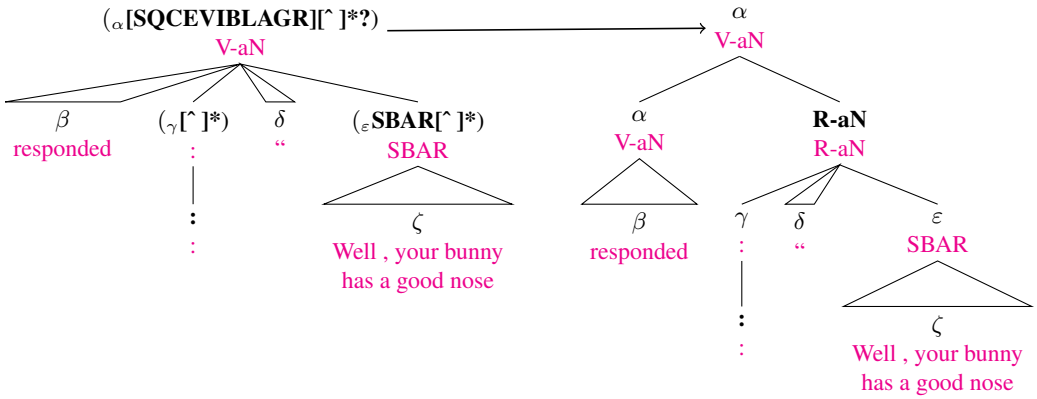


Figure 4.84: Branch off final **SBAR** as modifier **R-aN** colon. The top-left node in β must not be a **[^]*-ADV** and its sibling, if β has more than one top-level node, must not be a **([^]* :)**.

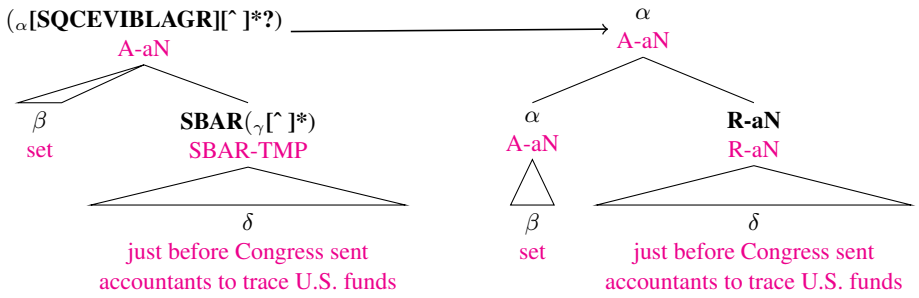


Figure 4.85: Branch off final **SBAR** as modifier **R-aN**. If γ is not ending with **-ADV**, **-LOC**, **-TMP**, or **-CLR** then the top-left node of δ must be an **IN** and its child must not be a **that**.

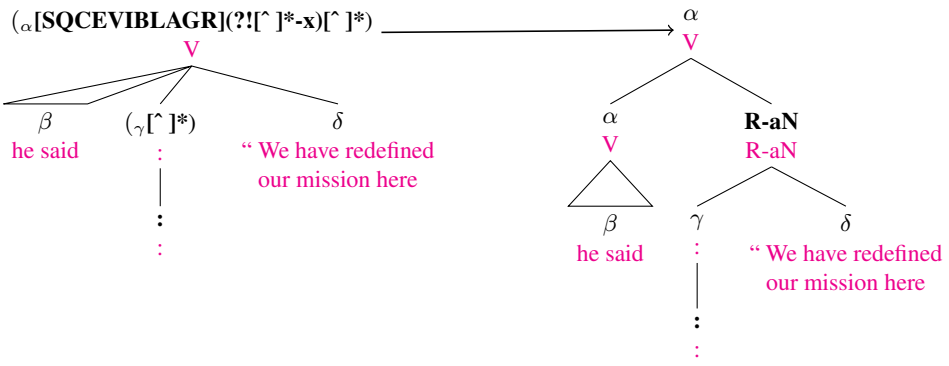


Figure 4.86: Branch off final modifier **R-aN** colon. The top-left node in β must not be a **.*-ADV** and its next sibling, if β has more than one top level node, must not be a **(.* :)**.

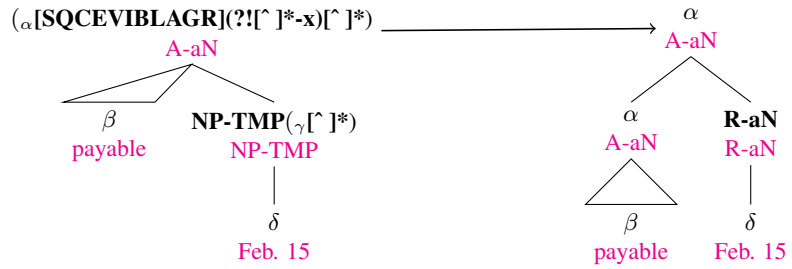


Figure 4.87: Branch off final modifier **R-aN**. This rule does not allow γ to have **-PRD**.

that catches all varieties of the category of the rightmost child to see if it could be a modifier as shown in Figure 4.88.

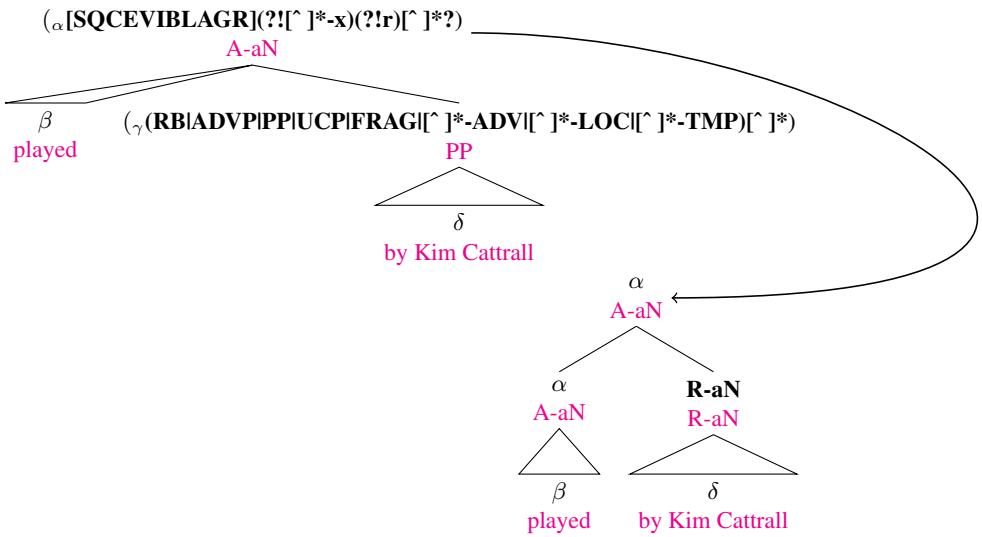


Figure 4.88: Branch off final modifier **R-aN**. The γ of this rule must not contain a **-PRD**.

4.3 Reannotation rules for coordinating conjunctions (-c/-d)

PTB trees use **CONJP** on coordinating words such as *and*, *or*, or *but* to denote a very flat coordinating conjunction with all conjuncts and coordinating words all on the same level. We further group commonly accepted conjunction phrases such as *as well as* under a **CONJP** node. All the **CONJP** are then turned into just **CC** as part of the preprocessing before the reannotation can take place. Complex conjunctions usually come with commas or semicolons that have been annotated with operator **-p** (punctuation), specifically **-pPs** for semicolons and **-pPc** for commas. This section will show how operators **-c** (initial conjunct), **-d** (final conjunct), and **-p** can be used to reannotate coordinating conjunctions.

First, we isolate any null element sibling away from the conjunction as shown in the rule at Figure 4.89. This is to counter the flatness problem of PTB. Any null element sibling here is believed to be linking to the entire conjunction node, not the individual conjuncts. There is also another isolation for the coordinating conjunction away from the prefix colon, again to counter

the flatness of PTB, as shown in the rule at Figure 4.90.

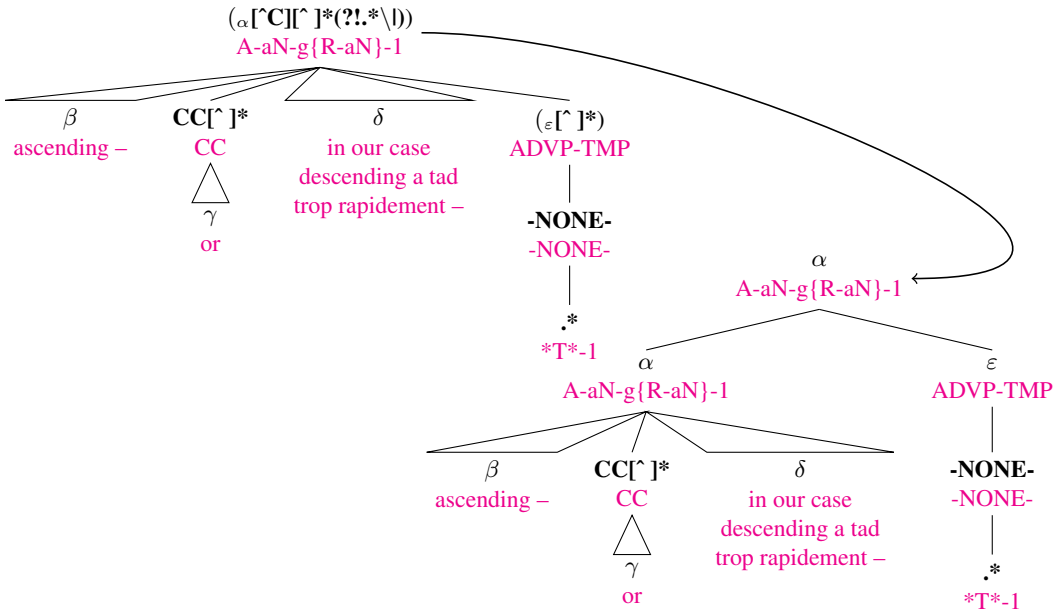


Figure 4.89: Pinch ... CC ... -NONE- and re-run. This example has α=N-aD.

The first in a series of rules to target the coordinating conjunction reannotation is the one to branch off the initial conjunct prior to the semicolon as seen in Figure 4.91. This branching-off uses the initial conjunct operator **-c**α on the newly created right child to denote that it has a left sibling as its initial conjunct argument α. The right child is also marked with a **-pPs** to denote that it has a semicolon as the first lexical item. This is a signal of work in progress in the next reannotation steps continuing down the tree to complete reannotating the entire coordinating conjunction. There are three other similar rules like this one. The one depicted in Figure 4.92 is for a comma separating the first conjunct, using **-pPc** instead of **-pPs**. The rules shown in Figure 4.93 and Figure 4.94 are like the one in Figure 4.91 and Figure 4.92, respectively, but for the case where parent node α is a primitive category, not a compositional category.

The (work in progress) signal introduced by either a **-pPs** (Figure 4.91 and Figure 4.93) or a **-pPc** (Figure 4.92 and Figure 4.94) above will continue to be rewritten by the rules at Figure 4.97 and Figure 4.98 to further branch off the semicolon or comma, respectively. Note that if the **-pPs/-pPc** is in the middle of the category, it means the semicolon/comma is first in its lexical coverage, but if they are at the end of the category then they mean there is an initial

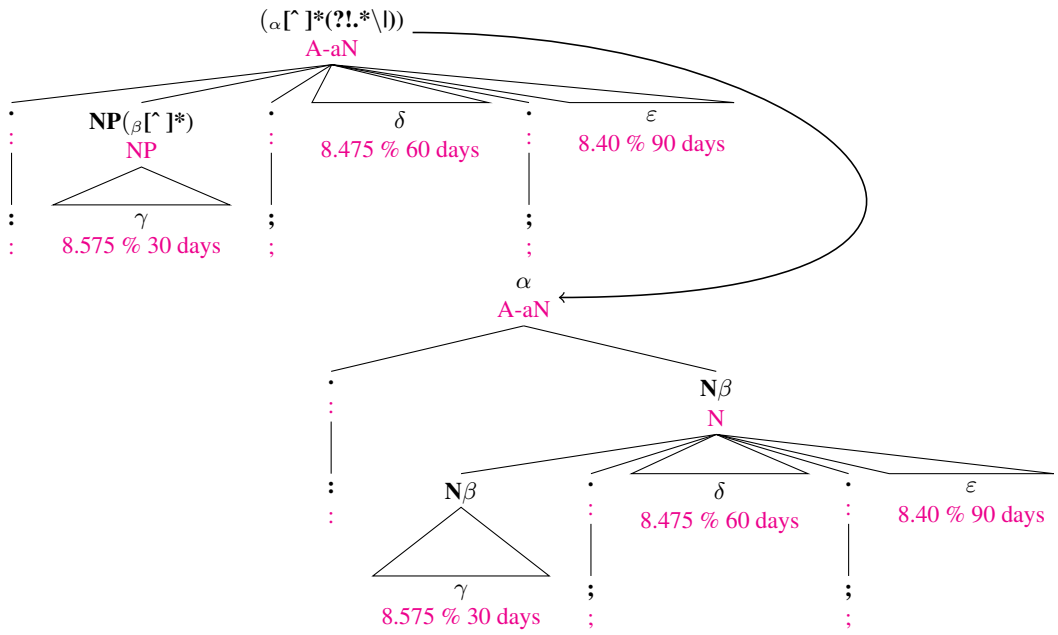


Figure 4.90: Branch off initial colon in colon...semicolon...semicolon construction. This example has α=A-aN and β=$\varnothing$.

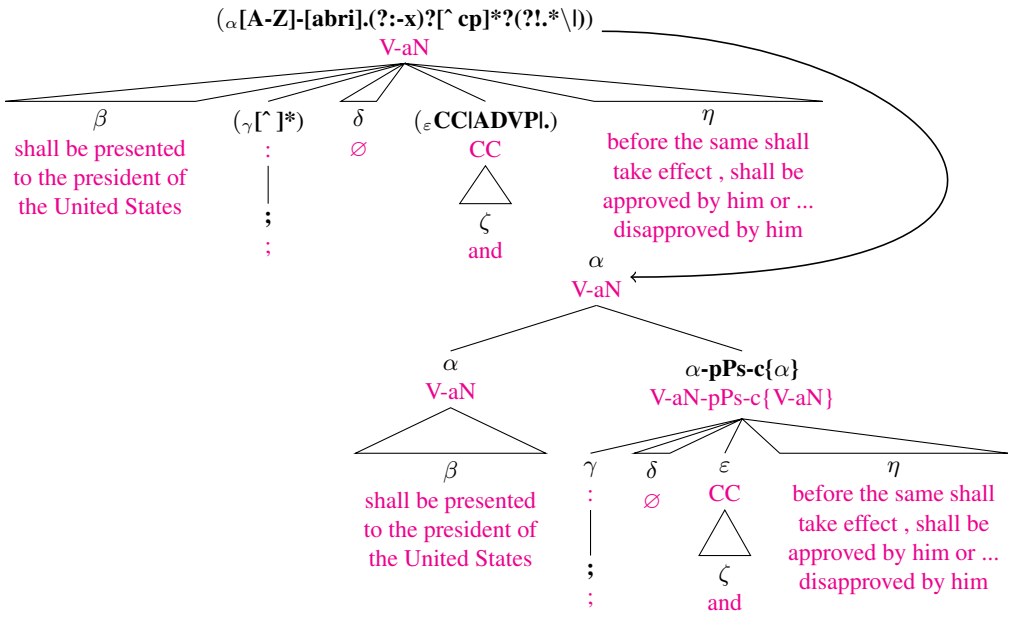


Figure 4.91: Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. If ε is a single character category such as **:** then it must be a pre-terminal and ζ=**;**. This example has α=V-aN.

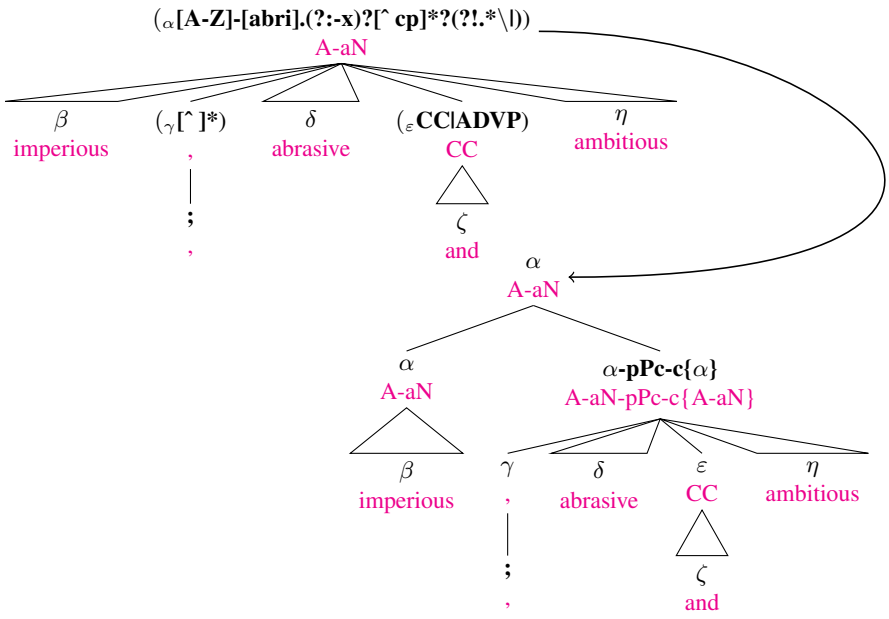


Figure 4.92: Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. This example has α=A-aN.

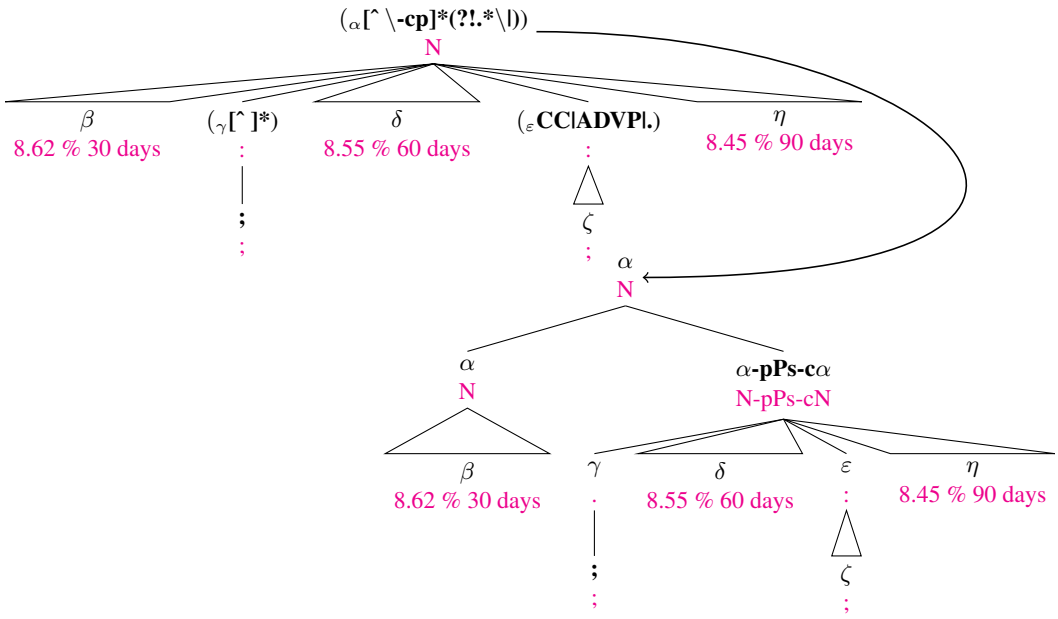


Figure 4.93: Branch off initial conjunct prior to semicolon delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. If ε is a single character category such as **:** then it must be a pre-terminal and ζ=**;**. This example has α=V-aN.

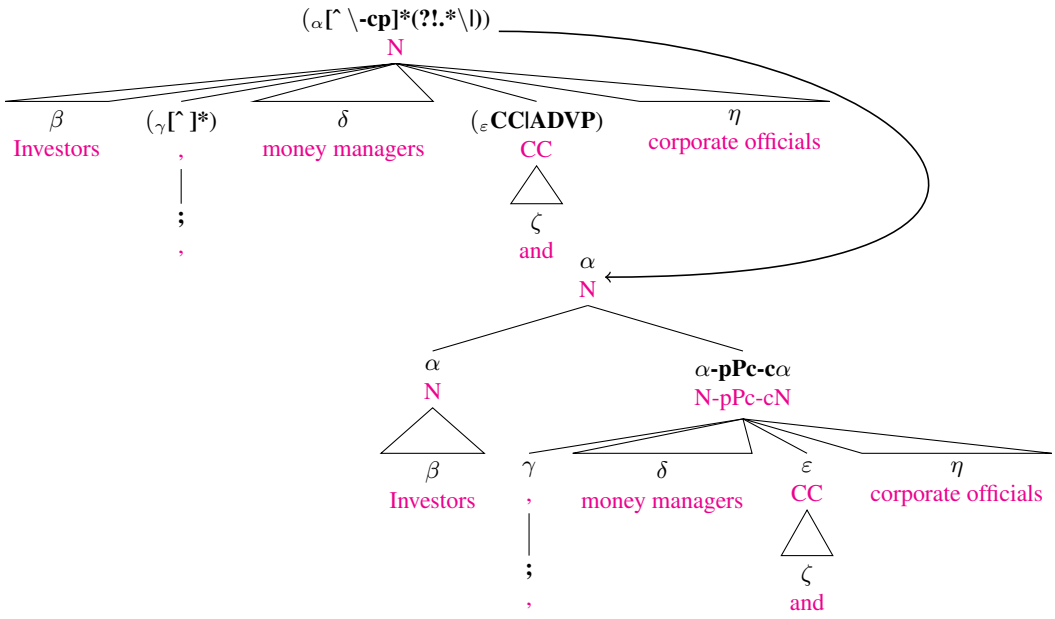


Figure 4.94: Branch off initial conjunct prior to comma delimiter. The β and η must not be $\varnothing$. If ε=**ADVP** then the left-most pre-terminal tree in ζ must either be **(RB then)** or **(RB not)**. This example has α=N.

punctuation semicolon/comma on their left sibling. This alternating process of reannotation can go on until (1) a conjunction word is encountered or (2) no conjunction word is encountered, i.e. a parallel structure of conjuncts delimited by semicolons or commas.

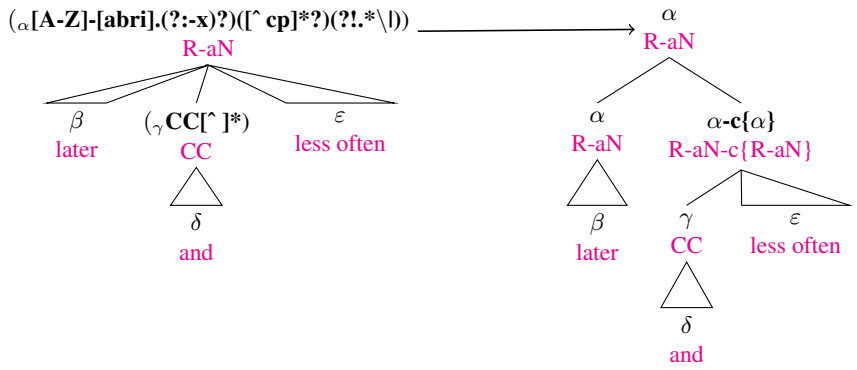


Figure 4.95: Branch off initial conjunct prior to conjunct delimiter.

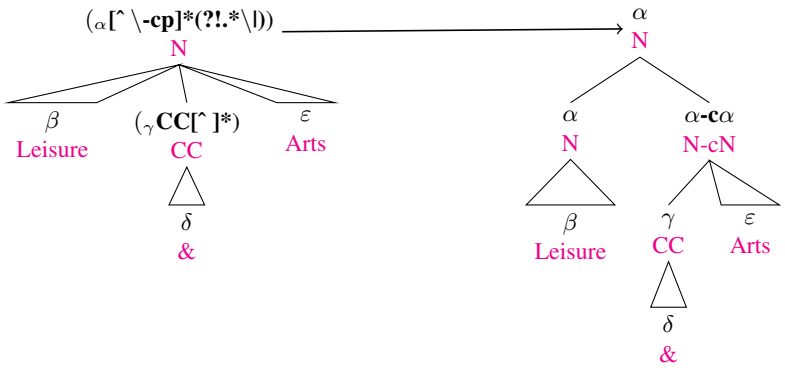


Figure 4.96: Branch off initial conjunct prior to conjunct delimiter.

If the bigger coordinating conjunction ends up at a smaller constituent covering a conjunction word connecting the last two conjuncts then this constituent must have the category composed of **-c** followed by a **-pPs** or **-pPc** due to the alternating position of **-c** and **-p** described above. The rule at Figure 4.99 will pick up this constituent to branch off the last initial conjunct α prior to the conjunction word and group the conjunction word together with the last conjunct into a common parent using the operator **-c** to link back to the last initial conjunct α. Another variation of this rule is the rule at Figure 4.100 that allows tagging along the coordinating

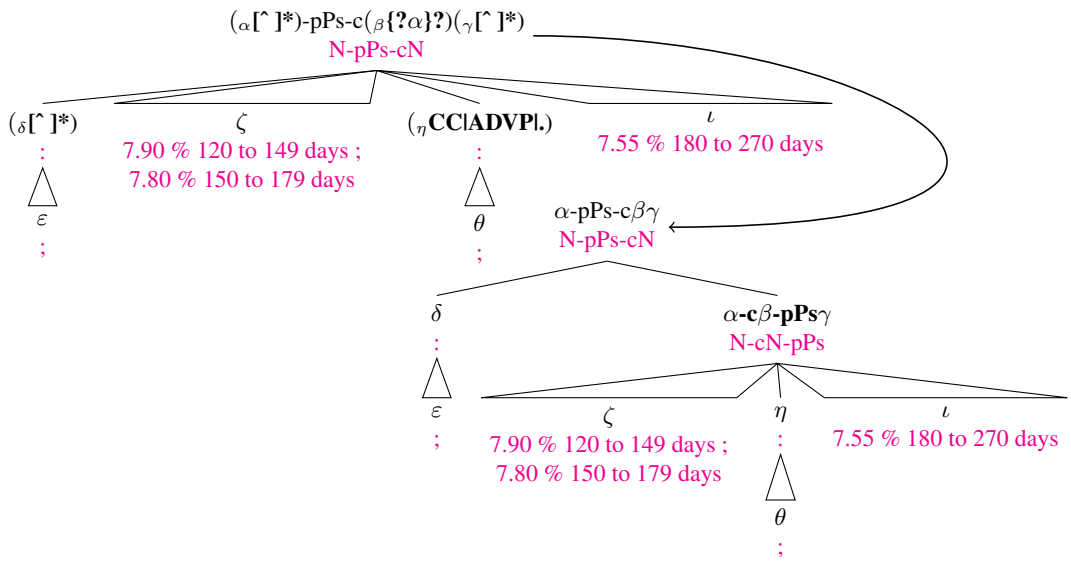


Figure 4.97: Branch off initial semicolon delimiter. β=α if α is not a composite category. Otherwise, β=$\{\alpha\}$. If η=**ADVP** then the left-most pre-terminal tree in θ must either be **(RB then)** or **(RB not)**. If η is a single character category such as **:** then it must be a pre-terminal and θ=**;**. This example has α=N, β=α=N, and γ=$\varnothing$.

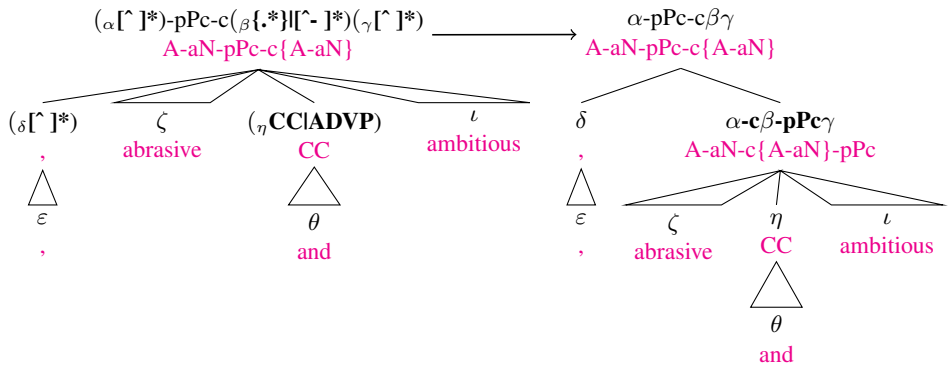


Figure 4.98: Branch off initial comma delimiter. If η=**ADVP** then the left-most pre-terminal tree in θ must either be **(RB then)** or **(RB not)**. This example has α=A-aN, β=$\{\alpha\}$=$\{$A-aN$\}$, and γ=$\varnothing$.

conjunction any non-local gap (**-g**) or right node raising (**-h**).

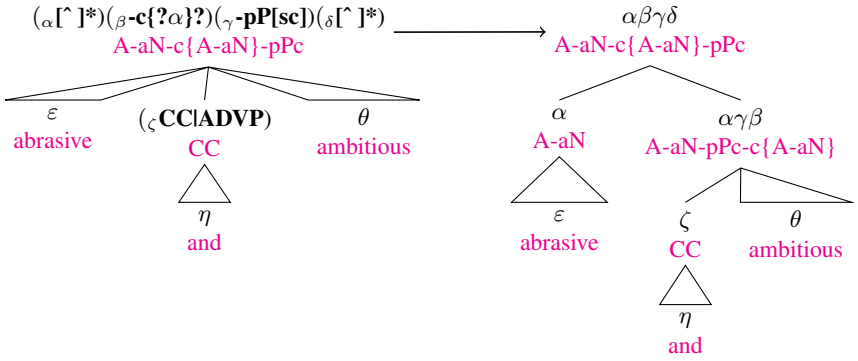


Figure 4.99: If ζ=**ADVP** then the left-most pre-terminal tree in η must either be **(RB then)** or **(RB not)**. β=**-c**α or **-c{**α**}** depending on whether α is a primitive or composite category. Subtree ε must not be empty. This example has α=A-aN, β=-c{α}=-c{A-aN}, γ=-pPc and δ=$\varnothing$.

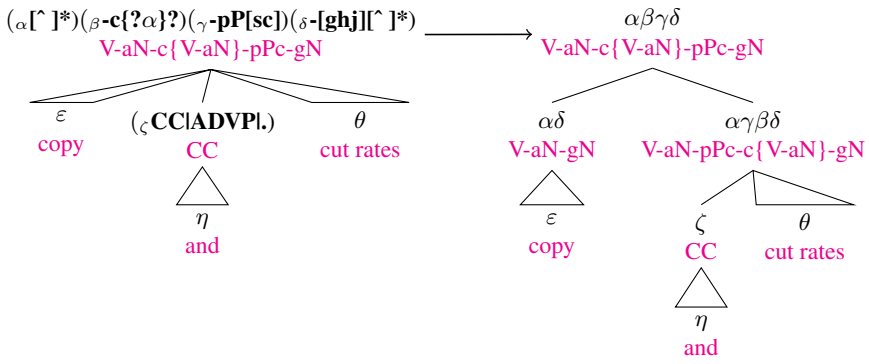


Figure 4.100: Branch off initial conjunct prior to conj delimiter (and don't pass -p down). If ζ=**ADVP** then the left-most pre-terminal tree in η must either be **(RB then)** or **(RB not)**. If ζ is a primitive category such as **:** then it must be a pre-terminal and η=**;**. β=α or **{**α**}** depending on whether α is a primitive or composite category. Subtree ζ must not be empty. This example has α=V-aN, β=-c{α}=-c{V-aN}, γ=-pPc and δ=-gN.

Similar but simpler versions of rules at Figure 4.99 and Figure 4.100 are the two rules at Figure 4.95 and Figure 4.96. These rules are for simple coordinating conjunctions having only a conjunction word connecting two conjuncts, without a comma or semicolon. Rule at Figure 4.95 is slightly more complex to support a compositional category at the parent node

while rule at Figure 4.96 is for parent of primitive category.

If the coordinating conjunction does not have a conjunction word but rather a parallel structure of conjuncts delimited by semicolons/commas, then the last semicolon/comma will be treated as a conjunction word. This is covered by the rule in Figure 4.101 for the semicolon and the rule in the Figure 4.102 for the comma.

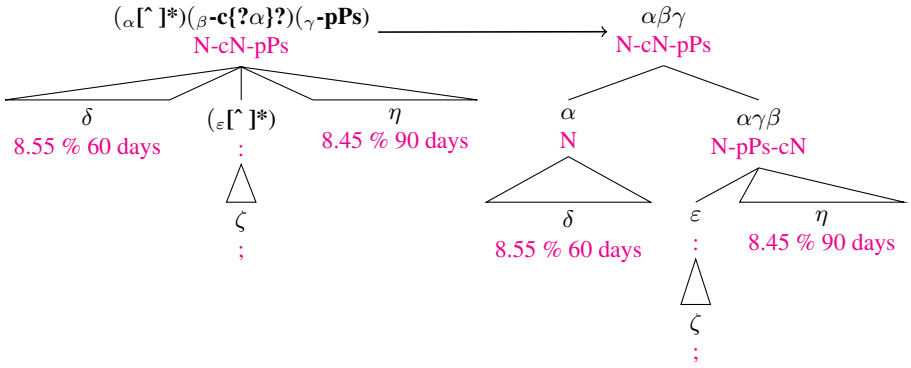


Figure 4.101: Branch off initial conjunct prior to semicolon delimiter. β=**-c**α or **-c{**α**}** depending on whether α is a primitive or composite category. Neither δ nor η could be empty. This example has α=N, β=-cα=-cN and γ=-pPs.

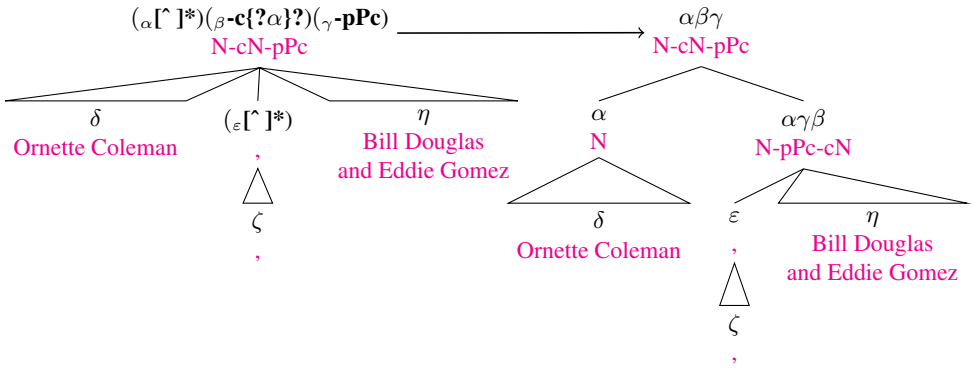


Figure 4.102: Branch off initial conjunct prior to comma delimiter. β=α or **{**α**}** depending on whether α is a primitive or composite category. Neither δ nor η could be empty. This example has α=N, β=-cα=-cN and γ=-pPc.

The last step to finish reannotating a coordinating conjunction is to introduce a final conjunct

node and rename the category of the conjunction word from a **CC** to an **X-cX-dX**. This is covered by the rule at Figure 4.103, or the one at Figure 4.104 when there is non-local gap (**-g**) or right node raising (**-h**) tagging along the coordinating conjunction being reannotated.

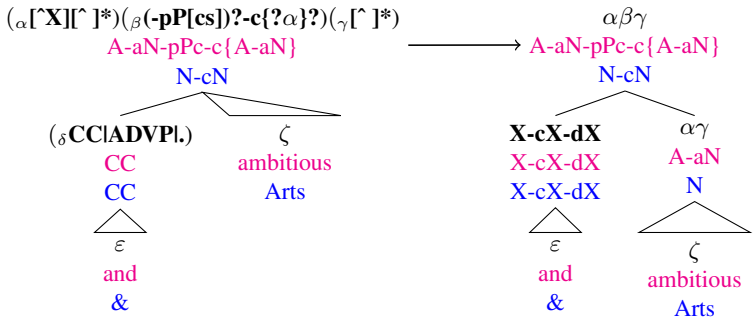


Figure 4.103: Branch off initial conjunct delimiter and final conjunct (no -p to remove). If δ=**ADVP** then the left-most pre-terminal tree in ε must either be **(RB then)** or **(RB not)**. If δ is a primitive category such as **:** then it must be a pre-terminal and ε=**;**. β=**-pP[cs]-c**α or **-pP[cs]-c{**α**}** depending on whether α is a primitive or composite category. This example has α=A-aN, β=-pPc-c{α}=-pPc-c{A-aN}, and γ=$\varnothing$.

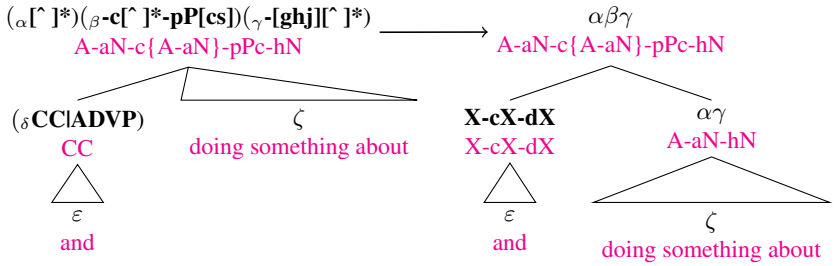


Figure 4.104: Branch off initial conjunct delimiter and final conjunct (and don't pass -p down). If δ=**ADVP** then the left-most pre-terminal tree in ε must either be **(RB then)** or **(RB not)**. This example has α=A-aN, β=-c{A-aN}-pPc, and γ=-hN.

4.4 Reannotation rules for filler attachment

The F-rules in Chapter 3 are to attach gapped clauses to modificands or relative or interrogative phrases in a variety of different constructs. This Section is about the reannotation rules to

rewrite the trees to make such constructs available. As specified in the description of the F-rules in Chapter 3, we will describe these correspondent reannotation rules below in three groups for the modificands (Fa and Fd), the relative (Fb), and the interrogative (Fc and Fe) phrases. Only some representatives of these rules will be discussed here. Otherwise, section A.3 in the Appendix has a complete reference to the filler attachment rules we used in this reannotation system.

4.4.1 Reannotation rules to apply gapped clauses to modificands (Fa and Fd)

While both Fa and Fd can be described as applying some right child gapped clause to some left child modificand, they are very different syntactically and semantically. In rule Fa, the syntax category of the gapped clause does not matter, so long as it contains a gap tag. This gapped clause contributes only to the semantic composition function. The syntax category of the parent is that of the left child modificand. This is suitable for some post nominal modifier as illustrated in Figure 4.105. For rule Fd, the category of the gap tag is constrained to be the same with the category of the modificand. The combination of the modificand and the gapped clause could produce a different syntactic category for the parent. This analysis is targeted for topicalization for the most part as shown in Figure 4.106.

4.4.2 Reannotation rules to apply gapped clauses to relative phrases (Fb)

Figure 4.107 shows a copy of the Fb rule. This rule is to first introduce the gap tag into the right tree branch on the same level with the left branch which is the filler. The gap tag is then progressing down on the right branch until it stops at the gap constituent to complete a chain from the filler to the gap, modeled as a series of local parent/child dependencies. The rule at Figure 4.107 make it possible to create the first step of this chain by matching the index on the filler with that of the null trace element. This helpful trace information on PTB is commonly ignored by other systems.

4.4.3 Reannotation rules to apply gapped clauses to interrogative phrases (Fc and Fe)

The syntactic constraint setup for the left branch interrogative phrase and the right branch gapped clause is the same for both Fc and Fe rules, i.e. the gap category of the right branch is the

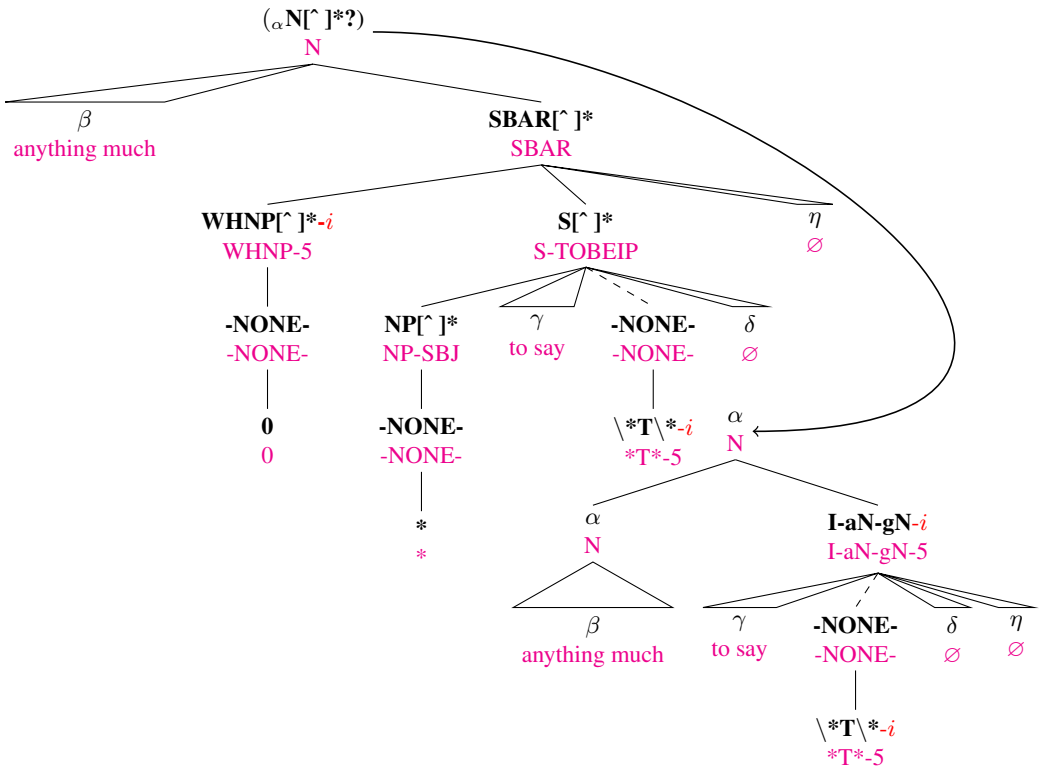


Figure 4.105: Branch off final **SBAR** as modifier **I-aN-gN**. This is an Fa rule **N** + **I-aN-gN** = **N**.

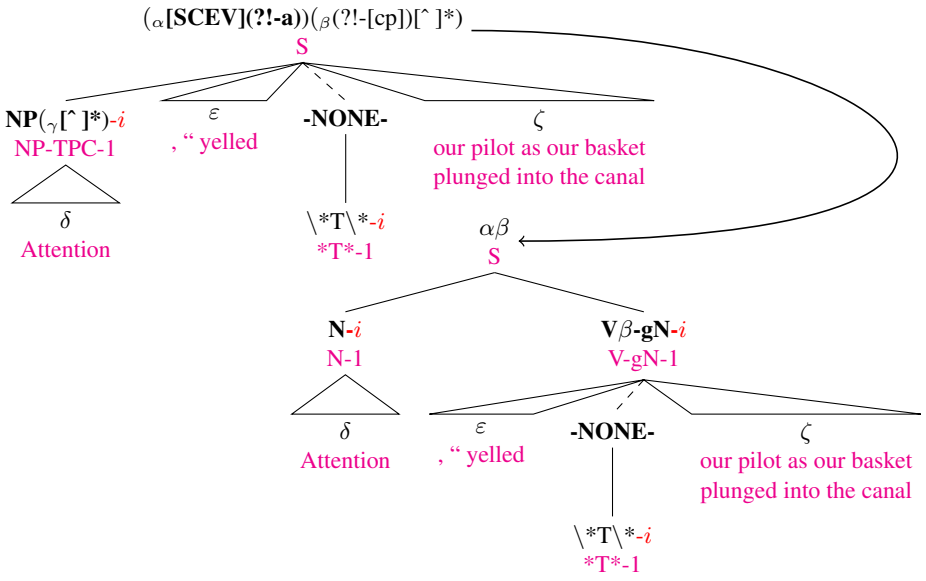


Figure 4.106: Topicalized sentence: branch off initial topic **N**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules.

same with the main part of the category of the left branch interrogative phrase. The difference comes in the way the parent category is determined. Fc chooses to compose the category of the parent in the normal way as if gap is hypothesized as an initial argument. This composition is suitable for the analysis of embedded questions as shown in Figure 4.108.

Fa on the other hand chooses the category of the interrogative to be the one for the parent. This is suitable for the analysis of nominalization to create some bigger nominal phrase as shown in Figure 4.109.

4.5 Reannotation rules to hypothesize gap as initial argument, final argument, or modifiers

Section 4.4 discussed rules that introduce a gap tag **-g** into a tree. This gap tag is normally accompanied by some index copied from the filler node. When it comes the time for a parent node with a gap tag to be inspected for a rewrite, there are rules that copy this gap tag to the child (left or right) containing trace information with the same index. This process recursively

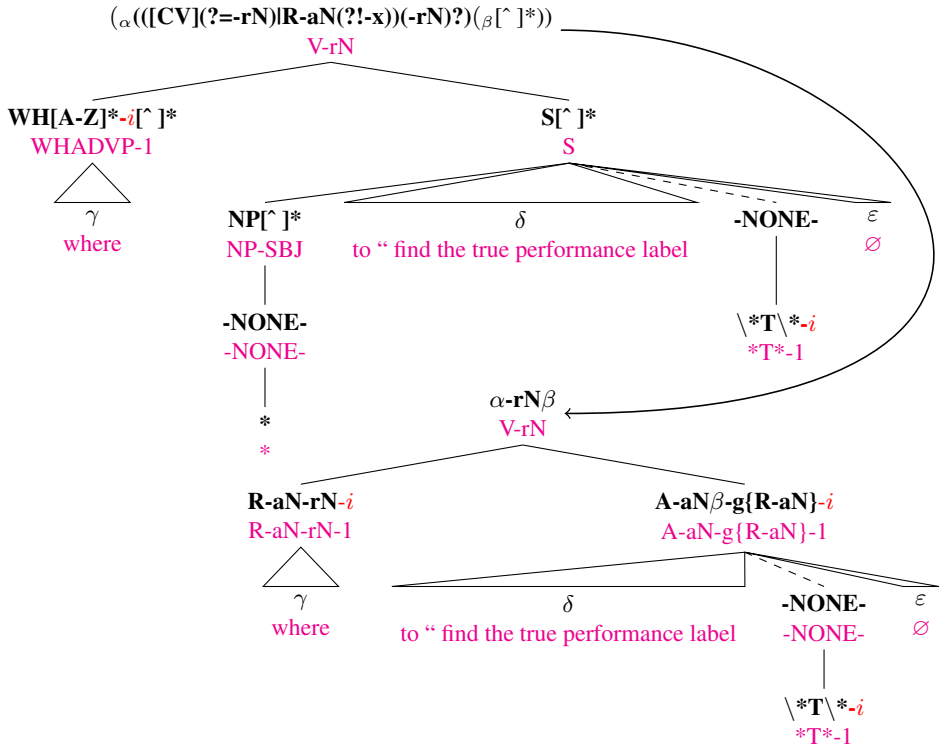


Figure 4.107: Branch off initial relative adverbial phrase with empty subject ('when in rome'). This example has α=V-rN, β=$\varnothing$ and i=1. This is an Fb rule **R-aN-rN + A-aN-g{R-aN} = A-aN-rN** followed by a type changing rule to change an **A-aN** to a **V**.

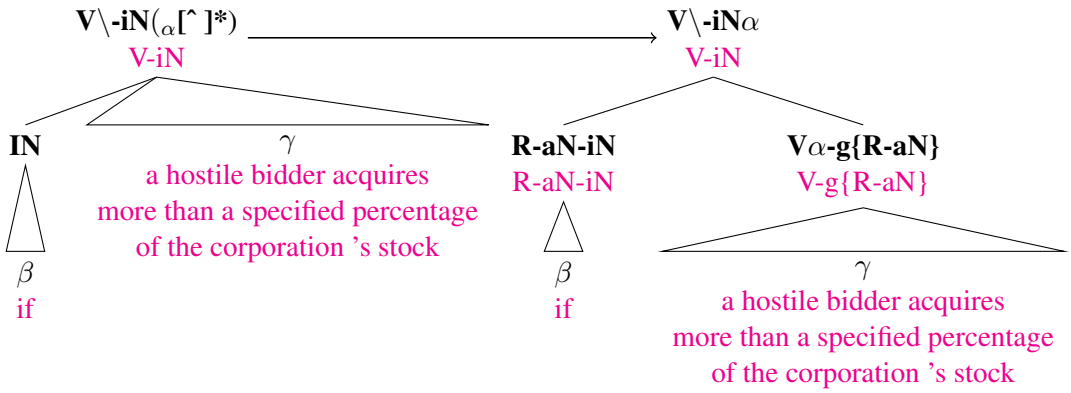


Figure 4.108: Embedded question: branch off initial interrogative **R-aN** of *whether* or *if*. This example has α=$\varnothing$. This is one of the Fc rules.

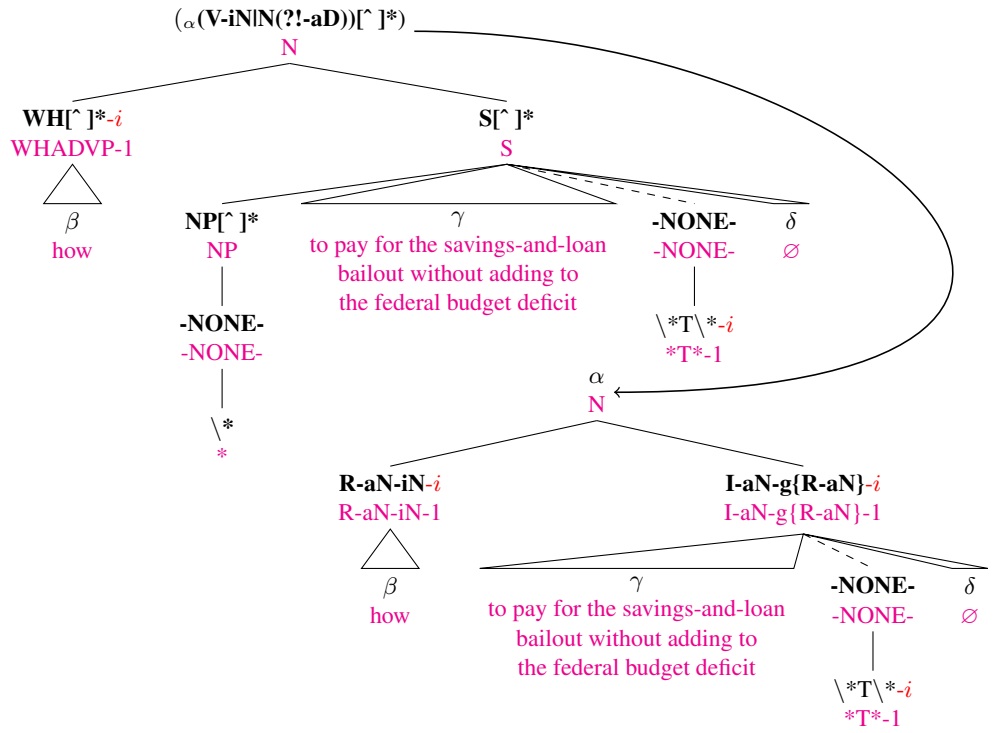


Figure 4.109: Embedded question / nom clause: branch off initial interrogative **R-aN** and final modifier **I-aN** with **R-aN** gap

creates a path of gap tags connecting the filler (e.g. a relative pronoun) to the gap (e.g. a noun phrase syntactically connected with the filler) where the attachment can take place.

4.6 Reannotation rules for relative pronoun attachment (-r)

There are only two R rules in Chapter 3 to attach pronominal relative clauses to their modificands depending on whether the modificand is ahead (Ra) or behind (Rb) the relative clause, and there is only one combination **-rN** because operator **-r** can only occur with an **N**. But there are a couple different known syntactic categories of the pronominal relative clauses.[2]

Among these modifier categories, a finite verbal **V-rN** and a complementized finite **C-rN** are found most common. For the modificand, the most common category is **N** (e.g. *[the executives] [who are really calling the shots]*), but any verbial category is entirely possible. For example, Figure 4.112 shows an **A-aN** for phrase *[offering a gift] [when consumers make a purchase]* or a **V-aN** is shown in the example of Figure 4.113 for phrase *[came in for some blocks in the secondary market ,] [which we have n't seen for awhile]*. The modificand can also be a sentential category as illustrated in the rule at Figure 4.114 for the example *[when market interest rates move up rapidly] [, increases in bank CD yields sometimes lag]*. For this analysis of English, a sentential modificand can stand before of after the pronominal relative clause, e.g. using either rule Ra or Rb, and a swap of the left and right child of the example in Figure 4.114 yield a grammatically valid sentence of the same meaning, but a nominal or verbial modificand seem to always happen before the pronominal relative clause (i.e. using only rule Ra). Because the modificand is the head of the phrase after the attachment, the syntactic category of the parent is that of the modificand.

The rules to reannotate PTB trees into this analysis of pronominal relative attachment depend on the parent having category **N**, or any verbial or sentential, that has the right child (rule Ra) or the left child (rule Rb) exposing some signal of being a relative clause (e.g. starting with a *wh*-word, having some **WH[^]** category in the case of reduced relative clause, or having the **WH[^]** category under some **SBAR** that marks a sentential modifier) to rewrite them into a modificand having the same category as the parent, and a pronominal relative clause of type **C-rN** or **V-rN** depending on the structure of the original relative clause.

[2] We had thought about doing **-rV** for finite clause modifier but haven't tried it yet.

4.6.1 Relative pronoun attachment for nominal phrase

A relative or pronominal reduced relative clause was annotated as **SBAR** in PTB. While Figure 4.110 shows the rule to detect a *wh*-word, the rule in Figure 4.111 only looks for a category **WH[^]** that possibly covers a null element so that we do not miss the case for reduced relative clauses. In all these cases, the relative pronominal clause is consistently analyzed as a complementized finite **C-rN**.

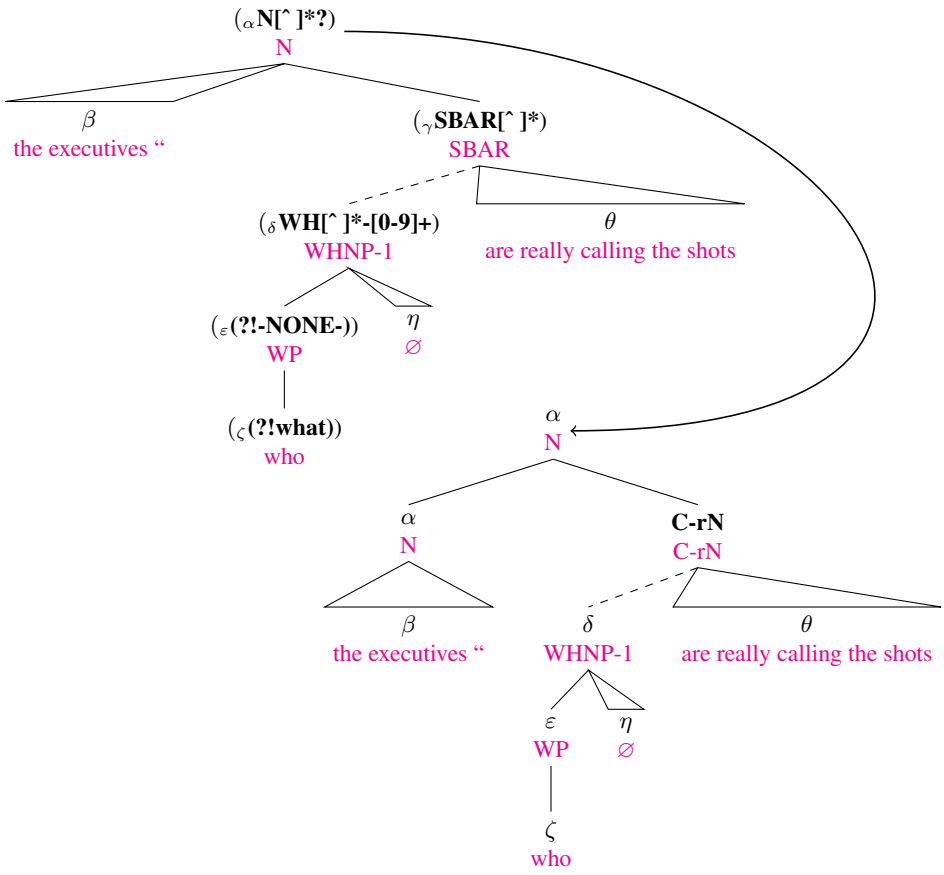


Figure 4.110: Branch off final **SBAR** as modifier **C-rN**. This is a relative pronoun attachment rule Ra.

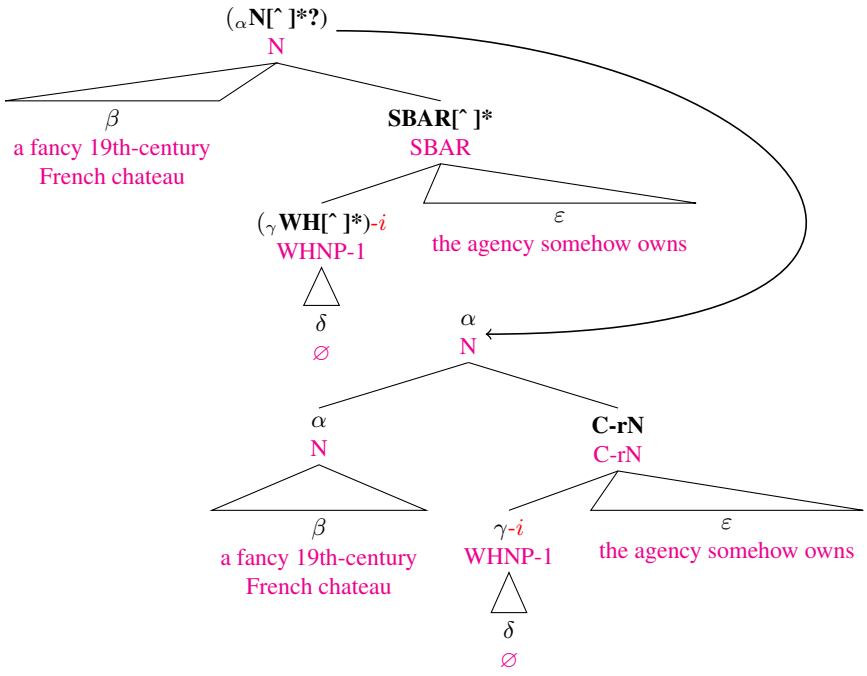


Figure 4.111: Branch off final **SBAR** as modifier **C-rN** (that|nil). This is a relative pronoun attachment rule Ra.

4.6.2 Relative pronoun attachment for verb phrase

One thing different about pronominal relative clause attachment for a verb phrase compared to the one for a nominal phrase is that the category of the pronominal relative clause is a finite verbal **V-rN** instead of a complementized finite **C-rN**. Another crucial difference is that the attachment must have a pronoun, so there is no reduced relative allowed.

With that said, the rules for relative pronoun attachment for verb phrase look very similar to the ones for nominal phrase in the way the pronominal relative clause is detected. For example, the matching condition of the right child of Figure 4.112 for the verb phrase looks similar to that of Figure 4.110 (one looks for any relative not a *what* and the other looks for *where* or *when*) for the nominal phrase. Similarly, the matching condition of the right child on Figure 4.113 for verb phrase looks almost the same as that of Figure 4.111 of the nominal phrase; it is only different in that it requires a relative pronoun *which* because no reduced relative is allowed in this case.

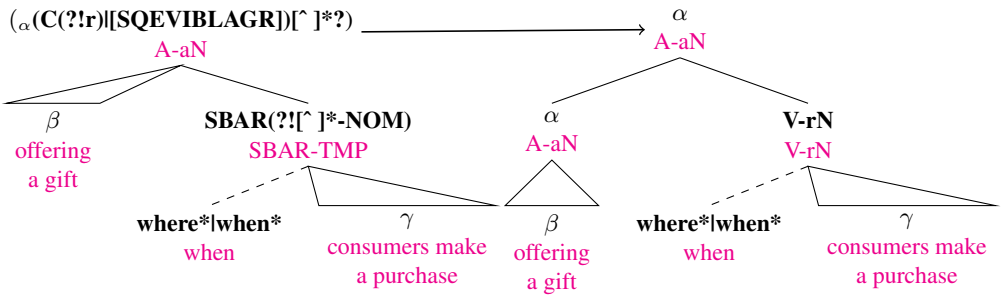


Figure 4.112: Branch off final **SBAR** as modifier **V-rN**. This is a relative pronoun attachment rule Ra.

4.6.3 Relative pronoun attachment for sentential phrase

One thing different about pronominal relative clause attachment for a sentential phrase compared to the one for a nominal or verbial phrase is that this one can have the relative clause as the left or right child. When the relative clause is the right child, it used rules Ra, e.g. the one at Figure 4.113 and when the relative clause is the left child, it used rules Rb, e.g. the one at Figure 4.114. The category of the pronominal relative clause seems to be only finite verbal **V-rN** instead of a complementized finite **C-rN** or a finite verbal **V-rN**. Another crucial difference is

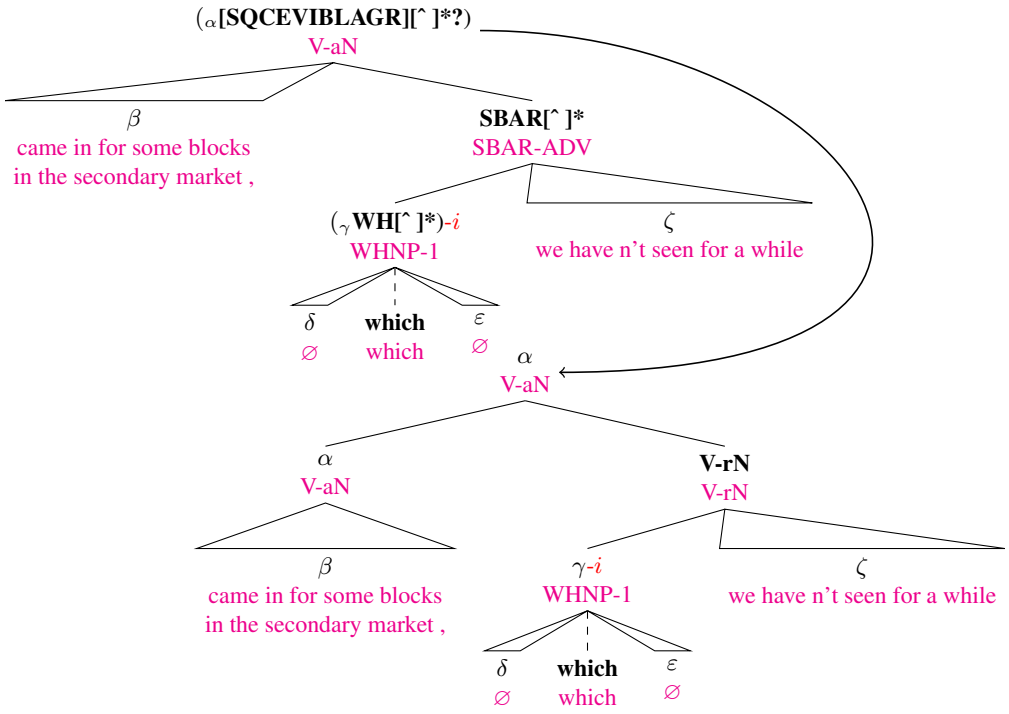


Figure 4.113: Branch off final **SBAR** as modifier **V-rN**. This is a relative pronoun attachment rule Ra.

that the attachment must have a pronoun, so there is no reduced relative allowed.

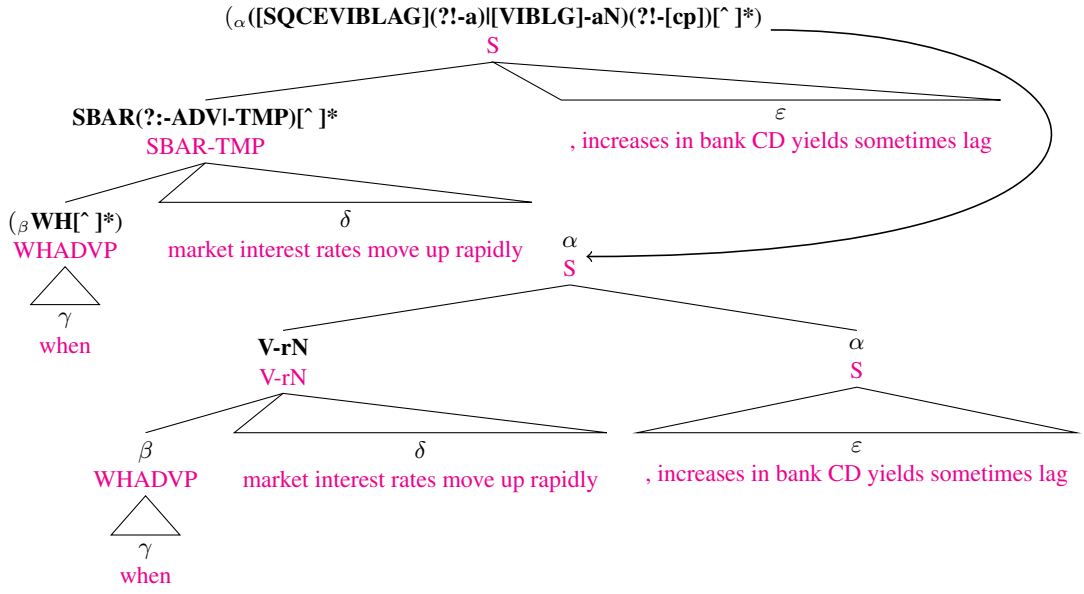


Figure 4.114: Branch off initial modifier **V-rN** from **SBAR-ADV** or **SBAR-TMP**. This is a relative pronoun attachment rule Rb.

4.7 Reannotation rules for argument elision (-a/-b)

The category for a common noun is usually **N-aD** to denote that they normally have an initial argument which is a determiner **D** to make up a noun phrase **N**, but sometimes a noun can be found without a determiner. To cope with this irregularity, we need a rule to simply transform an **N-aD** to an **N**, or simply eliding the **-aN** from the category. Other scenarios could be some verbs that can be used as a transitive verb **V-aN-bN** or as an intransitive verb **V-aN**, e.g. *drive*, *do*, *play*, etc. In this case, we need a rule to elide the **-bN** from the normal transitive context to make up the intransitive one. Whenever possible, we embedded elision rules into the syntactic analysis of other rules for brevity.

4.8 Reannotation rules for right node raising (-h)

PTB trees use a special node *RNR* with an index to denote right node raising, but we do not always agree with their analysis of what right node raising is. For example, *[availability] and [raising cost] of insurance* was annotated with *RNR* in PTB, but we see it as an adjectival modifier phrases **A-aN** that modifies a conjunction of two conjuncts instead, as seen in Figure 4.115. Note that we do not use operator **-h** in this case.

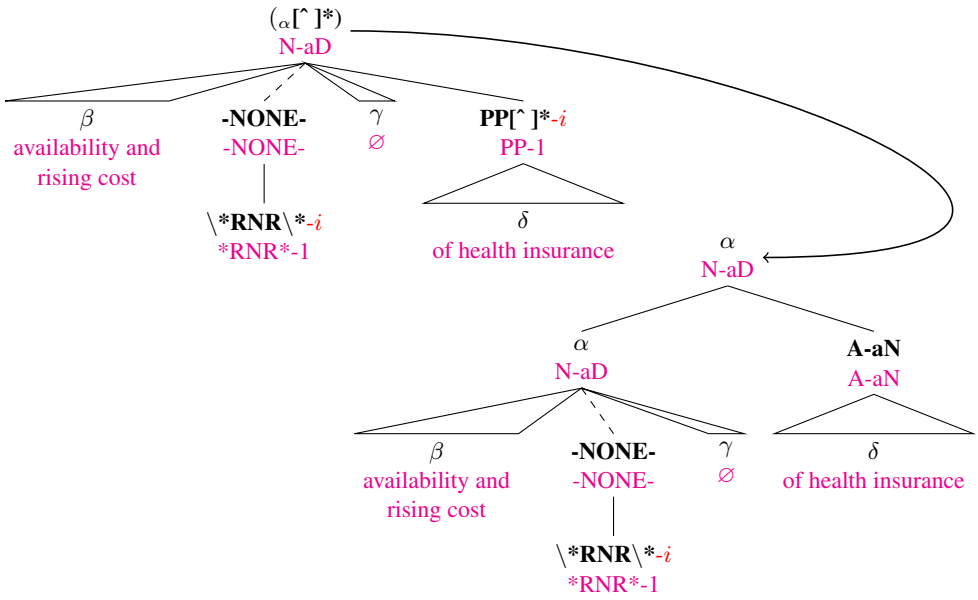


Figure 4.115: Branch final right-node-raising modifier **A-aN**. This example has α=N-aD.

When we do agree with PTB analysis, we use the operator **-h** to mark a right node raising as seen in Figure 4.116. The key to this agreement or disagreement of right node raising is at the category of the right periphery: the **NP** is but the **PP** is not.

4.9 Reannotation rules for type changing

Any grammar has the need for some form of type-changing rules to cope with language specific features in a more regular formalism. As seen in Hockenmaier and Steedman (2007), the CCG reannotation also used quite a few type changing rules to reduce the number of lexical category

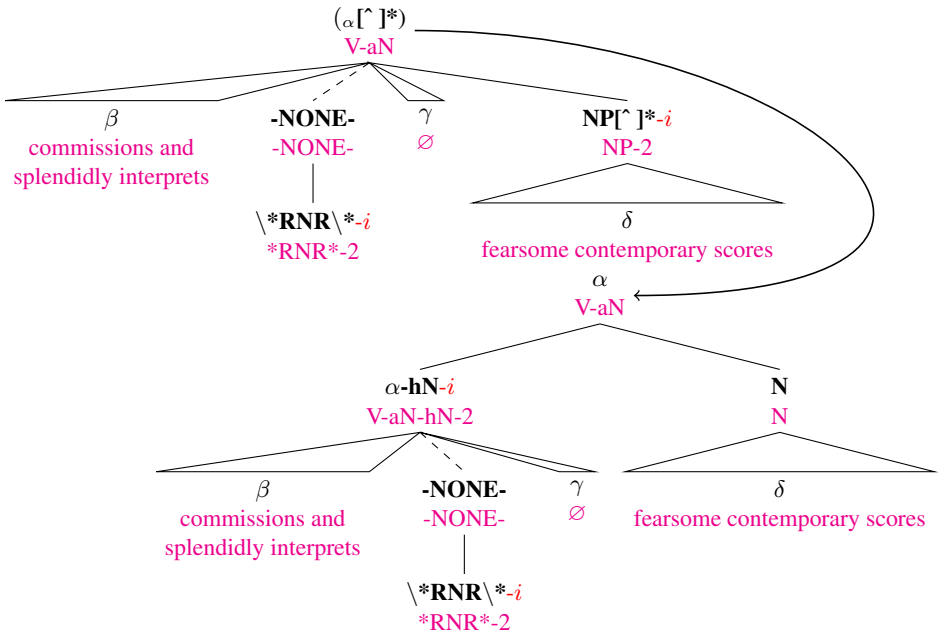


Figure 4.116: Branch final right-node-raising complement **N**. This example has α=V-aN.

types required to model complex adjuncts, NP-extraposition for arbitrary types of predicative noun phrase, etc. We need some type changing rules, too.

Sometimes we combined the type changing with other syntactic analysis instead of having their own rules just for brevity. For example, one of the rules for initial modifier at Figure 4.57 also shows a type changing from an **I-aN** to an **R-aN** on the left branch. We know by looking at the PTB category **TO** that this is an infinitive clause, hence should be annotated with an **I-aN**, but this clause is analyzed as an initial modifier of the right branch, so it should be an **R-aN**. Another example is shown at Figure 4.117 where the PTB category extension of **-NOM** is a good indication that it should be an **N**, but it is a final modifier **A-aN** in this analysis, hence an embedded type changing rule comes to change from **N** to **A-aN**.

At other times we may have separate rules just to deal with the type changing analysis. For example, Figure 4.118 shows a rule to change a **B-aN** to an **S**. We know it should be a base-form verbal phrase **B-aN** based on the PTB category **VB**, but it could also be an imperative sentence **S** if that is everything it has (i.e. no subject). Section A.4 in the Appendix has a complete reference to the type changing rules we used in this reannotation system.

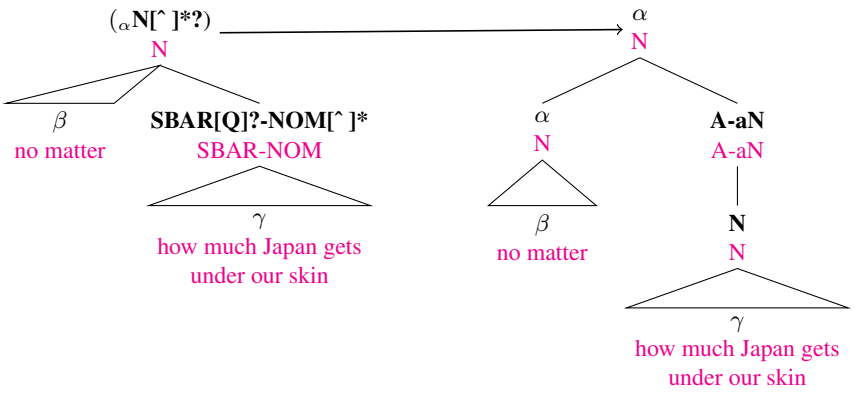


Figure 4.117: Branch off final **SBAR** as modifier **A-aN** then **N** (nominal clause): Direct children of α in β must not be any **CC**.

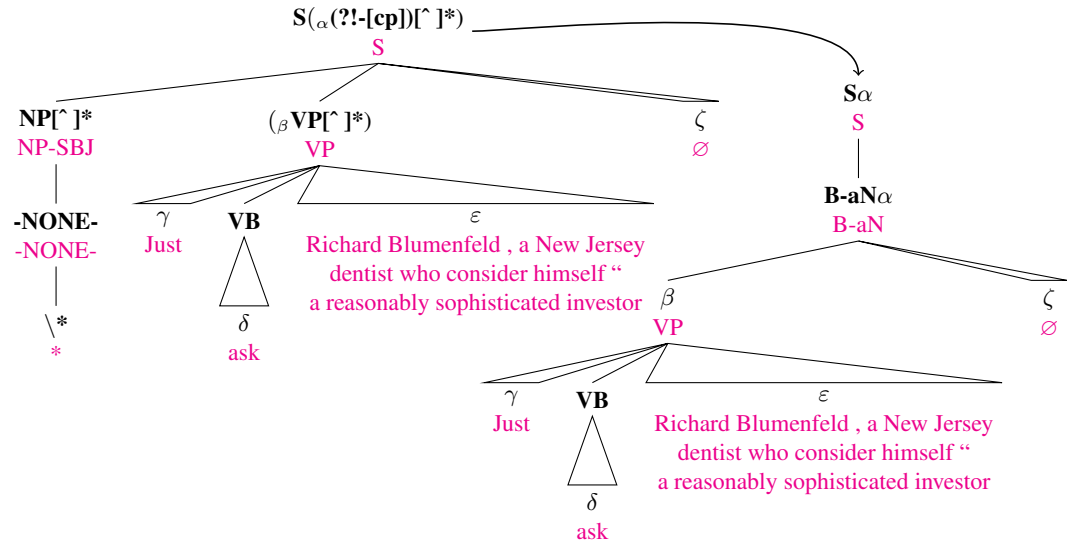


Figure 4.118: Imperative sentence: delete empty **NP**. Top level node in δ must not be a **VP** head, i.e. one of (**VB**, **JJ**, **MD**, **TO**)

4.10 Preprocessing Penn Treebank trees

PTB trees are inconsistent due to their being manually annotated. To fully automate the reannotation process, we must first normalize these inconsistencies in a preprocessing step. This preprocessing step also does some head percolation to reveal the needed tree structure information internally in the tree up to its root to help guide the starting point of the reannotation process walking down the tree.

4.10.1 Normalize PTB inconsistencies

Treebank trees use null-subjects as traces to link back to functional subjects. It gets more complicated when it is not a null-subject-**S** but a conjunction of multiple null-subjects. Our GCG grammar does not have null elements. The conversion process will eventually remove them all. In this preprocessing step, we convert conjunctions of null-subject-**S**'s into a null-subject-**S** with a conjunction of **VP**'s as shown in Figure 4.120. We also turn the modifier of null-subject-**S** followed by a **VP** into a null-subject-**S** with a modifier of the **VP** as shown in Figure 4.119

4.10.2 Head Percolation

As apposed to the top-down reannotation, head percolation is a bottom up process. The head percolations we did only focus on the **VP**-head and **S**-head. Treebank trees use the verbal category **VP** to denote a top-level verb phrase, and its immediate children such as **MD** (modal), **TO** (infinitive), **VB** (base-form, non-third person), **VBZ** (base-form, third person), **VBG** (gerund), **VBD** (past tense), **VBN** (participial) further denote more specific types of verb phrases. Our GCG does not have two levels of verb phrase like that, so we need to percolate the Treebank tree **VP** to reflect its specific verbal type before the reannotation can start. Our GCG has seven different primitive verbal categories: **V** (finite verbal), **I** (infinitive verbal), **B** (base-form verbal), **L** (participial verbal), **A** (adjectival or predicative), **R** (adverbial), and **G** (gerund), but not all of these seven types can be easily identified as a proper mapping from the Treebank tree. For example, a Treebank **VBN** could be equivalent to an **L** but more likely to be an **A**. This is why Figure 4.124 shows that we percolate the head to a **VP-TOBEAP** instead of a **VP-TOBELP**.

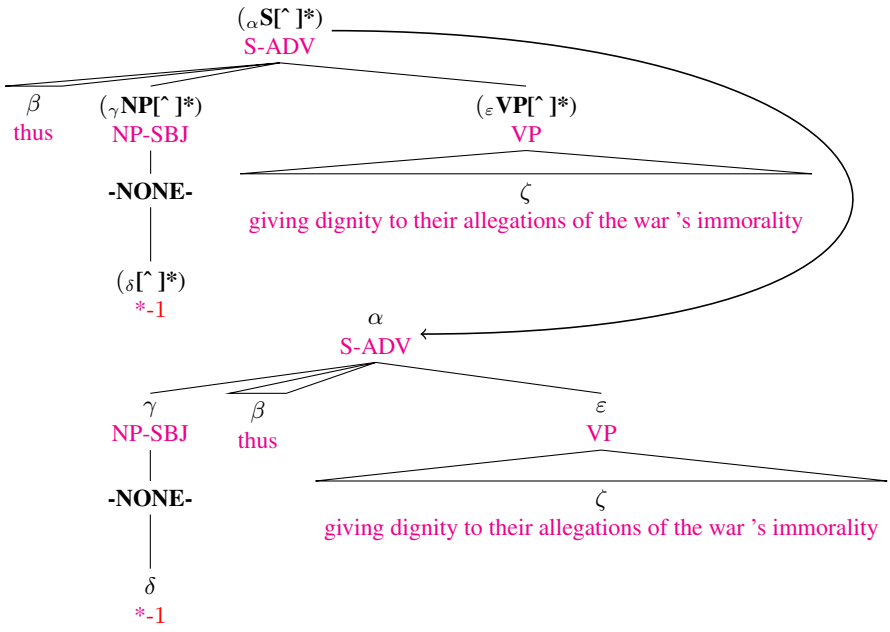


Figure 4.119: Transform a modifier of null-subject-**S** into a null-subject-**S** with a modifier of **VP**.

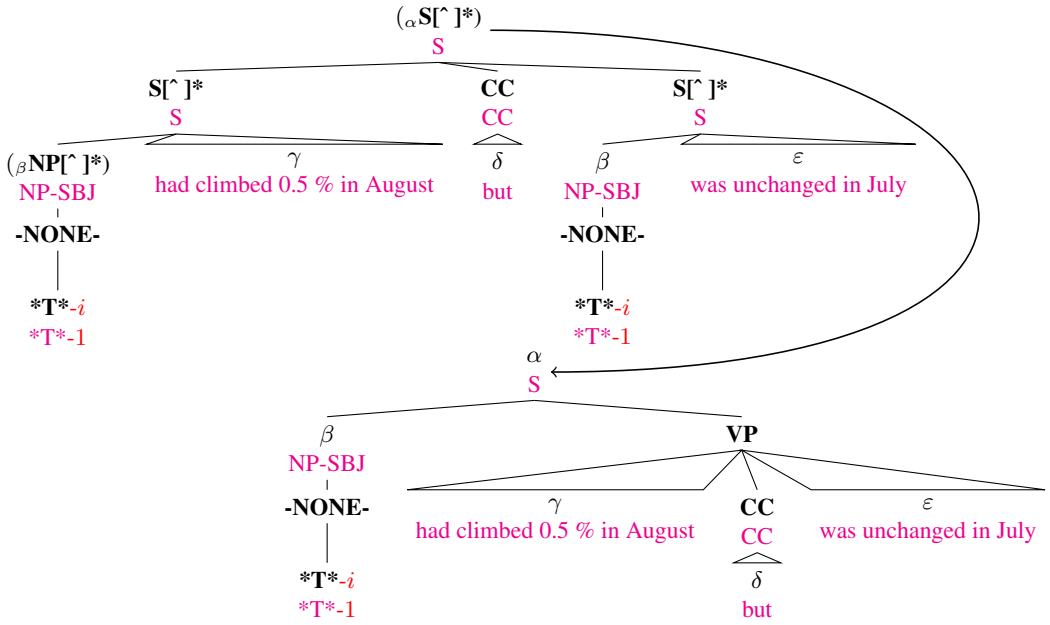


Figure 4.120: Transform a conjunction of multiple null-subject-**S**'s into a null-subject-**S** with a **VP**-conjunction.

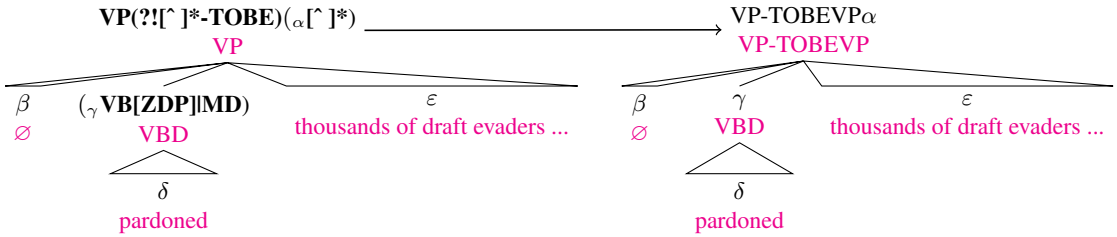


Figure 4.121: Percolate a **VP-TOBEVP**-head if this head has a direct child of category **VB[ZDP]** or **MD**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**.

Percolating VP

We temporarily appended **-TOBE*x*P** with *x* be one of **V**, **I**, **B**, **L**, **A**, **G**, into the PTB's **VP** based on the type of its child's head as follows. Note that we cannot reliably determine a **-TOBELP** at this point, but that will come later into the reannotation process as we know more about the structure of the tree. We don't need **-TOBEGP** as that verbal phrase is more likely to be transformed into an **A** instead.

1. If **VP** child's head is a **VBZ**, catVBD or **VBP** then *x*=**V** (Figure 4.121).
2. If **VP** child's head is a **TO** then *x*=**I** (Figure 4.122).
3. If **VP** child's head is a **VB** then *x*=**B** (Figure 4.123).
4. If **VP** child's head is a **VBG** or **VBN** then *x*=**A** (Figure 4.124).
5. Propagate the **-TOBE*x*P** up to its parent **VP** if it is part of a conjunction as the left conjunct (Figure 4.125) or as the right conjunct (Figure 4.126).
6. Propagate the first child **-TOBE*x*P** up to the parent **VP** (Figure 4.127).
7. Snap a **-TOBEVP** to a **VP** when the only child it has is a null element (Figure 4.128).
8. Snap a **-TOBEAP** to a **VP** if it does not have any child of category **VP.*** (Figure 4.129).

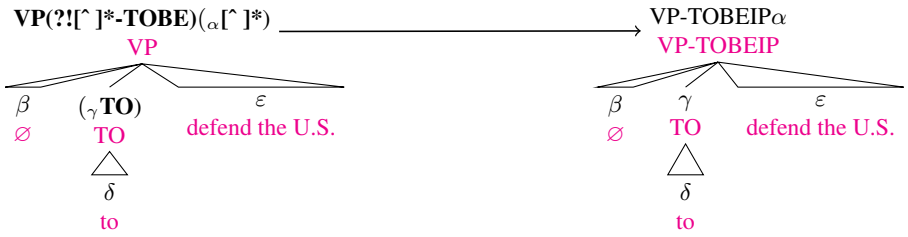


Figure 4.122: Percolate a **VP-TOBEIP**-head if this head has a direct child of category **TO**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**.

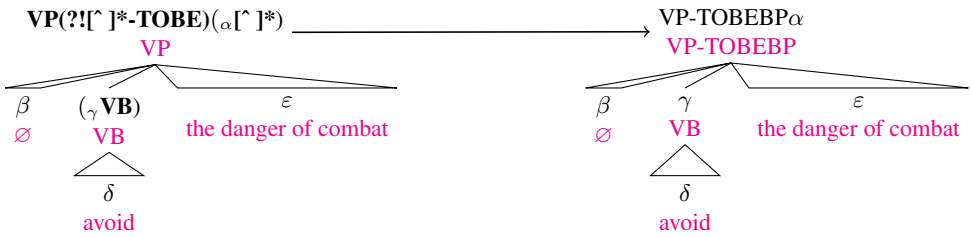


Figure 4.123: Percolate a **VP-TOBEBP**-head if this head has a direct child of category **VB**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**.

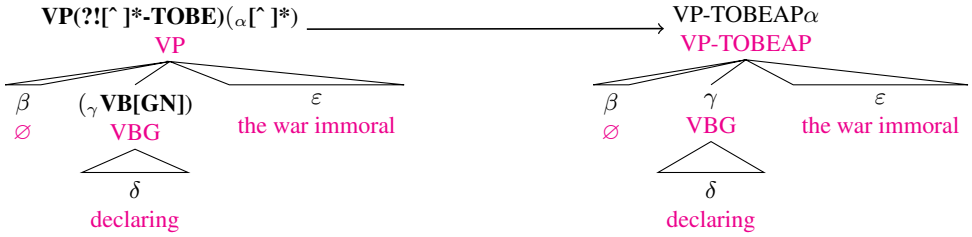


Figure 4.124: Percolate a **VP-TOBEAP**-head if this head has a direct child of category **VB[GN]**. The top-level nodes in the branch β must not contain any **VB.*** or **TO**.

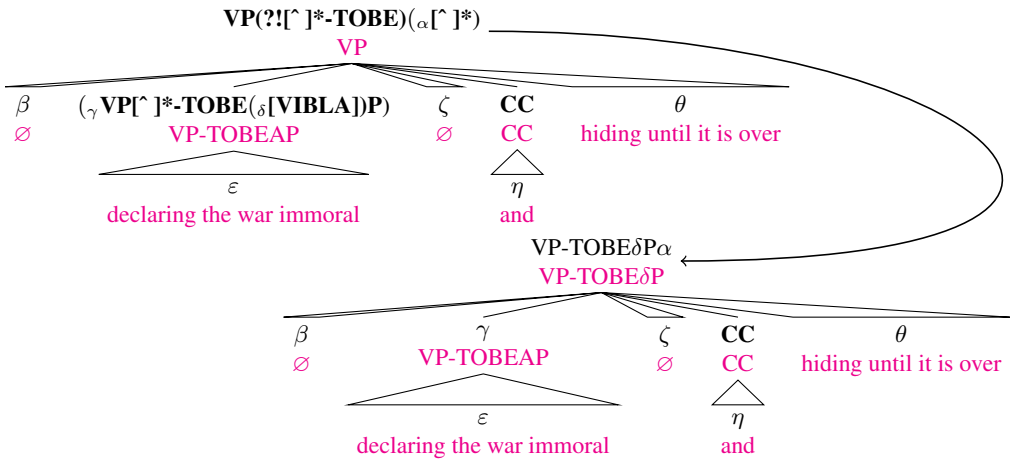


Figure 4.125: Percolate a **VP-TOBE*x*P**-head if this head is the head of a conjunction and the left conjunct has been annotated with a **VP-TOBE*x*P** already. This is a variation of Figure 4.126 where it looks at the left conjunct instead of the right one.

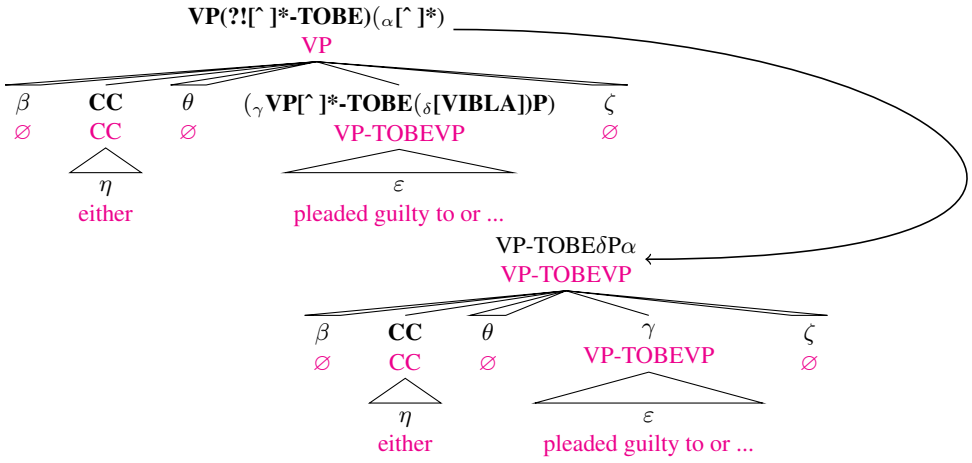


Figure 4.126: Percolate a **VP-TOBE*x*P**-head if this head is the head of a conjunction and the right conjunct has been annotated with a **VP-TOBE*x*P** already. This is a variation of Figure 4.125 where it looks at the right conjunct instead of the left one.

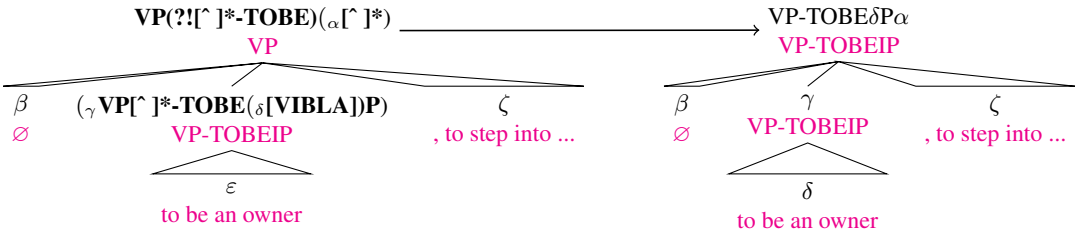


Figure 4.127: Percolate a **VP-TOBE*x*P**-head if this head has a direct child **VP-TOBE*x*P**-head already.

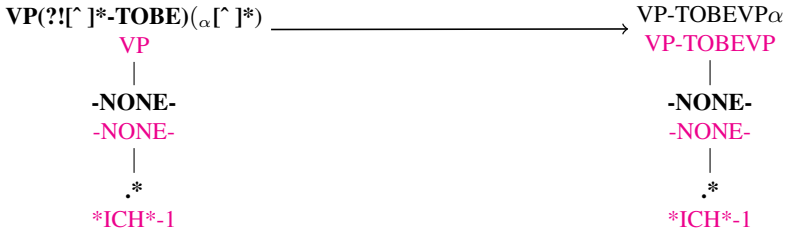


Figure 4.128: Percolate a **VP-TOBEVP**-head if this head has only one child which is a null element.

Figure 4.129: Percolate a **VP-TOBEAP**-head if this head has no direct child of category **VP.***. This means the β in this rule has not top-level node of type **VP.***.

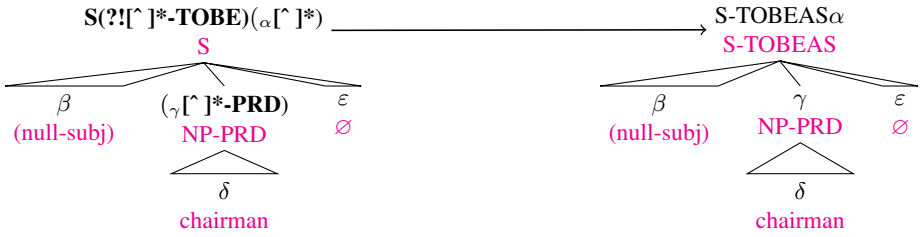


Figure 4.130: The null-subj usually is **(NP-SBJ (-NONE- .*))**. Percolate an **S-TOBEAS**-head if it has a direct child of category **[^]*-PRD** as this is the PTB marker of "predicative".

Percolating S

Similar to **VP**, **S**-head is also appended with **-TOBE*x*S** with *x* be one of **V**, **I**, **B**, **L**, **A**, depending on the now recognized type of its **VP-TOBE*x*P** child or some other **S-TOBE*x*S** child. Below are the rules for these percolations.

1. Append a **-TOBEAS** to the **S** that has its child head of **.*-PRD** as this is the PTB marker of 'predicative' (Figure 4.130).

2. Append a **-TOBE*x*S** to an **S** if it has a null-subject child followed by a **VP-TOBE*x*P** child (Figure 4.131).

3. Append a **-TOBE*x*S** to an **S** if it has a **VP-TOBE*x*P** child (Figure 4.132).

4. Propagate the **-TOBE*x*S** up to its parent **S** if it is part of a conjunction as the left conjunct (Figure 4.133) or as the right conjunct (Figure 4.134).

5. Append a **-TOBE*x*S** to an **S** if it has an **S-TOBE*x*S** child (Figure 4.135).

6. Append a **-TOBEVS** to an **S** if the only child it has is a null-subject (Figure 4.136).

7. Append a **-TOBEAS** to an **S** if none of the child it has is a **VP.*** (Figure 4.137).

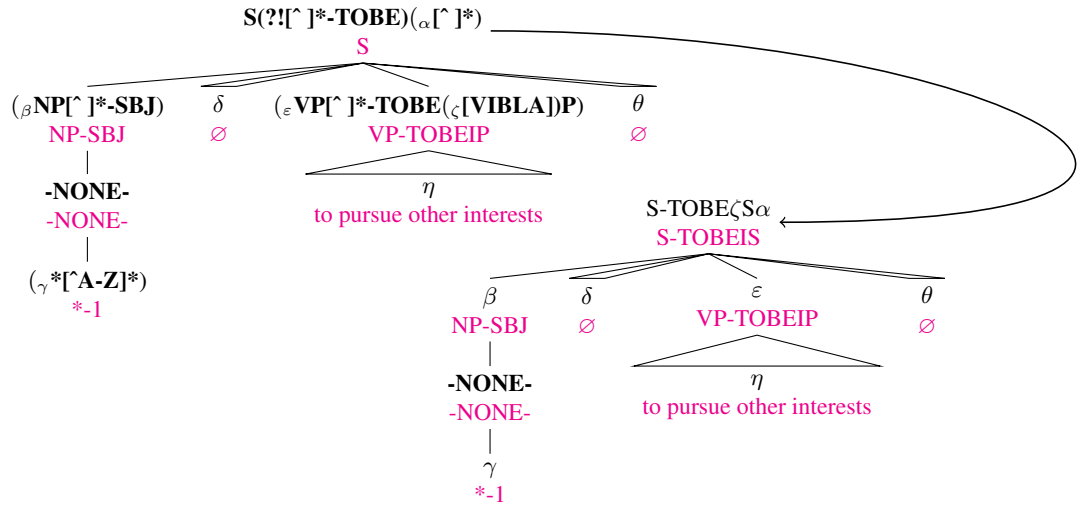


Figure 4.131: If **S** covers a null-subject and a **VP-TOBE*x*P** then append a **-TOBE*x*S** to it. This is a more specific rule of the one in Figure 4.132 where we don't even need a null-subject.

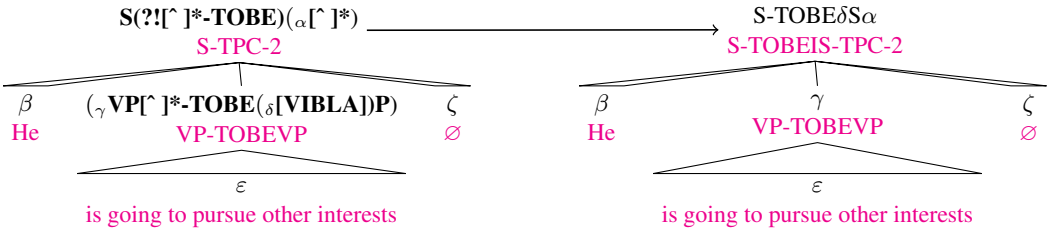


Figure 4.132: If **S** covers a **VP-TOBE*x*P** then append a **-TOBE*x*S** to it. This is a more general version of the rule in Figure 4.131.

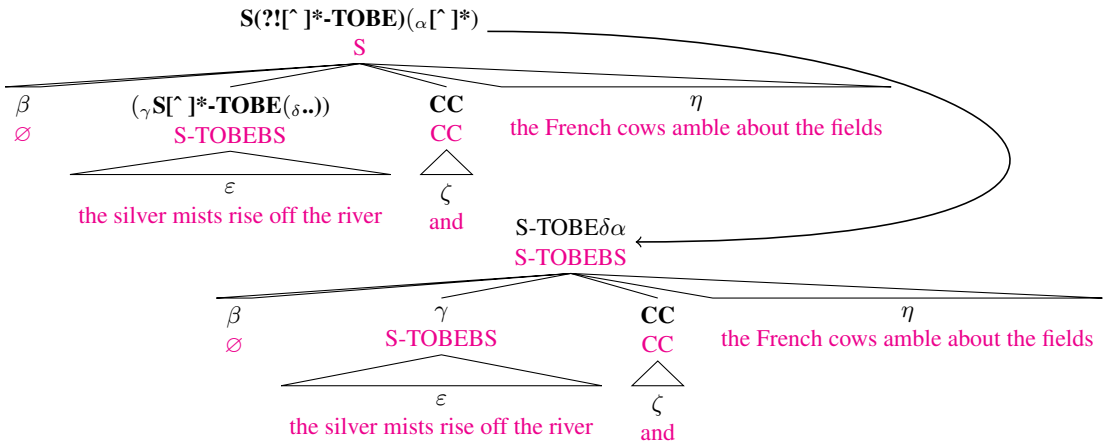


Figure 4.133: If **S** covers a conjunction and one of the conjunct was already percolated with a **-TOBE*x*S** then append a **-TOBE*x*S** to it. This rule has the conjunct as a left one. Figure 4.134 as a similar version of this rule but for the right conjunct.

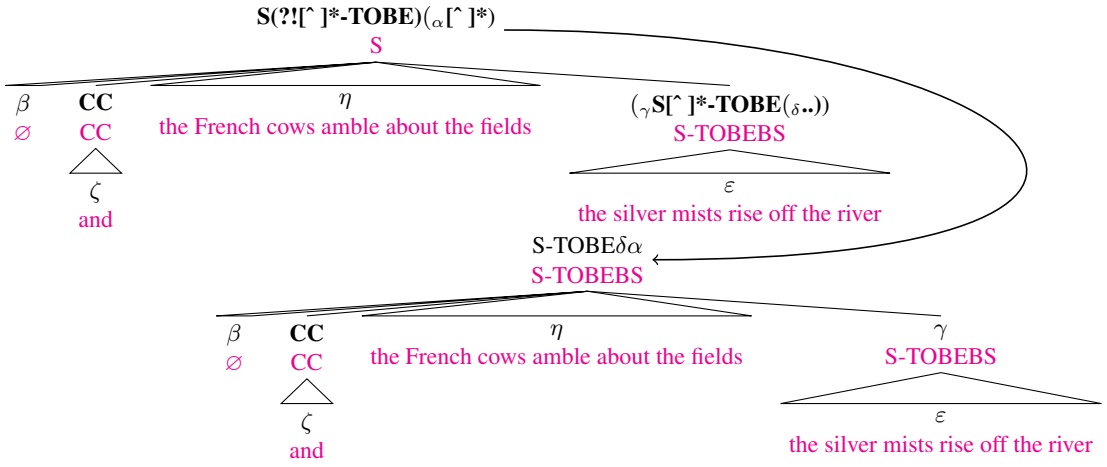


Figure 4.134: If **S** covers a conjunction and one of the conjunct was already percolated with a **-TOBE*x*S** then append a **-TOBE*x*S** to it. This rule has the conjunct as a right one. Figure 4.133 as a similar version of this rule but for the left conjunct.

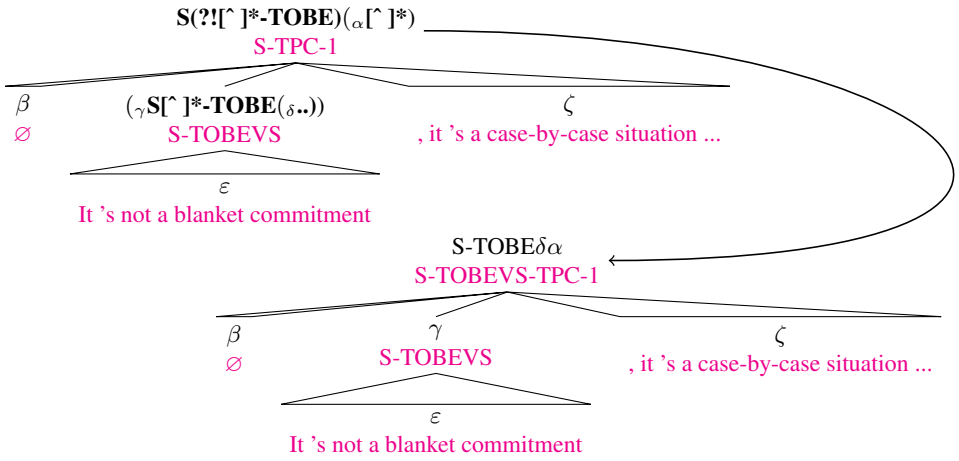


Figure 4.135: If **S** covers another **S** that was already percolated with a **-TOBE*x*S** then keep percolating on to this parent **S**.

Figure 4.136: If **S** covers only a null-element, append it with a **-TOBEVS**.

Figure 4.137: The β in this rule has no top-level node of type **VP.***. If **S** has no children of type **VP.*** then it is percolated with a **-TOBEAS**.

Chapter 5

Syntax Evaluations

This chapter describes the syntax evaluation to check the performance of the system. This syntax evaluation uses the Berkley parser (Petrov and Klein, 2007). The same method of evaluation is done on our reannotated GCG grammar and the CCG grammar from (Clark and Curran, 2007). Each corpus is used to define its own probabilistic grammar which is automatically annotated with additional latent variable values (Petrov and Klein, 2007) to introduce distinctions based on distributions of words and syntactic categories that increase the probability of the corpus (and improve the accuracy of parsing on held-out data), but do not affect the calculation of dependency structure. Each latent-variable-augmented probabilistic grammar is used to parse sentences of WSJ section 23. Each parsed result is fed through the standard evalb[1] to report the precision, recall, and F-measure. Because of using the standard evalb, the format of our reported results look similar to that of other parsing system. We also run a pair-wise McNemar's and Student's t-test on the two results to confirm that GCG result is significantly more accurate than CCG on both tests, hence GCG seems to be more learnable than CCG.

In addition to this strict syntax evaluation, we also tried a couple more relaxed evaluations. In the first one, called "onlyval", we only care about the compositional structure of each syntax category without looking at its composed primitive categories. The next one is called "unlabeled" where the evaluation is relaxed even further to only care about the structure of the parsed trees without looking at the syntax category at any node. Both of these relaxed evaluations once again show that GCG is significantly more accurate than CCG on the parsing task.

The syntax evaluation involves two steps. The first step is the *parsing* and the second step is

[1] http://nlp.cs.nyu.edu/evalb/

evaluating the parse result. This two step separation helps make it easier to describe later evaluations on the recovery of dependency relations (Chapter 6) and long-distance or unbounded dependency relations (Chapter 7) because these evaluations always need syntactic parsing as the first step.

5.1 Syntax Evaluation on GCG

For GCG, the *parsing* step is illustrated in Figure 5.1 and the *evaluating* step is shown in Figure 5.2. This section will describe these two steps in more detail and conclude with the result of this evaluation in Table 5.1.

5.1.1 Syntax Parsing in GCG

There are a total of 7 steps to syntax parsing using the GCG grammar. These steps are annotated by the yellow circles in Figure 5.1. Training data used section 02 to 21 from the Wall Street Journal (WSJ) of the Penn Treebank, hence these filenames started with "wsj02to21". Testing data is section 23, so these filenames started with "wsj23". The flows of training and testing can be seen from Figure 5.1 as follow:

- Gold data provided for training as PTB's "wsj02to21" is passed through a series of *step 1*, *step 3*, *step 4*, *step 5*, *step 6* in that exact ordering. The *step 6* is the Berkley Parser that is waiting for the raw test data coming as another input, described next.

- Hypothesis test data flow starts from PTB's "wsj23" and goes through *step 2* and *step 6* which is the Berkley Parser. The ".parsed.output" then goes to *step 7* to produce ".parsed.linetrees" which is the hypothesis of syntax parsing that will be used as input to subsequent evaluations for GCG grammar.

- Gold test data flow starts from PTB's "wsj23" and goes through *step 1* then *step 3* as the gold standard for this syntax parsing. This will be used as input to subsequent evaluations for GCG grammar.

step 1: **Convert PTB's trees to .linetrees**

PTB's trees are multi-line of indented structure to be friendly reading by human. However, without even an empty line delimiter between the sentences, this format is not very easy for a program to parse as it has to keep track of pushing/popping the open and close parentheses to detect their balance at the end of a sentence. For example, the two sentences appear like this in wsj_0202.mrg:

```
( (S
   (NP-SBJ (DT The) (JJ new) (NN rate) )
   (VP (MD will)
     (VP (VB be)
       (ADJP-PRD (JJ payable)
         (NP-TMP (NNP Feb.) (CD 15) ))))
   (. .) ))
( (S
   (NP-SBJ-1 (DT A) (NN record) (NN date) )
   (VP (VBZ has) (RB n't)
     (VP (VBN been)
       (VP (VBN set)
         (NP (-NONE- *-1) ))))
   (. .) ))
```

This *step 1* is the Perl script editabletrees2linetrees.pl. This script reformats the data to place a tree on each line and get rid of unnecessary outermost pair of parentheses so the two trees above will look like the following in the .linetrees file.[2]

```
(S (NP-SBJ (DT The) (JJ new) (NN rate) ) (VP (MD will) (VP (VB
   be) (ADJP-PRD (JJ payable) (NP-TMP (NNP Feb.) (CD 15) ))))
   (. .) )
(S (NP-SBJ-1 (DT A) (NN record) (NN date) ) (VP (VBZ has) (RB n
   't) (VP (VBN been) (VP (VBN set) (NP (-NONE- *-1) )))) (. .)
    )
```

[2] Note that the text may look wrapped on multiple lines but that is only because of the width of this document. There is no newline character anywhere but the end of each sentence.

step 2: **Extract only words of sentences in .linetrees to create .sents**

This command simply removes all the annotation of each sentence, so the two example sentences in Section 5.1.1 above would look like this in the .sents file:

```
The new rate will be payable Feb. 15 .
A record date has n't been set .
```

step 3: **Reannotate PTB grammar into GCG grammar**

This *step 3* undertakes the biggest task in this evaluation process which is to reannotate the PTB corpus into our GCG grammar. It is a chain of 4 sub-commands described below.

- The first one is annotateFixes.pl containing 167 rules designed to fix various issues in PTB. Examples of these fixes ranging from simple typo introduced by annotators, e.g. "diversifed" instead of "diversified", inconsistencies in using null elements for trace, e.g. **(-NONE- *-i)** vs **(-NONE- *T*-i)**, or mistakes in the analysis of past tense **VBD** vs past participle **VPN**, etc. This is by no means a complete set of fixes. There are more problems in PTB that are very difficult to fix. For example, particle or phrasal verbs like "look after", "dig up", "stand by", etc. are not easily identifiable in PTB. The particle following the verb, while forming a semantic unit with the verb and should be annotated with a **PRT**, is often annotated as a preposition **IN** or **TO**. Together with the mostly flat structure of the trees, it is impossible to know when a preposition should be grouped with the verb to form a phrasal verb or should be grouped with the following (usually nominal) phrase.

- Next in the chain is annotate-gcg13.pl containing about 175 rules to reannotate PTB trees into our GCG format as described in detail in Chapter 4. The number 13 in the name stands for version of year 2013 that contains a number of improvements over prior versions including annotate-gcg12.pl which was used in (Nguyen et al., 2012). One of these significant improvements is a change to make all primitive categories a single-character and the capability to compose nested categories.

- Third in the chain is killUnaries.pl. This script turns unary branches of the form A $\rightarrow$ B $\rightarrow$ C into just A $\rightarrow$ C. This is mostly just to evaluate the parser on a fair format, with always the same number of constituents in each tree for the same sentence.

- Last in the chain are 3 in-line Perl scripts. The first one remove all the sentences that failed to go through the reannotation process completely. The signal of this failure is the tree still having some category consisting of only capital letters because trees coming out of the annotate-gcg13.pl must have at least one dash "-" at every category, e.g. a "-f" to record the "from" category which is the category the node was in before being transformed to this category. This "from" field on every category is just a debug information and will be removed by the next in-line Perl script. Last in the chain is another in-line Perl script to move the "-lX" field to the end of the category with X being "I": the Identity or head of the constituent, "A": the argument, "M": the modifier, "C": the coordination conjunction, or "N": no relation. Moving this field to the end helps avoid categories with the same features differing in order (e.g. **-bN-lA** vs **-lA-bN**) sparsifying our data unnecessarily.

step 4: **Modify trees to conform to the format expected by the Berkley GrammarTrainer**

This step is a chain of 2 sub-commands to do minor reformatting of the trees before feeding them through the Berkley Grammar Trainer. The first sub-command is an in-line Perl script to turn every dash "-" in the category to the ampersand "&" because the Berkley Grammar Trainer uses the "-" as a delimiter in its split-merge algorithm. The last sub-command in the chain is a sed script to put back the extra set of open and close parentheses to wrap around each tree, hence the name of the output file got "extrpar" in it. This sub-command is needed because that is the format of the original PTB and also the one the Berkley Grammar Trainer accepts.

step 5: **Train the Berkley Grammar Trainer on GCG**

This step is Java code that can be downloaded directly from the Berkley NLP Group.[3] We use the last 1671 sentences of wsj02to21 for validation. The larger the split-merge cycles used, the slower but higher accuracy is accomplished. This command is very time and memory intensive, only second to the parsing task which is the next command down the command chain.

step 6: **Parse using the Berkley Parser**

This step is also Java code: the Berkley Parser that can be downloaded as a bundle with the Berkley Grammar Trainer above. This is the most time and memory consuming task. Three

[3] http://nlp.cs.berkeley.edu/Software.shtml

split-merge cycles would require at least 4GB of memory allocated to the JVM to run while five split-merge cycles require at least 8GB of memory.

step 7: **Reformat the output from the Berkley Parser**

This step consists of a chain of 3 sub-commands to do various reformatting to turn the output from the Berkley Parser into our GCG trees. The first sub-step in the chain does the reversed task of the first sub-step of *step 4* above, replacing the "&" in the category back to their original "-". Next in the chain is a sed step to remove the extra set of open and close parentheses around each tree as they are not needed in our GCG format.

5.1.2 Evaluating GCG Parse Result

This step has only one sub-step, namely *step 1* in the yellow circle in Figure 5.2. The two inputs ".parsed.linetrees" (hypothesis) and ".gcg13.linetrees" (gold) are the outputs of the parsing step above. The output from evalb is ".gcg13.syneval" which is a report of precision, recall, and F-measure of the system on GCG grammar. This result is shown in Table 5.1.

5.2 Syntax Evaluation on CCG

For CCG, the *parsing* step is illustrated in Figure 5.3 and the *evaluation* step is shown in Figure 5.4. This section will describe these two steps in more detail and conclude with the result of this evaluation in Table 5.2.

5.2.1 Syntax Parsing on CCG

There are a total of 6 steps to do syntax parsing using the CCG grammar, one command less than the parsing using GCG because we start from CCG Bank so there is no need for a step to reannotate PTB. These commands are annotated by the yellow circles in Figure 5.3 in much the same way as the parsing for GCG in Section 5.1. The main difference between this CCG evaluation compared to the GCG evaluation is that this one starts from the CCG corpus (the CCG version of the PTB), not the original PTB, and it does not have the step to convert the corpus into GCG format. Training data used the same sections 02 to 21 from the Wall Street Journal (WSJ) of the CCG Bank, hence the filename started with "wsj02to21". Testing data is

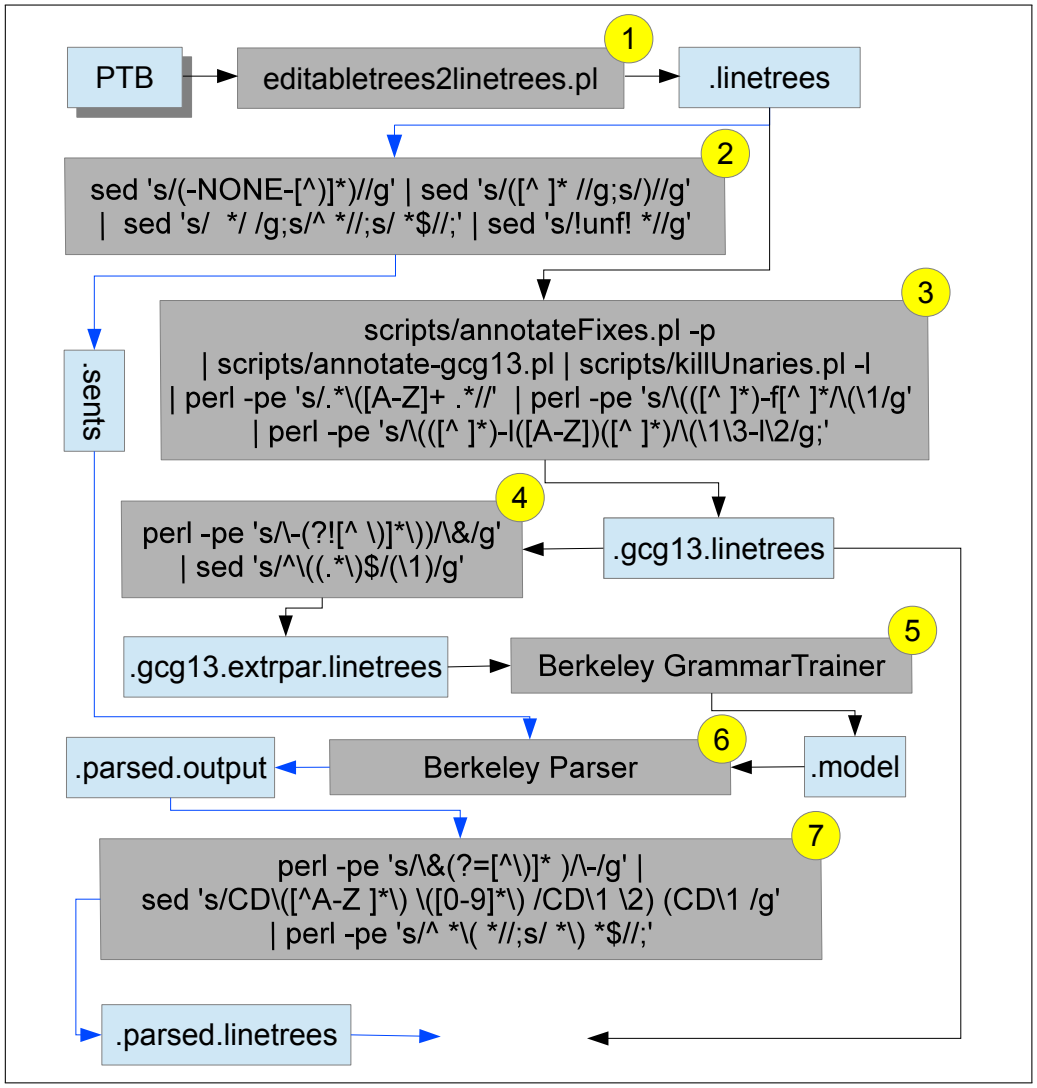


Figure 5.1: Syntax parsing for GCG. The darker color rectangles denote the commands with arrows coming in as inputs and going out as outputs. The number *i* in the yellow circle at the upper right corner of the command is referred to as *step i* in the writeup description of these commands for the figure. The lighter color rectangles denote the extensions of the files being generated or consumped by the commands. The special lighter color rectangles with a shade denote the files from corpora, i.e. "PTB" for PennTreebank corpus. There are two kinds of arrows: the black ones are for the flow of gold data and the blue ones are for the hypothesis data. If a command has at least one incoming blue arrow then its outgoing arrow must be a blue one. The two outputs going out of this Figure are ".parsed.linetrees" (hypothesis) and ".gcg13.linetrees" (gold). They will be used in a number of different syntax, dependency, filler-gap, and proposition evaluations for GCG.

=== Summary ===		
– All –		
Number of sentence	=	1795
Number of Error sentence	=	0
Number of Skip sentence	=	0
Number of Valid sentence	=	1795
Bracketing Recall	=	89.74
Bracketing Precision	=	89.93
Bracketing FMeasure	=	89.84
Complete match	=	40.56
Average crossing	=	2.00
No crossing	=	47.30
2 or less crossing	=	69.69
Tagging accuracy	=	94.49
– len<=40 –		
Number of sentence	=	1705
Number of Error sentence	=	0
Number of Skip sentence	=	0
Number of Valid sentence	=	1705
Bracketing Recall	=	90.17
Bracketing Precision	=	90.35
Bracketing FMeasure	=	90.26
Complete match	=	42.46
Average crossing	=	1.77
No crossing	=	49.50
2 or less crossing	=	71.96
Tagging accuracy	=	94.54

Table 5.1: Syntax evaluation for GCG on Berkley Parser.

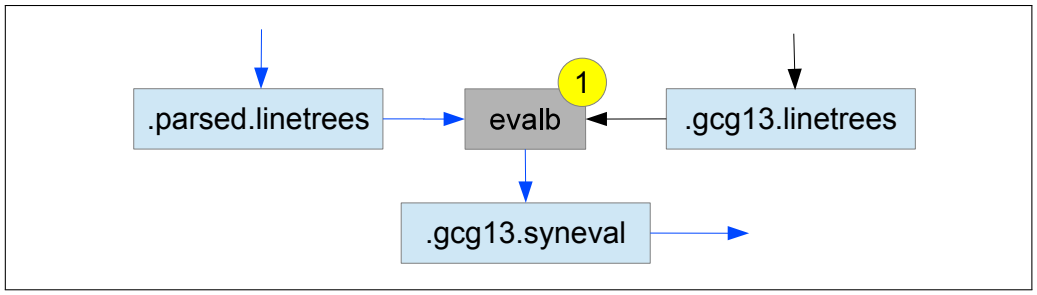


Figure 5.2: Syntax evaluation for GCG. The hypothesis file ".parsed.linetrees" and gold file ".gcg13.linetrees" inputs to the standard "evalb" script are coming from the outputs toward the end of Figure 5.1. The output ".gcg13.syneval" of this command is the official result of the GCG evaluation on syntax parsing. This result is shown in Table 5.1.

section 23, so the filename started with "wsj23". The flows of training and testing can be seen from Figure 5.3 as followed:

- Gold data provided for training as CCG's "wsj02to21" → *step 1* → *step 3* → *step 4* → *step 5* which is the Berkley Parser that is waiting for the raw test data coming as another input, described next.
- Hypothesis test data flow starts from CCG's "wsj23" → *step 2* → *step 5* which is the Berkley Parser. The "ccg.parsed.output" → *step 6* to produce ".ccg.parsed.linetrees" which is the hypothesis of syntax parsing that will be used as input to subsequent evaluations for CCG grammar.
- Gold test data flow starts from CCG's "wsj23" → *step 1* to produce the gold standard ".ccg.linetrees" for this syntax parsing. This will be used as input to subsequent evaluations for CCG grammar.

step 1: **Convert CCG's trees to .linetrees**

This command is the Perl script ccglinetrees2nicelinetrees.pl. CCG Bank already has one tree each line, the format we prefer over the original PTB. However, each tree in CCG Bank comes prefixed by a line to contain the tracing information back to its original PTB. The same example of the two trees shown in Section 5.1.1 but on CCG Bank is listed below:

```
ID=wsj_0202.2 PARSER=GOLD NUMPARSE=1
```

```
(<T S[dcl] 0 2> (<T S[dcl] 1 2> (<T NP 1 2> (<L NP[nb]/N DT DT
   The NP[nb]_129/N_129>) (<T N 1 2> (<L N/N JJ JJ new N_124/
   N_124>) (<L N NN NN rate N>) ) ) (<T S[dcl]\NP 0 2> (<L (S[
   dcl]\NP)/(S[b]\NP) MD MD will (S[dcl]\NP_88)/(S[b]_89\NP_88:
   B)_89>) (<T S[b]\NP 0 2> (<L (S[b]\NP)/(S[adj]\NP) VB VB be
   (S[b]\NP_98)/(S[adj]_99\NP_98:B)_99>) (<T S[adj]\NP 0 2> (<L
    (S[adj]\NP)/NP JJ JJ payable (S[adj]\NP_106)/NP_107>) (<T
   NP 0 1> (<T N 0 2> (<L N/N[num] NNP NNP Feb. N/N[num]_112>)
   (<L N[num] CD CD 15 N[num]>) ) ) ) ) ) ) (<L . . . . .>) )
ID=wsj_0202.3 PARSER=GOLD NUMPARSE=1
(<T S[dcl] 0 2> (<T S[dcl] 1 2> (<T NP 1 2> (<L NP[nb]/N DT DT
   A NP[nb]_97/N_97>) (<T N 1 2> (<L N/N NN NN record N_92/N_92
   >) (<L N NN NN date N>) ) ) (<T S[dcl]\NP 0 2> (<T (S[dcl]\
   NP)/(S[pt]\NP) 0 2> (<L (S[dcl]\NP)/(S[pt]\NP) VBZ VBZ has (
   S[dcl]\NP_55)/(S[pt]_56\NP_55:B)_56>) (<L (S\NP)\(S\NP) RB
   RB n't (S_68\NP_63)_68\(S_68\NP_63)_68>) ) (<T S[pt]\NP 0 2>
    (<L (S[pt]\NP)/(S[pss]\NP) VBN VBN been (S[pt]\NP_77)/(S[
   pss]_78\NP_77:B)_78>) (<L S[pss]\NP VBN VBN set S[pss]\NP_83
   >) ) ) ) (<L . . . . .>) )
```

We will need to strip off the prefix line of each tree. We also need to remove the co-indexations and non-local dependencies information on the tree because the Berkley grammar trainer and parser won't be able to make use of those. We preserve the feature on each category but remove the square brackets around them so **S[dcl]** would look like **Sdcl**. The result of this *step 1* of the CCG Bank trees above will look like the ones below:

```
(Sdcl (Sdcl (NP (NPnb/N The) (N (N/N new) (N rate))) (Sdcl\NP
   ({Sdcl\NP}/{Sb\NP} will) (Sb\NP ({Sb\NP}/{Sadj\NP} be) (Sadj
   \NP ({Sadj\NP}/NP payable) (NP (N (N/Nnum Feb.) (Nnum 15))))
   ))) (. .))
```

```
(Sdcl (Sdcl (NP (NPnb/N A) (N (N/N record) (N date))) (Sdcl\NP
   ({Sdcl\NP}/{Spt\NP} ({Sdcl\NP}/{Spt\NP} has) ({S\NP}\{S\NP}
   n't)) (Spt\NP ({Spt\NP}/{Spss\NP} been) (Spss\NP set)))) (.
   .))
```

step 2: **Extract only words of sentences in .linetrees to create .sents**

This step is the same as *step 2* of Section 5.1.1, stripping off all the CCG annotations to leave only the text of the sentence to feed through the parser.

steps 3 to 6

These steps are exactly the *steps 4 to 7* described on the syntax parsing for GCG above. Similarly, the two outputs "ccg.parsed.linetrees" (hypothesis) and ".ccg.linetrees" (gold) coming out of this parsing step can be used as inputs into various evaluations later.

5.2.2 Evaluating CCG Parse Result

This step, as shown in Figure 5.4, used the standard evalb script, exactly like the evaluation of GCG parse result above, but only different in the inputs and output. The inputs this time are the gold and hypothesis data coming from the parsing step of Figure 5.3. The result of this CCG syntax evaluation is shown in Table 5.2.

5.3 Significance Tests on Syntax Evaluations for GCG vs CCG

From the syntax evaluation results of GCG in Table 5.1 and that of CCG in Table 5.2, it shows that GCG can parse more accurately than CCG on Section 23 of WSJ. To generalize the claim that GCG is more learnable than CCG on syntax parsing, we do significance tests on series of data points extracted from the parsing results of Section 23 using each grammar. Specifically, we run the two series of data points side-by-side through a Student's t-test or McNemar's test to reject the null hypothesis that the difference between the two series is random.

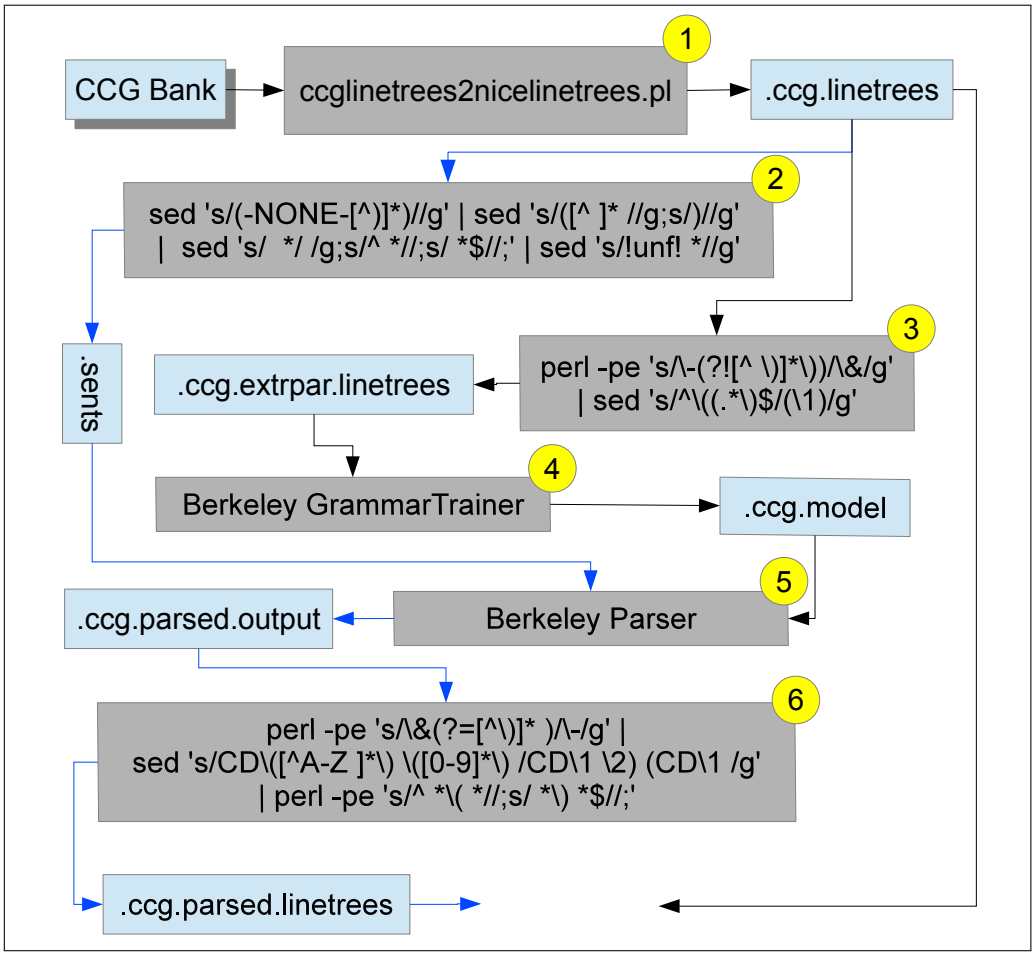


Figure 5.3: Syntax parsing for CCG. The darker color rectangles denote the commands with arrows coming in as inputs and going out as outputs. The number *i* in the yellow circle at the upper right corner of the command is referred to as *step i* in the writeup description of these steps. The lighter color rectangles denote the extensions of the files being generated or consumped by the commands. The special lighter color rectangles with a shading denote the files from corpora, i.e. "CCG Bank" for CCG Treebank corpus. There are two kinds of arrows: the black ones are for the flow of gold data and the blue ones are for the hypothesis data. If a command has at least one incoming blue arrow then its outgoing arrow must be a blue one. The two outputs going out of this Figure are ".ccg.parsed.linetrees" (hypothesis) and ".ccg.linetrees" (gold). They will be used in a number of different syntax, dependency, and filler-gap evaluations for CCG.

=== Summary ===		
– All –		
Number of sentence	=	1795
Number of Error sentence	=	0
Number of Skip sentence	=	1
Number of Valid sentence	=	1794
Bracketing Recall	=	86.96
Bracketing Precision	=	86.79
Bracketing FMeasure	=	86.87
Complete match	=	34.56
Average crossing	=	2.12
No crossing	=	43.65
2 or less crossing	=	68.62
Tagging accuracy	=	93.51
– len<=40 –		
Number of sentence	=	1705
Number of Error sentence	=	0
Number of Skip sentence	=	1
Number of Valid sentence	=	1704
Bracketing Recall	=	87.31
Bracketing Precision	=	87.17
Bracketing FMeasure	=	87.24
Complete match	=	36.03
Average crossing	=	1.90
No crossing	=	45.31
2 or less crossing	=	71.07
Tagging accuracy	=	93.52

Table 5.2: Syntax evaluation for CCG on Berkley Parser.

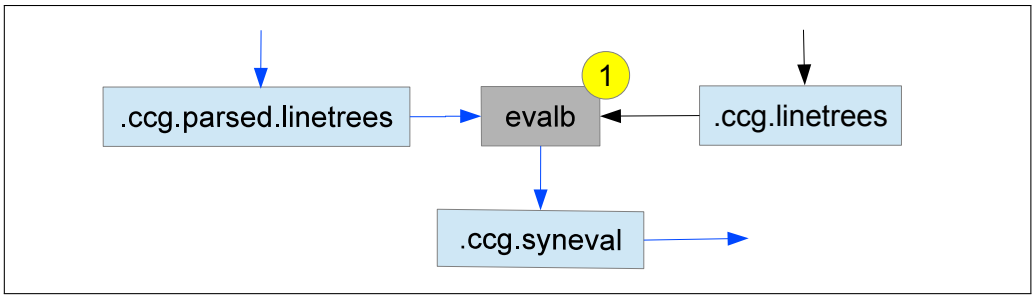


Figure 5.4: Syntax evaluation for CCG. The hypothesis ".ccg.parsed.linetrees" and gold ".ccg.linetrees" inputs to the standard "evalb" of *step 1* are coming from the outputs toward the end of Figure 5.3. The output ".ccg.syneval" of this command is the result of the CCG evaluation on syntax parsing. This result is shown in Table 5.2. The standard "evalb" used in this Figure is exactly the one used in Figure 5.2. This shows that the syntax evaluations for GCG and CCG are done in the exact same way, only different in the grammar formalism.

5.3.1 Student's t-test on Syntax Evaluations for GCG vs CCG

The two inputs ".gcg13.syneval" and ".ccg.syneval" coming to Figure 5.5 are the outputs of the syntax evaluations for GCG and CCG from Figure 5.2 and Figure 5.4, respectively. The *step 1* in Figure 5.5 composed of an "egrep" and an inline Perl script. This command depends on the format of the output generated by the commonly used "evalb" script. The "egrep" extracts only the lines with numbers separated by spaces which is the first line of each individual sentence from ".gcg13.syneval" and ".ccg.syneval" as apposed to the final summary results of all the sentences shown in Table 5.1 and Table 5.2. These lines show a summary of parse results on each sentence. The inline Perl script then takes the last number of each line which is the parsing accuracy of each sentence into the files ".gcg13.syneval.corr" and ".ccg.syneval.corr" (the ".corr" mean "correct" instances). These two ".corr" files are input into the "ttest.r" script to compute the Student's t-test result. The result of this significance test is in ".gcg13.ccg.ttestsignif" file and the content of this file is shown in Table 5.3. This test shows that the *p-value* is less than 5%, so it is concluded that GCG is statistically significantly more accurate compared to CCG.

5.3.2 McNemar's test on Syntax Evaluations for GCG vs CCG

To confirm one more time that GCG is significantly more accurate on parsing than CCG, we do another pair-wise test of the parsing results using McNemar's test as shown in Figure 5.6. This time, instead of starting from the ".gcg13.syneval" and ".ccg.syneval" to extract the correct

```
==> 131028/wsj23-inboth..gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.gcg13_syneval..ccg.
    wsj02to21-ccg-1671-5sm.fullberk.parsed.ccg_syneval..ttestsignif <==
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.gcg13_syneval.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.ccg_syneval.corr"
[1] 88.73589
[1] 85.98558

      Paired t-test

data: d1[["dat"]] and d2[["dat"]]
t = 6.7563, df = 1794, p-value = 1.906e-11
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.951933 3.548703
sample estimates:
mean of the differences
           2.750318
```

Table 5.3: Student's t-test for Syntax Evaluation between GCG and CCG on the Berkley parser using the R stats software.

percentage of each individual sentence to be used as data points, we start right after the *parsing* step to bypass the "evalb" as we are looking for the exact match sentences as data points instead of the percentage of constituents matched.

There are two steps in Figure 5.6 but *step 1* was duplicated in two places to make it clear by reducing the number of inputs/outputs coming to and going out from it. There are four inputs, or two pair of them. The upper pair of inputs are ".parsed.linetrees" (GCG hypothesis) and ".gcg13.linetrees" (GCG gold) coming from the outputs of Figure 5.1. The lower pair of inputs are ".ccg.parsed.linetrees" (CCG hypothesis) and ".ccg.linetrees" (CCG gold) coming from the outputs of Figure 5.3. The *step 1* composed of a shell command "sdiff" to compare the hypothesis and the gold data line by line. The output of this "sdiff" is piped through a "grep" to eliminate all the failed parse sentences. Last in the chain is an inline Perl script to replace each unmatched pair of sentences by a 0 and each matched pair by a 1. The result of this *step 1* is a series of 0 and 1, with 0 representing sentences that were not perfectly parsed (may be partially correct), and 1 representing sentences that were parsed 100% correct. These two series of 0 and 1 in ".gcg13.synmatch.corr" and ".ccg.synmatch.corr" are used as data points input to *step 2* which is an R script to run the McNemar test. The content of the result file ".gcg13.ccg.symmatch.signif" is shown in Table 5.4, and once again confirm that GCG can parse statistically significantly more accurate than CCG because the *p-value* of this McNemar's test is less than 5%.

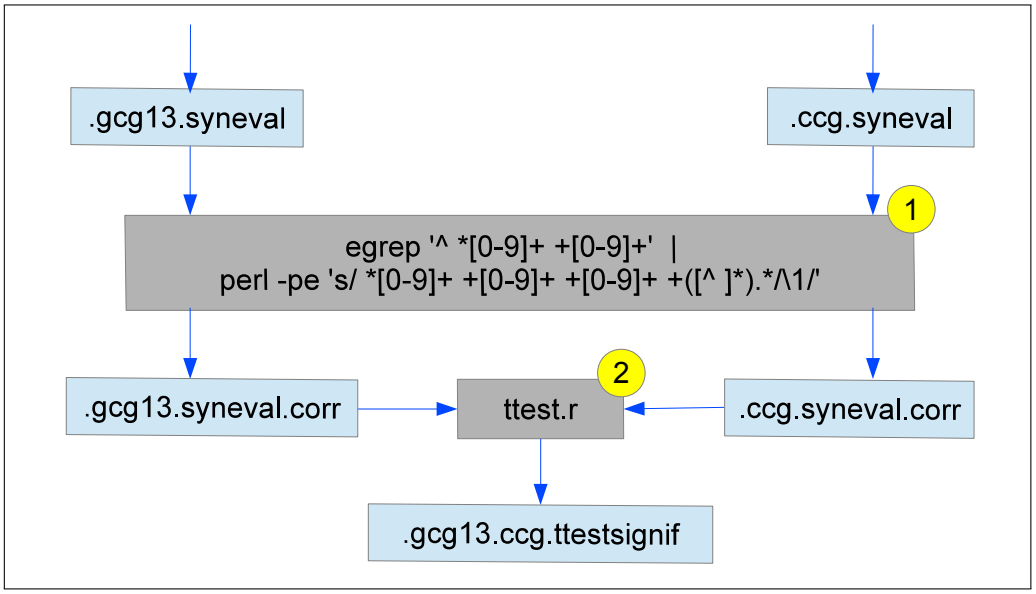


Figure 5.5: Student's t-test to measure significance for syntax evaluation between GCG and CCG. The two inputs ".gcg13.syneval" and ".ccg.syneval" are coming from the last outputs of Figure 5.2 and Figure 5.4, respectively. The final output produced is ".gcg13.ccg.ttestsignif" as shown in Table 5.3. This result shows the syntax evaluation on GCG is significantly more accurate than that of CCG.

```
==> 131028/wsj23-inboth..gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.synmatch..ccg.
    wsj02to21-ccg-1671-5sm.fullberk.parsed.synmatch..signif <==
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.synmatch.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.synmatch.corr"
[1] 0.3799443
[1] 0.3420613

      McNemar's Chi-squared test with continuity correction

data: d1[["dat"]] and d2[["dat"]]
McNemar's chi-squared = 9.3912, df = 1, p-value = 0.00218
```

Table 5.4: McNemar test for Syntax Evaluation between GCG and CCG on the Berkley parser

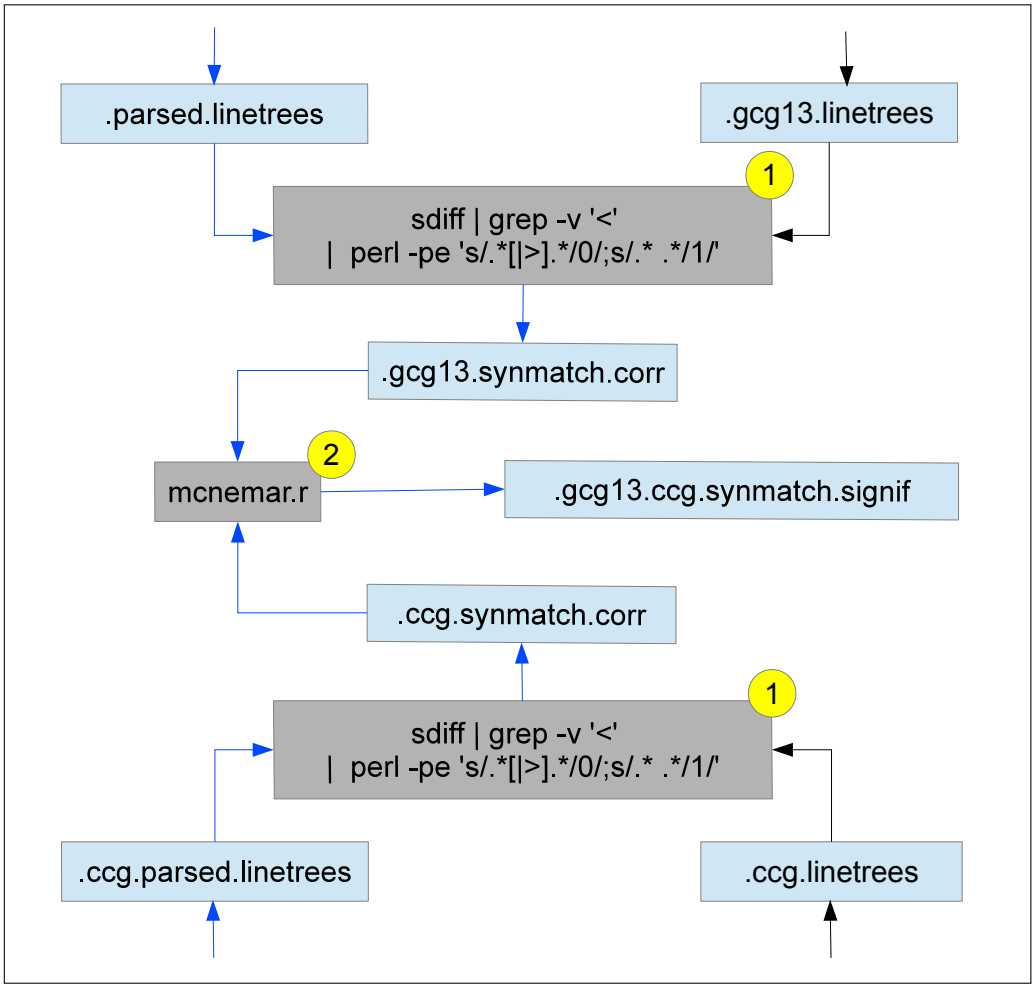


Figure 5.6: McNemar's significance test for syntax evaluation between GCG and CCG. The upper pair of inputs are ".parsed.linetrees" and ".gcg13.linetrees", coming from the outputs toward the end of Figure 5.1 to represent the syntax parsing on GCG. The lower pair of inputs are ".ccg.parsed.linetrees" and ".ccg.linetrees", coming from the outputs toward the end of Figure 5.3 to represent the syntax parsing on CCG. The *step 1* is duplicated in 2 places to make it clear by reducing the number of its inputs and outputs. This *step 1* extracts only the perfect matches in parsing on either GCG or CCG. The ".corr" file name implies taking only the perfectly "correct" parsed sentences. The *step 2* is an R script to compute the McNemar significance test. This result as shown in Table 5.4, one more time, confirms that GCG is significantly more accurate than CCG on syntax parsing.

5.4 Relaxed Syntax Evaluations

We conduct two relaxed syntax evaluations for both GCG and CCG. The first relaxation is to ignore the type of individual primitive category and the **-l** tag encoding the local predicate-argument dependencies. For this relaxation, the syntax category at each tree node is measured only by the correct composition of the number of individual generic primitive categories (does not care about the exact correctness of each compositional primitive category). This relaxation also drops the predicate-argument structure and is referred to in the code as the "onlyval" evaluation. The second relaxation went further to check only the correctness of the tree structure without caring what category the tree has at each node. This relaxation is referred to as "unlabeled" syntax evaluation.

5.4.1 "Onlyval" Syntax Evaluations

For these relaxed evaluations, the parsed results of GCG and CCG are routed through an additional step to reset every primitive category to just an **X** and remove the **-l** tag encoding the local predicate-argument. For example, the category for a transitive main verb of a sentence may be **V-aN-bN-lI** and will be simplified to just **X+X+X** where the **V**, the **-aN**, and the **-bN** are all relaxed to just **X**'s and the **-lI** is removed. This relaxation maintains the compositional structure of the categories, like in this example, the category **X+X+X** denotes that it composed of 3 primitive categories but does not care what exactly each one of those 3 are. This relaxation also maintains the complex compositional structure, e.g. a category of **L-aN-bA-aN-lM** will be relaxed to **X+X+{X+X}**.

This relaxation is done in *step 1* of Figure 5.7. The upper pair of inputs is coming from the outputs in Figure 5.1 and the lower pair is coming from the outputs of Figure 5.3. The corresponding pairs of outputs are routed through the the evalb script in the same way the complete syntax evaluations were done as shown in Figure 5.2 and Figure 5.4. The results of these two evaluations is shown in Table 5.5 and the accompanied pair-wise Student's t-test and McNemar's test are in Table 5.6. These evaluations show that GCG is significantly more accurate than CCG again on this relaxed setting of the syntax.

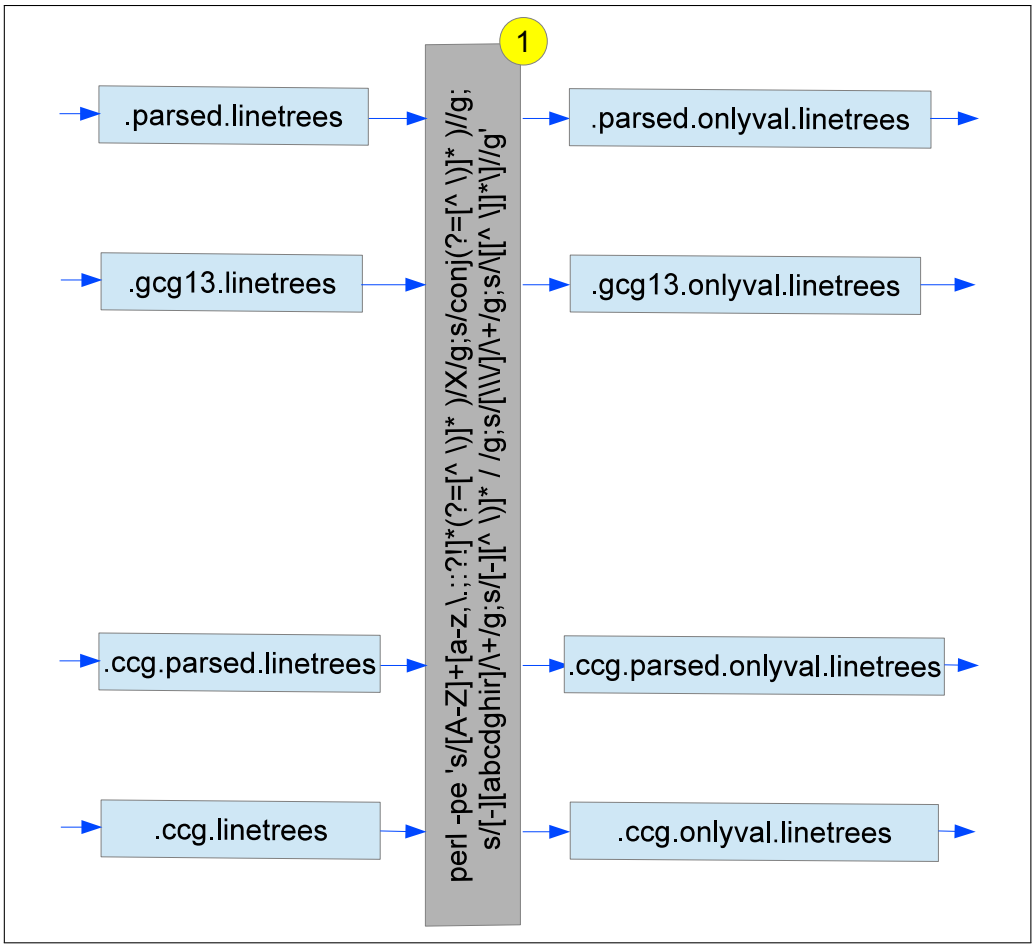


Figure 5.7: "onlyval" relaxation drops the check for correctness of each individual primitive category as they were all **X**'ed out. This also drops the **-l** tag to not care about the local predicate-argument dependencies. This relaxation is meant to only check the correct compositional structure of the syntax category at each node as well as the correct structure of the tree overall.

5.4.2 "Unlabeled" Syntax Evaluations

These evaluations take the relaxation one step further where even the compositional structure of the syntax categories is ignored, hence every syntax category is simplified to just an **X**. A parsed tree under this relaxation is considered a match if it has the same tree structure with the gold tree without caring about any syntax category at every node.

For these relaxed evaluations, the parsed results of GCG and CCG are routed through an

GCG			CCG		
=== Summary ===			=== Summary ===		
– All –			– All –		
Number of sentence	=	1795	Number of sentence	=	1795
Number of Error sentence	=	0	Number of Error sentence	=	0
Number of Skip sentence	=	0	Number of Skip sentence	=	1
Number of Valid sentence	=	1795	Number of Valid sentence	=	1794
Bracketing Recall	=	88.12	Bracketing Recall	=	87.30
Bracketing Precision	=	88.12	Bracketing Precision	=	87.30
Bracketing FMeasure	=	88.12	Bracketing FMeasure	=	87.30
Complete match	=	44.18	Complete match	=	37.57
Average crossing	=	2.00	Average crossing	=	2.01
No crossing	=	47.30	No crossing	=	43.65
2 or less crossing	=	69.69	2 or less crossing	=	70.23
Tagging accuracy	=	97.32	Tagging accuracy	=	94.62
– len<=40 –			– len<=40 –		
Number of sentence	=	1705	Number of sentence	=	1705
Number of Error sentence	=	0	Number of Error sentence	=	0
Number of Skip sentence	=	0	Number of Skip sentence	=	1
Number of Valid sentence	=	1705	Number of Valid sentence	=	1704
Bracketing Recall	=	88.71	Bracketing Recall	=	87.73
Bracketing Precision	=	88.71	Bracketing Precision	=	87.73
Bracketing FMeasure	=	88.71	Bracketing FMeasure	=	87.73
Complete match	=	46.28	Complete match	=	39.14
Average crossing	=	1.77	Average crossing	=	1.80
No crossing	=	49.50	No crossing	=	45.31
2 or less crossing	=	71.96	2 or less crossing	=	72.65
Tagging accuracy	=	97.37	Tagging accuracy	=	94.65

Table 5.5: “onlyval” syntax evaluations result for GCG on the left and CCG on the right. These relaxing results also consistently show that GCG is a better parsing tool than CCG on WSJ section 23.

additional step to reset every syntax category to just an **X** before sending to the evalb script. This is illustrated in Figure 5.8. The upper pair of inputs is coming from the outputs in Figure 5.1 and the lower pair is coming from the outputs of Figure 5.3. The corresponding pairs of outputs are routed through the the evalb script in the same way the complete syntax evaluations were done as shown in Figure 5.2 and Figure 5.4. The results of these two evaluations is shown in

```
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.onlyval.
    gcg13_syneval.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.onlyval.ccg_syneval.corr
    "
[1] 89.4162
[1] 87.55787

      Paired t-test

data: d1[["dat"]] and d2[["dat"]]
t = 4.5862, df = 1794, p-value = 4.826e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.063620 2.653049
sample estimates:
mean of the differences
           1.858334
```

```
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.synmatch.onlyval.
    corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.synmatch.onlyval.corr"
[1] 0.416156
[1] 0.3582173

      McNemar's Chi-squared test with continuity correction

data: d1[["dat"]] and d2[["dat"]]
McNemar's chi-squared = 21.563, df = 1, p-value = 3.424e-06
```

Table 5.6: significance test results for "onlyval" evaluations: Student's t-test (above) and McNemar test (below). Both tests confirm that GCG is significantly more accurate than CCG on this relaxed syntax parsing task.

Table 5.7 and the accompanied pair-wise Student's t-test and McNemar's test are in Table 5.8. These evaluations show that GCG is significantly more accurate than CCG once again on this much relaxed setting of the syntax.

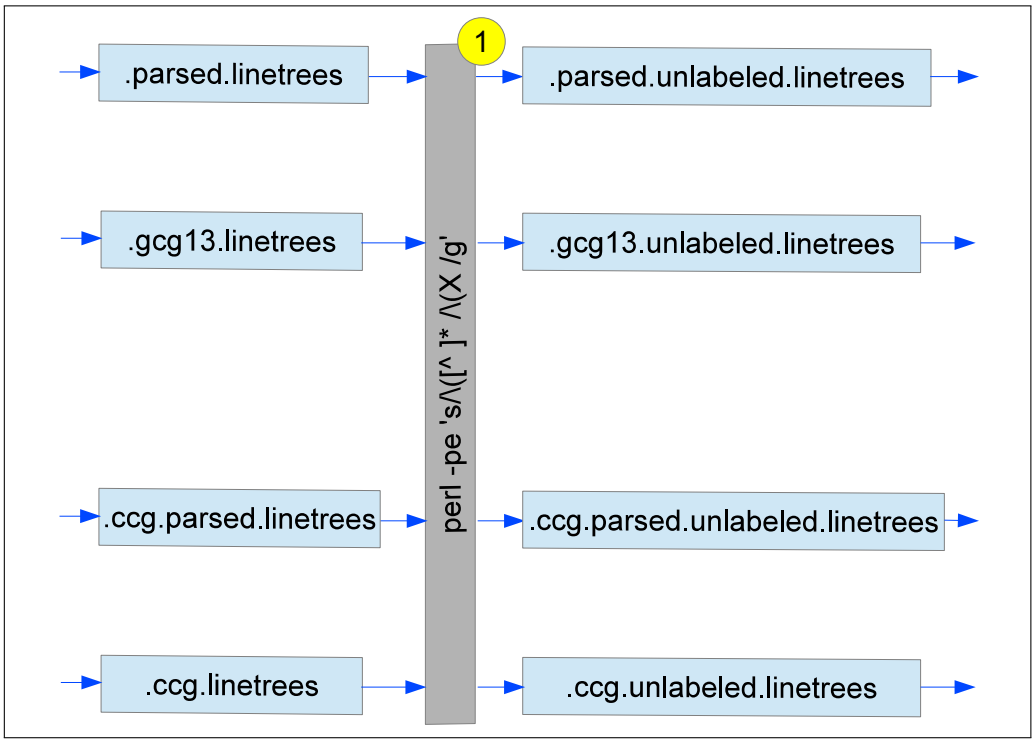


Figure 5.8: "unlabeled" relaxation turns each syntax category at every node into just an **X**. This relaxation therefore only evaluates the parsing on its capability to recover the tree structure.

GCG			CCG		
=== Summary ===			=== Summary ===		
– All –			– All –		
Number of sentence	=	1795	Number of sentence	=	1795
Number of Error sentence	=	0	Number of Error sentence	=	0
Number of Skip sentence	=	0	Number of Skip sentence	=	1
Number of Valid sentence	=	1795	Number of Valid sentence	=	1794
Bracketing Recall	=	90.60	Bracketing Recall	=	90.55
Bracketing Precision	=	90.60	Bracketing Precision	=	90.55
Bracketing FMeasure	=	90.60	Bracketing FMeasure	=	90.55
Complete match	=	47.30	Complete match	=	43.65
Average crossing	=	2.00	Average crossing	=	2.01
No crossing	=	47.30	No crossing	=	43.65
2 or less crossing	=	69.69	2 or less crossing	=	70.23
Tagging accuracy	=	100.00	Tagging accuracy	=	100.00
– len<=40 –			– len<=40 –		
Number of sentence	=	1705	Number of sentence	=	1705
Number of Error sentence	=	0	Number of Error sentence	=	0
Number of Skip sentence	=	0	Number of Skip sentence	=	1
Number of Valid sentence	=	1705	Number of Valid sentence	=	1704
Bracketing Recall	=	91.11	Bracketing Recall	=	90.96
Bracketing Precision	=	91.11	Bracketing Precision	=	90.96
Bracketing FMeasure	=	91.11	Bracketing FMeasure	=	90.96
Complete match	=	49.50	Complete match	=	45.31
Average crossing	=	1.77	Average crossing	=	1.80
No crossing	=	49.50	No crossing	=	45.31
2 or less crossing	=	71.96	2 or less crossing	=	72.65
Tagging accuracy	=	100.00	Tagging accuracy	=	100.00

Table 5.7: “unlabeled” syntax evaluations result for GCG on the left and CCG on the right. These relaxing results also consistently show that GCG is a better parsing tool than CCG on WSJ section 23.

```
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.unlabeled.
    gcg13_syneval.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.unlabeled.ccg_syneval.
    corr"
[1] 91.96688
[1] 90.6714

      Paired t-test

data: d1[["dat"]] and d2[["dat"]]
t = 3.8736, df = 1794, p-value = 0.0001111
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.6395591 1.9514047
sample estimates:
mean of the differences
           1.295482
```

```
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.synmatch.unlabeled.
    corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.synmatch.unlabeled.corr"
[1] 0.4729805
[1] 0.4362117

      McNemar's Chi-squared test with continuity correction

data: d1[["dat"]] and d2[["dat"]]
McNemar's chi-squared = 8.4163, df = 1, p-value = 0.003719
```

Table 5.8: significance test results for "unlabeled" evaluations: Student's t-test (above) and McNemar test (below). Both tests confirm that GCG is significantly more accurate than CCG on this relaxed syntax parsing task in general.

Chapter 6

Dependency Evaluations

Encouraged by the good result on the Syntax Evaluation described in previous chapter, we move on to the next evaluation which is to check the capability of this grammar formalism on the recovery of syntax dependencies. This chapter will describe this evaluation process for both grammars, our GCG and CCG, again on the same settings of scripts and Berkley grammar trainer and parser being used. For example, with a sentence *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990 ,* the target of this task is to evaluate the capability to generate predicate-argument dependencies as shown in the Figure 6.1.

The extraction of these local syntax dependencies from the parsed trees is done as an additional step after the parsing. Therefore, the flow of this evaluation for both GCG and CCG will look similar to their counterparts in the Syntax Evaluations at the first steps up to and including the generation of the parsed trees. These parsed trees will then go through a script to extract local predicate-argument dependencies before passing to a dependency evaluation script that counts the number of dependencies being recovered (recall) as well as the percentage of correct predictions (precision) and computes the F-Measure accordingly. The dependency evaluation result of each grammar will then go through the same Student's t-test and McNemar's test, as done in the Syntax Evaluations, to confirm or deny that the two results are significantly different (better or worse).

```
r0/0/A-aN-x#Rolls-Royce r0/1/i3 m1/0/A-aN-x#Motor m1/1/i3 c2/0/
   A-aN-x#Cars c2/1/i3 i3/0/N-aD#Inc. s4/0/V-aN-bC#said s4/1/i3
    s4/2/e6 i5/0/N#it e6/0/V-aN-bI#expects e6/1/i5 e6/3/s9 e6
   /2/r11 i7/0/D#its i7/1/s9 u8/0/A-aN-x#U.S. u8/1/s9 s9/0/N-aD
   #sales t10/0/I-aN-b{B-aN}#to t10/1/r11 r11/0/B-aN-b{A-aN}#
   remain r11/1/s9 r11/2/s12 s12/0/A-aN#steady a13/0/R-aN-bN#at
    a13/1/r11 a13/2/c16 a14/0/R-aN-x#about a14/1/115 115/0/A-aN
   -x#1,200 115/1/c16 c16/0/N-aD#cars i17/0/R-aN-bN#in i17/1/
   r11 i17/2/118 118/0/N#1990 .19/0/.#. .19/1/s4
```

Figure 6.1: The format of these dependencies presented as "word1/number/word2" to mean "word2" is a numeric argument "number" of predicate "word1." For example, in the phrase "... it expects ... sales remain ...", the "expects" predicate has 3 arguments: "it" is its argument 1, "remain" is its argument 2, and "sales" is its argument 3. Argument 0 is used as the identify relation of each word and usually tagged along with the syntax category of the word, a pound sign (#) as a delimiter, and the word itself. For brevity in the code, we use the lowercase initial of the word instead of the word itself over and over again at all the non-zero relations.

```
  (S (S-lI (N-lA (A-aN-x-lM (A-aN-x-lI Rolls-Royce))
                 (N-aD-lI (A-aN-x-lM (A-aN-x-lI Motor))
                          (N-aD-lI (A-aN-x-lM (A-aN-x-lI Cars))
                                   (N-aD-lI Inc.))))
           (V-aN-lI (V-aN-bC-lI said)
                    (C-lA (N-lA (N-lI it))
                          (V-aN-lI (V-aN-bI-lI expects)
                                   (I-lA (N-lA (D-lM (D-lI its))
                                               (N-aD-lI (A-aN-x-lM (A-aN-x-lI U.S.))
                                                        (N-aD-lI sales)))
                                         (I-aN-lI (I-aN-b{B-aN}-lM (I-aN-b{B-aN}-lI to))
                                                  (B-aN-lI (B-aN-lI (B-aN-lI (B-aN-b{A-aN}-lI remain)
                                                                             (A-aN-lA (A-aN-lI steady)))
                                                                    (R-aN-lM (R-aN-bN-lI at)
                                                                             (N-lA (A-aN-x-lM (R-aN-x-lM (R-aN-x-lI about))
                                                                                              (A-aN-x-lI 1,200))
                                                                                   (N-aD-lI cars))))
                                                           (R-aN-lM (R-aN-bN-lI in)
                                                                    (N-lA (N-lI 1990)))))))))))
    (.-lM (.-lI .)))
```

Figure 6.2: A complete example of GCG tree for the sentence: *Rolls-Royce Motor Cars Inc. said it expects its U.S. sales to remain steady at about 1,200 cars in 1990 .*

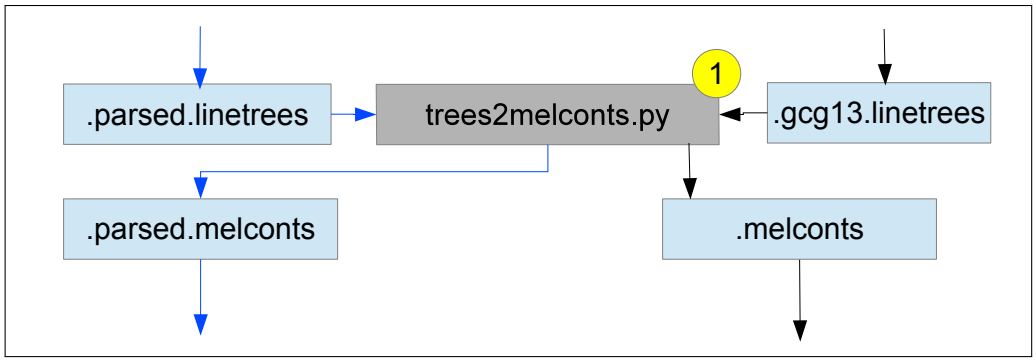


Figure 6.3: Compute Dependency Relations for GCG.

6.1 Dependency Evaluation on GCG

Figure 6.3 shows a work flow for GCG Dependency Evaluation. The two inputs ".parsed.linetrees" (hypothesis) and ".gcg13.linetrees" (gold) are coming from the parsing step, i.e. the outputs toward the end of Figure 5.1. The *step 1* in this Figure 6.3 is a Python script "trees2melconts.py" that extracts syntax dependencies as Melcuk-like constructs, hence the file name ending with a ".melconts", from GCG formatted ".linetrees" coming out from the parsing step. For example, if input is a ".linetrees" sentence in Figure 6.2 then the output is what shown in Figure 6.1.

The "melconts" part of the name implies that we model our syntax dependency constructs similar to the model of Deep Syntax introduced by (Mel'čuk, 1988). This "trees2melconts.py" script walks down the tree looking at the **-l** tag part of the syntax category of each node to deterministically print out the predicate-argument dependency between the nodes. Our GCG used **-lI** for Head, **-lA** for Argument, **-lM** for Modifier, and **-lC** for Coordination Conjunction. The arguments of a predicate are numbered labels, started from 0 which is the predicate itself to mean an identity relation. These numbers increase from 1 to 4 to denote argument 1 through argument 4. The lower the number, the more important or directly related the argument is to the predicate, e.g. if the predicate is the main verb of the sentence then the head word of the subject would be the argument 1 and the head word of the object would be argument 2. Experimental results show that we only need a few argument 5's to reannotate the entire PTB, so argument number 5 could safely be ignored.

The ".parsed.melconts" (hypothesis) and ".melconts" (gold) coming out from Figure 6.3 are used as inputs into the *step 1* of Figure 6.4 which is a Python script "depeval.py". This

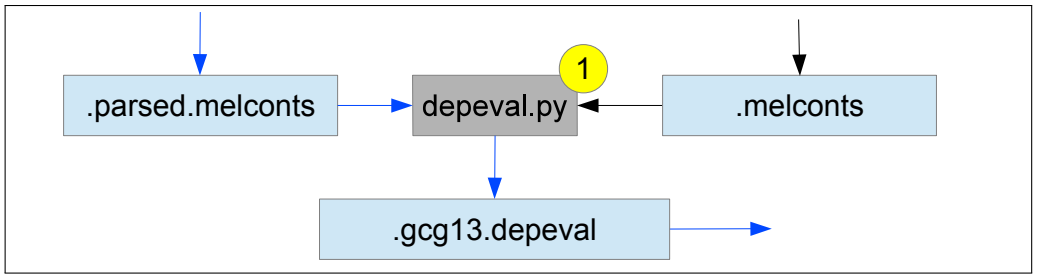


Figure 6.4: Dependency evaluation for GCG.

```
tail -2 131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.gcg13_depeval
TOT recall: 34861.0/38440.0 precis: 34861.0/38435.0
PCT recall: 0.906893860562 precis: 0.907011838168 fscore: 0.906952845528
```

Table 6.1: Dependency evaluation for GCG.

script simply counts the total number of melconts dependencies recovered (recall), the number of correct ones within those recovered (precision), and computes the F-Measure based on those recall and precision. The result of this evaluation is shown in Table 6.1.

6.2 Dependency Evaluation on CCG

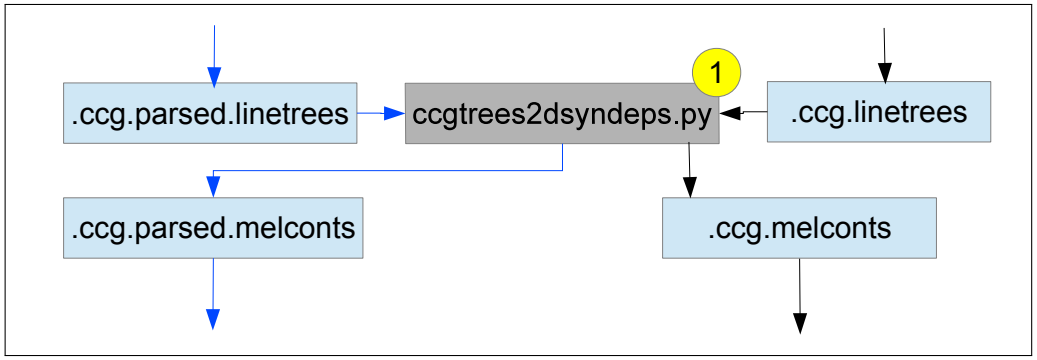


Figure 6.5: Compute Dependency Relation for CCG.

In the exact same manner with the Dependency Evaluation on GCG, this CCG Dependency Evaluation starts from the output files ".ccg.parsed.linetrees" (hypothesis) and ".ccg.linetrees" (gold) from the CCG Syntax Parsing step shown in Figure 5.3 as the inputs for Figure 6.5 that has the Python script "ccgtrees2dsyndeps.py" to extract the same type of ".melconts" from

```
tail -2 131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.ccg_depeval
TOT recall: 39811.0/45256.0 precis: 39811.0/45187.0
PCT recall: 0.879684461729 precis: 0.881027729214 fscore: 0.880355583074
```

Table 6.2: Dependency evaluation for CCG.

the functor-argument relations. The ".ccg.parsed.melconts" (hypothesis) and ".ccg.melconts" (gold) outputs from Figure 6.5 are then used as inputs into Figure 6.6 that has the same "depeval.py" script used in Figure 6.4 for the GCG Dependency Evaluation. The content of the output file ".ccg.depeval" is shown in Table 6.2. This result shows that the CCG Dependency recovery can get to about 88.03%, a 2% higher than the 85.78% (using Clark and Curran's parser (Clark and Curran, 2007)) or 86.01% (using Petrov and Klein's parser (Petrov and Klein, 2007)) reported as the highest result on this task for CCG by Fowler and Penn (2010). This result however is lower than the 90.70% of the GCG on the same task as shown in Table 6.1. All of these results are on section 23 of the WSJ.

6.3 Significance Tests on Dependency Evaluations for GCG vs CCG

We know that GCG is about 2 points better than CCG on the Dependency Evaluations on Section 23. To confirm that GCG is in fact significantly more accurate for Dependency recovery than CCG, we conduct significance tests using Student's t-test and McNemar's test the same way we did in the Syntax Evaluations.

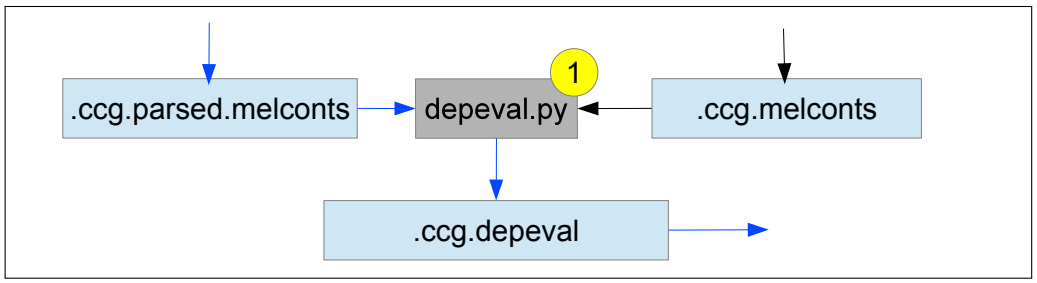


Figure 6.6: Dependency evaluation for CCG.

6.3.1 Student's t-test on Dependency Evaluations for GCG vs CCG

Figure 6.7 shows a flow to calculate Student's t-test for Dependency Evaluations between GCG and CCG. It looks almost the same as the flow to calculate Student's t-test for Syntax Evaluations between these two grammars in Figure 5.5. What is different this time are the inputs and the way to extract data points from these inputs in *step 1*. As shown on Figure 6.7, the input files ".gcg13.depeval" and ".ccg.depeval" are coming from the outputs of the Dependency Evaluations for GCG shown on Figure 6.4 and for CCG shown on Figure 6.6, respectively. These two input files have the same format as they both come out from the "depeval.py" script. This format has one sentence per line that look like the following:

```
6: (12/15) [+]'01/2/02', [+]'03/2/01', [ ]'03/3/07', [+]'
   04/2/05', [+]'05/2/06', [+]'07/2/08', [+]'07/3/04', [+]'
   08/2/09', [+]'09/2/10', [ ]'11/2/03', [+]'11/3/12', [ ]'
   13/2/11', [+]'13/3/16', [+]'14/2/15', [+]'16/2/14']
```

The first field is the sentence number or line number that can be ignored. The second field wrapped in parentheses is the ratio of the correct number of dependency relations recovered over the total number of dependency relations of the sentence. Following is the list of dependency relations that are prefixed by a "[+]" to denote a correct dependency recovered or an "[ ]" to denote a missing one. The example sentence above is for sentence number 6 that has 12 correct dependency relations recovered over a total of 15 dependency relations. Only the second field is needed for the purpose of generating data points and this is done by an "awk" script shown in *step 1* of Figure 6.7 to translate the ratio into a real decimal number. The outputs of *step 1* are fed into the same "ttest.r" script to compute the Student's t-test result in the file ".gcg13.ccg.depeval.ttestsignif". This result on Table 6.3 shows a p-value much less than 5%, so GCG is significantly better than CCG on the Dependency Relation Recovery task.

6.3.2 McNemar's test on Dependency Evaluations for GCG vs CCG

Figure 6.8 shows a flow to calculate the McNemar's test for Dependency Evaluations between GCG and CCG. Other than the fact that the inputs are now in ".melconts" which is the representation of dependency relations, this looks exactly like the McNemar's test on Syntax Evaluations between GCG and CCG on Figure 5.6. The GCG (upper) pair of inputs are coming from Figure 6.3 and the CCG (lower) pair of inputs are coming from Figure 6.5. Both

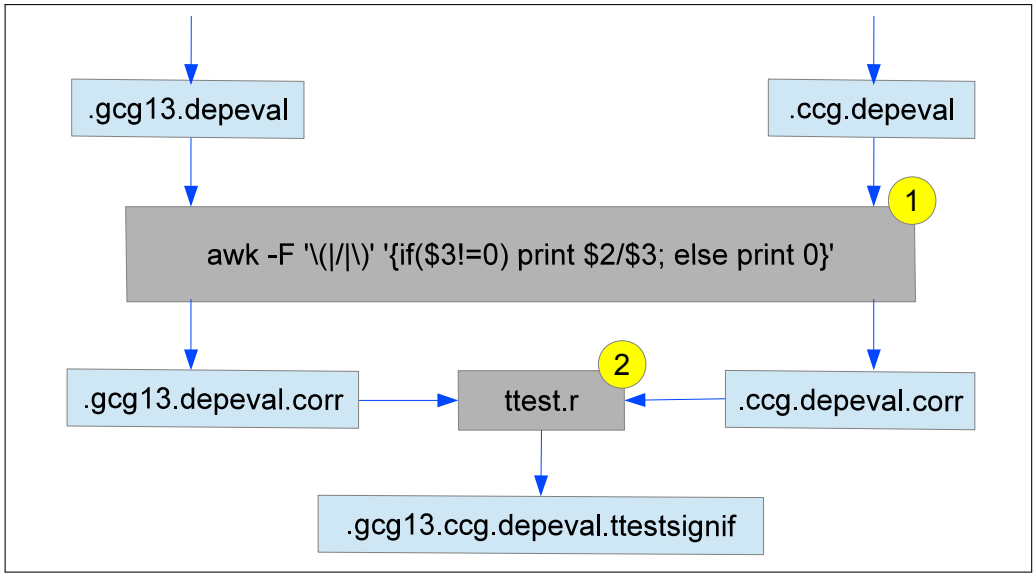


Figure 6.7: Student's t-test to measure significant for dependency evaluation between GCG and CCG. The two inputs ".gcg13.depeval" and ".ccg.depeval" are coming from the last outputs of Figure 6.4 and Figure 6.6, respectively. The final output produced is ".gcg13.ccg.depeval.ttestsignif" as shown in Table 6.3. This result shows the dependency evaluation on GCG is significantly better than that of CCG.

```
==> 131028/wsj23-inboth..gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.gcg13_depeval..ccg.
    wsj02to21-ccg-1671-5sm.fullberk.parsed.ccg_depeval..ttestsignif <==
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.gcg13_depeval.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.ccg_depeval.corr"
[1] NaN
[1] NaN

      Paired t-test

data: d1[["dat"]] and d2[["dat"]]
t = 9.135, df = 1795, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.03130461 0.04842197
sample estimates:
mean of the differences
         0.03986329
```

Table 6.3: Student's t-test result for Dependency Evaluation between GCG and CCG

```
==> 131028/wsj23-inboth..gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.depmatch..ccg.
    wsj02to21-ccg-1671-5sm.fullberk.parsed.depmatch..signif <==
[1] "131028/wsj23-inboth.gcg13.wsj02to21-gcg13-1671-5sm.fullberk.parsed.depmatch.corr"
[2] "131028/wsj23-inboth.ccg.wsj02to21-ccg-1671-5sm.fullberk.parsed.depmatch.corr"
[1] 0.3910864
[1] 0.4167131

      McNemar's Chi-squared test with continuity correction

data: d1[["dat"]] and d2[["dat"]]
McNemar's chi-squared = 4.05, df = 1, p-value = 0.04417
```

Table 6.4: McNemar test result for Dependency Evaluation between GCG and CCG

step 1 and *step 2* are exactly the same with those in Figure 5.6. The content of the result file ".gcg13.ccg.depmatch.signif" is shown in Table 6.4. The p-value is 4.4%, less than 5%. This is once again confirming that GCG is significantly better than CCG on the dependency recovery task.

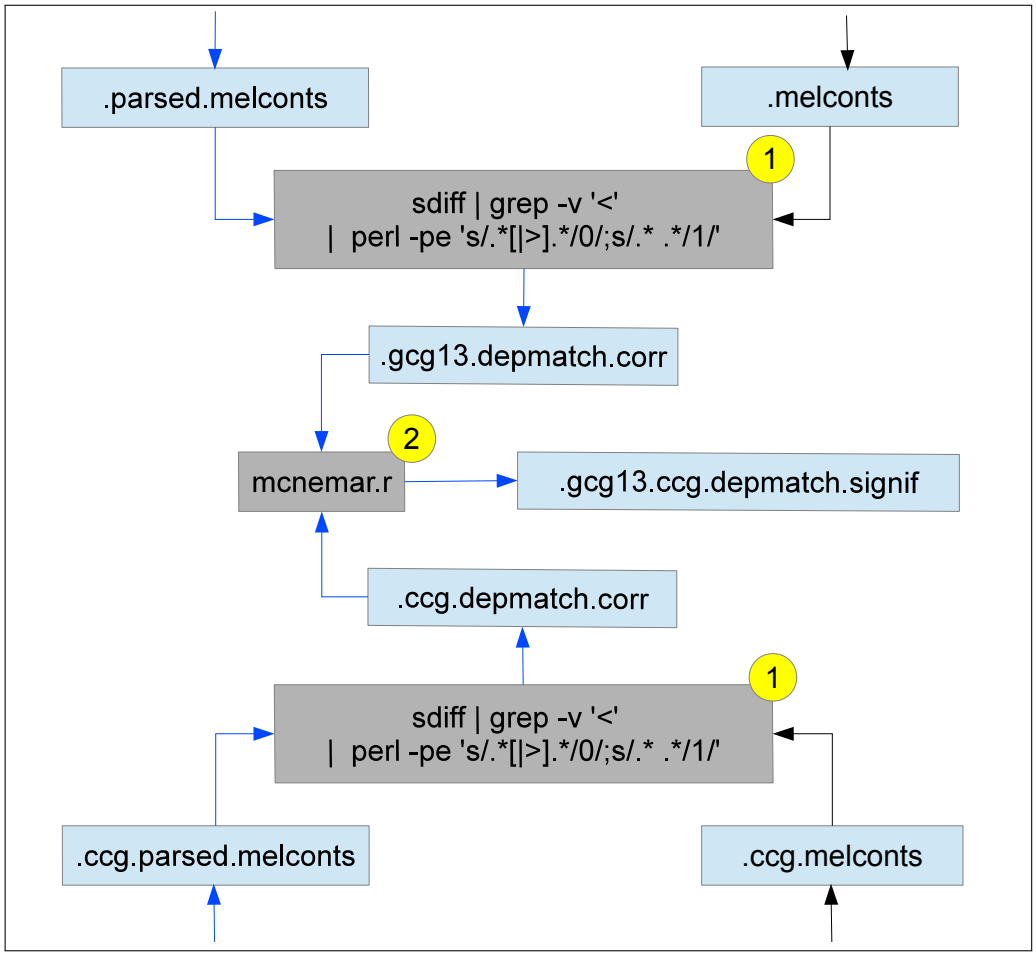


Figure 6.8: McNemar's significance test for dependency evaluation between GCG and CCG. The upper pair of inputs are ".parsed.melconts" and ".melconts", coming from the outputs toward the end of Figure 6.3 to represent the dependency parsing on GCG. The lower pair of inputs are ".ccg.parsed.melconts" and ".ccg.melconts", coming from the outputs toward the end of Figure 6.5 to represent the dependency parsing on CCG. The *step 1* is duplicated in 2 places to make it clear by reducing the number of its inputs and outputs. This *step 1* extracts only the perfect matches in dependency parsing on either GCG or CCG. The ".corr" file name implies taking only the perfectly "correct" dependency parsed sentences. The *step 2* is an R script to compute the McNemar significance test. This result as shown in Table 6.4, one more time, confirms that GCG is significantly more accurate than CCG on dependency parsing.

Chapter 7

Unbounded Dependency Evaluations

The dependency relations evaluated in the previous chapter are general syntactic dependencies because they are direct relations from the argument to the predicate. By direct, it means there is a single arrow going from the argument word to the predicate word on a dependency grammar representation, or the phrase headed by the argument is a sibling or a direct child of the phrase headed by the predicate on a phrase structure grammar representation. Most of these dependencies are easier to recover due to this direct type of relationship and their readiness to be extracted from the syntax representation. This chapter focus the evaluation of our GCG grammar on its capability to recover a difficult subset of these dependencies known as *unbounded dependencies* or sometimes *long-range dependencies*.

Unbounded dependencies are dependencies between constituents and points of attachment that have other constituents syntactically intervening. For example, the sentence *What does the First Amendment protect?* has a preposed constituent *what* that functions as a direct object of the transitive verb *protect*. The long range between the source and destination of this type of dependency, paired with the relatively low probability of their occurrence in the language, and the fact that filler-gap annotations in syntactic resources such as the Penn Treebank are often stripped out, makes it very difficult for parsers to recognize this type of dependency correctly. While difficult to parse, this type of dependency is vital to the meaning of the sentence and of great importance in applications such as question answering and information extraction.

Many current interpretation models are based on PCFGs, trained on syntactic annotations

from the Penn Treebank (Marcus et al., 1993). These often recover dependencies as a post-process to parsing, and often are not able to retrieve unbounded dependencies if they are optimized on syntactic representations that leave these dependencies out.

Categorial grammars, on the other hand, have well-defined unbounded dependency representations based on functor-argument relations in a small and easily-learnable set of composition operations. Such grammars — in particular, Combinatory Categorial Grammar (CCG) (Steedman, 2000, Clark and Curran, 2007) — do well on unbounded dependency recovery tasks (Rimell et al., 2009) but not as well as models based on Head Driven Phrase-structure Grammar (HPSG) (Pollard and Sag, 1994, Miyao and Tsujii, 2005), given the same training. This may be attributed to implicit tradeoffs in many categorial frameworks that minimize the number of composition operations at the expense of large numbers of possible categories for each lexical item, which may lead to sparse data effects in training. HPSG models, in contrast, maintain a relatively large number of composition operations and a relatively small set of possible lexical categories, which are then used in a wider set of contexts.

Can categorial grammars, which have well-studied semantic representations and are well suited for interpretation, obtain better performance on a general unbounded dependency extraction task if it adopts an HPSG-like strategy of re-using types in various contexts? The attempt to answer that question is the driving force for us to develop our GCG which, like HPSG, is generalized to limit the number of categories used to those needed to enforce grammatical constraints, but like other categorial grammars, imposes a small, uniform, and easily learnable set of semantic composition operations based on functor-argument relations.

The previous two evaluations on the syntax parsing and dependency recovery have shown the promising, but this evaluation on the recovery of unbounded dependencies is an important task of the grammar as a main building block for a state-of-the-art interpretation model. To go on with this evaluation, we leverage a research from Rimell et al. (2009) where the authors had built a minimal corpus consisting of 700 sentences representing the most common 7 types of unbounded dependency (100 sentences each type, 80 for development and 20 for test, and nothing else). This corpus is referred to as "LR" (cf. long range) in Figure 7.1. We also put the result of our system into comparison with that of 7 other systems reported by these authors. Below is a brief description of these 7 common types of unbounded dependency constructions.

- Object extraction from a relative clause (Obj RC): This construction has a relative clause headed by a relative pronoun that is extracted from the object position of the clause. For

example, *"The* **cart** *that the horse* **pulled** *broke."* has the relative clause *"that the horse pulled"* where the pronoun "*that*" is extracted from the object position of that clause. Possible pronouns in English are *wh*-words and *that*.

- Object extraction from a reduced relative clause (Obj Red): This construction is similar to the one above but without the relative pronoun, hence the name "reduced relative clause". For example, *"The* **cart** *the horse* **pulled** *broke."* has the reduced relative clause *"the horse pulled"* where the omitting pronoun at the beginning of the clause is extracted from the object position of that clause.

- Subject extraction from a relative clause (Sbj RC): This construction has a relative clause headed by a relative pronoun that is extracted from the subject position of the clause. For example, *"The* **horse** *that* **pulled** *the cart died."* has the relative clause *"that pulled the cart"* where the pronoun "*that*" is extracted from the subject position of that clause. Note that a pronoun is required in this construction, so English does not have subject extraction from a reduced relative clause.

- Free relative (Free): This construction has a relative pronoun without an antecedent. For example, "*I know* **what** *he* **did.**" has the pronoun "**what**" having no antecedent and can be interpreted as "*the thing*" or something similar.

- Object *wh*-question (Obj Q): This construction has the *wh*-word play the role of a semantic object of the main verb or a preposition of the main verb. For example, "**what** *did you* **do***?*" or "*what* **hotel** *did you stay* **in***?*".

- Right node raising (RNR): This construction has coordinated phrases from which a shared component moves to the right, e.g. "*Mary* **peeled** *and Pete* **ate** *the* **shrimps**." This example has RNR at the sentence level, but it could also happen at various levels like verb phrases (**Peeling** *and* **eating** *those shrimps is time consuming.*), noun phrases (*The* **old** *and the* **new** *iPad displayed side-by-side.*), or prepositional phrases (*I left before* **his** *and after* **her arrival**.)

- Subject extraction from an embedded clause (Sbj Embed): This construction has a semantic subject which is extracted across two clause boundaries, e.g. "*There was some* **money** *in my wallet which I thought* **was** *not there.*" This example shows that "**money**" from the main clause is a semantic subject of the verb "**was**" (be) on the embedded clause.

Figure 7.1 shows the flow of this evaluation. The two additional shaded boxes of "LR test raw" (contains raw sentences without any annotation, as in our .sents files) and "LR test" (contains the same sentences with their accompanied gold unbounded dependencies) are coming from the LR corpus.

- *Step 1* through *Step 6* are exactly the ones being used in the GCG parsing step shown in Figure 5.1, but this time being used only for the training data to build the model for the parser. This is because the test data is coming from the long-range (LR) corpus studied by Rimell et al. (2009), not PTB. We evaluate our GCG on the LR corpus in order to compare with the results of 7 other systems as reported by Rimell et al. (2009).

- *Step 7* takes input from the raw test data without any annotation from the long-range (LR) corpus studied by Rimell et al. (2009) and replaces "(" with "-LRB-" and ")" with "-RRB-" to be consistent with the GCG grammar model learnt on the Berkley parser.

- *Step 8* is the same script "trees2melconts.py" used to extract dependency relations for GCG in Figure 6.3, but this time is used with option "-c" to mean shifting the head of the conjunction from the last conjunct to the conjunction word. In GCG formalism, the head of a conjunction, i.e. *A and B*, is *B*, but the gold standard in LR corpus assumed the head to be the conjunction word *and*. The option "-c" helps shift the conjunction head to make a fair comparison with other systems studied by Rimell et al. (2009). To note this difference in the type of dependencies extracted by this script, we used ".tbconts" extension for the output file name instead of the standard ".melconts".

- *Step 9* is a Python script "convertGoldUnbound.py" to translate the representations of unbounded dependencies into our ".tbconts" format. The format of dependency relations used by Rimell et al. (2009) is that of de Marneffe et al. (2006) but only for the unbounded dependencies, not the whole sentence. This Python script deterministically maps from this format to our numeric relations format. This simplification of dependency labels to numbers can be losslessly reversed by looking at the categories of the involved predicates. Specifically, the mapping is done as follows:

  - Dependencies 'nsubj' and 'nsubjpass' are mapped to a '1' relation.
  - Dependencies 'dobj', 'pobj', 'infmod', 'xcomp', and 'obj2' are mapped to a '2' relation.

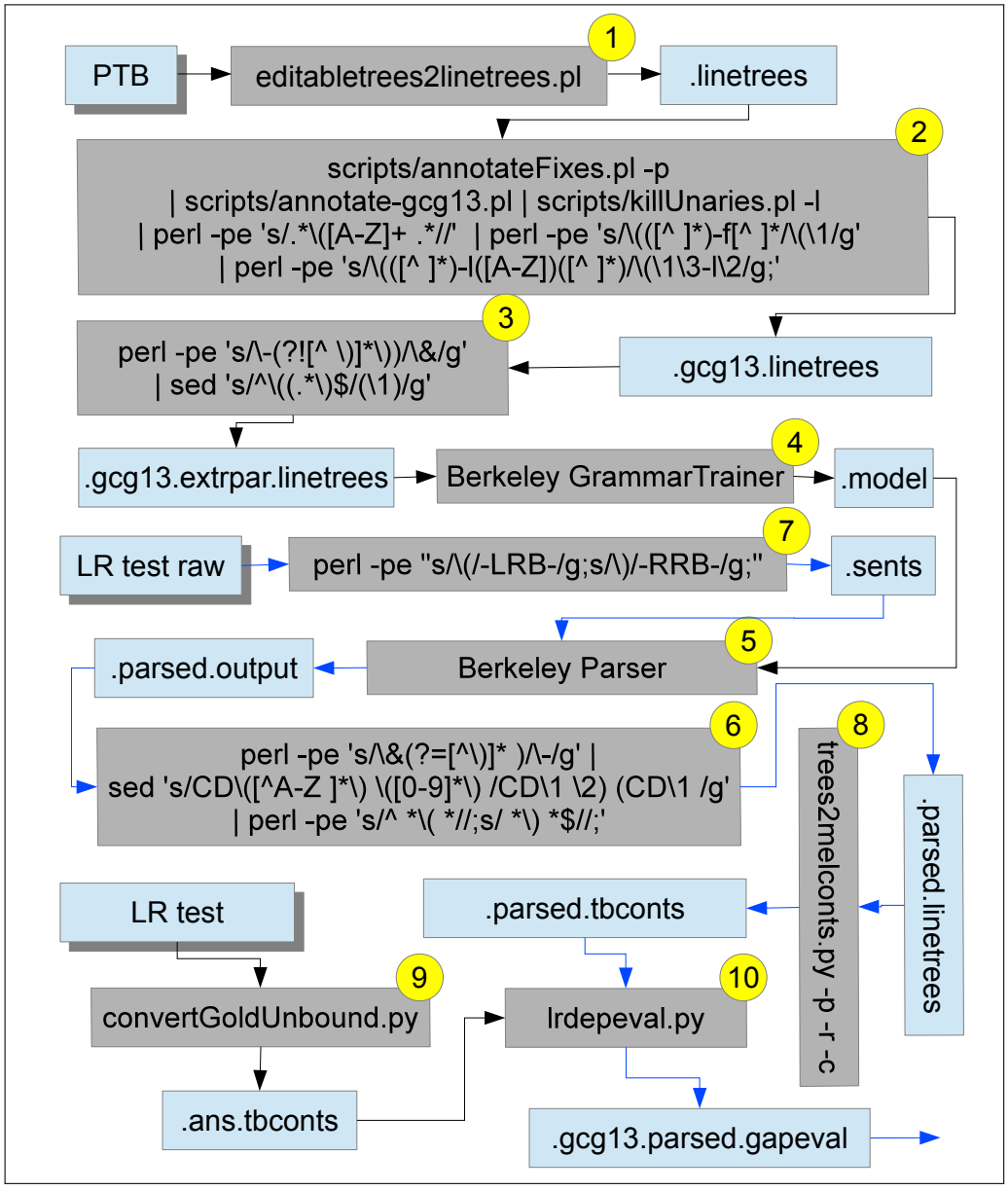


Figure 7.1: Filler gap evaluation for GCG.

  - Dependencies ‘advmod’, ‘prep’, ‘amod’, ‘partmod’, ‘auxpass’, ‘iobj’, and ‘nn’ are also mapped to a ‘1’ relation with the direction of the dependency reversed. This reversal of direction for modifier dependencies is similar to that described in dependency accounts of Tree Adjoining Grammars (Joshi, 1985, Candito and Kahane, 1998).
  - Dependency ‘cop’ is also mapped to a ‘2’ relation with the direction of the dependency reversed.

- *Step 10* takes the hypothesis data from ".parsed.tbconts" and gold data from ".ans.tbconts" to count the recall and precision, and to compute F-score for the task. Due to differences between the de Marneffe et al. (2006) dependency representation and that of our current system, some deterministic modifications were required for evaluation against the Rimell et al. (2009) corpus.[1]

  1. If the hypothesized target of a dependency is a conjunction, the dependencies to each of its conjuncts are hypothesized instead;
  2. If the target of a dependency is a relativizer or a relative pronoun, the predicate it modifies is used in its place; and
  3. If the source predicate of a dependency has a category of **O**, the predicate that depends on the hypothesized target is hypothesized as the target.

The results on the 7 types of unbounded dependencies are shown in Table 7.1 in context of the results reported by Nivre et al. (2010), an extended version to include the Malt and MST parsers to the list of systems first chosen to study for this task by Rimell et al. (2009). This extension brought the total numbers of systems to 7. Below are the short descriptions of the grammar and parsing technique for each of these 7 systems to help put the comparison of these systems into the perspective of recovering unbounded dependencies.

1. Enju: This system used Head-Driven Phrase Structure Grammar (HPSG) introduced by Pollard and Sag (1994). Enju is a well known representative wide-coverage HPSG parser from Miyao and Tsujii (2005). These authors took the same approach to reannotate PTB

[1] This automated scoring makes the evaluation less generous than the manual output interpretations given in Rimell et al. (2009) and Nivre et al. (2010), but has the advantage of being easily reproducible.

into an HPSG formalism hence making the first wide-coverage parser for HPSG; and rolled out their own parser for this grammar. This parser produced head-word dependencies reflecting the underlying predicate-argument structure of sentence making it a good candidate for the unbounded dependencies recovery task.

2. C&C: The grammar being used by this system is CCG (Hockenmaier and Steedman, 2007), a well known variation of a radically lexicalist categorial grammar. Like our GCG, this CCG was the result of reannotating PTB and was designed to specifically capture the unbounded dependencies. Unlike our GCG, we are more moderately lexicalist, i.e. we try to use fewer lexical categories in much the same way as "signs" in HPSG, and prefer to use a richer set of inference rules based on the underlying syntax structure. This system used the "candc" parser from Clark and Curran (2007).

3. Malt: First introduced to this evaluation by Nivre et al. (2010). This is a dependency parser implementing the parsing models introduced by Nivre et al. (2006a,b). It is a data-driven parser generator for dependency parsing and is categorized as a transition-based parsing system producing dependency trees by greedily transitioning through abstract state machines. Transition-based parsers learn models to predict the next state given the current state, the features over the history of parsing decisions, and the input sentence. The greedy nature of these systems make them fast, but can lead to enormous error propagation if they pick some incorrect states at early predictions.

4. MST: Also first introduced to this evaluation by Nivre et al. (2010). This is another dependency parser implemented by McDonald (2006). It is categorized as a graph-based parsing system that learns and finds directed maximum spanning trees from a dense graph representation of the sentence. In term of complexity, this problem is a typical NP-hard problem. This system therefore must try to limit the scope of their features to a small number of adjacent arcs (usually two) and/or turn to approximation algorithms for inference (McDonald and Pereira, 2006).

5. Stanford: This Stanford parser represents PTB parsers, exemplified by Collins (1997) and Charniak (2000). This parser works on the straight PTB grammar, but ignores all the trace information, so is not ideal for the unbounded dependencies recovery task, but was selected by Rimell et al. (2009) for this study because of its popularity. This is

a phrase-structure, not dependency, parser. The phrase structure trees output from this parser were piped through a set of manually defined rules to extract dependencies needed for the evaluation (de Marneffe et al., 2006).

6. DCU: This system is from Cahill et al. (2004). It is a post processor of PTB parsers such as that of Charniak (2000), and based on Lexical-Functional Grammar (Kaplan and Bresnan, 1982, Dalrymple, 2001). The authors tried to exploit functional tags (e.g. **-LOC**, **-TMP**, **-TPC**, etc.) and traces from PTB to implement an automatic LFG f-structure annotation algorithm that associates nodes in PTB trees with f-structure annotations in the form of attribute-value structure equations representing abstract predicate-argument structure or dependency relations. They then extracted LFG subcategorization frames and paths linking unbounded dependency reentrancies from f-structures generated to build a finite approximation algorithm to recognize unbounded dependencies. These authors prefer the term "long distance dependency" (LDD) instead of the more commonly used "unbounded dependency."

7. RASP: The name stands for Robust Accurate Statistical Parsing. The system being evaluated is the second release (RASPv2) developed by Briscoe et al. (2006). This parser consists of a POS tag-sequence grammar, a statistical parse selection component, and a robust partial-parsing technique which allows it to always return a parsed result for input sentences even when they do not obtain a full spanning analysis according to the grammar. This system is a shallow parser and was not designed to capture many of the unbounded dependencies being studied, but was included based on its popularity.

Despite a wide gap of almost 5% between our GCG system and the second best on LR test data, we would like to conduct significance tests in order to confirm that GCG is the best choice for unbounded dependency recovery in general. However, our effort to contact the authors asking for data points of that study failed, so we cannot move forward with significance tests.

	Obj RC	Obj Red	Sbj RC	Free	Obj Q	RNR	Sbj Embed	Total
Enju	47.3	65.9	82.1	**76.2**	32.5	47.1	32.9	54.4
C&C	**59.3**	62.6	80.0	72.6	27.5	**49.4**	22.4	53.6
Malt	40.7	50.5	**84.2**	70.2	16.2	39.7	23.5	46.4
MST	34.1	47.3	78.9	65.5	18.8	45.4	37.6	46.1
Stanford	22.0	1.1	74.7	64.3	41.2	45.4	10.6	38.1
DCU	23.1	41.8	56.8	46.4	27.5	40.8	5.9	35.7
RASP	16.5	1.1	53.7	17.9	27.5	34.5	15.3	25.3
This system	52.7	**71.4**	78.9	71.4	**52.5**	36.2	**51.8**	**59.3**

Table 7.1: Unbounded dependency results compared to those of other systems studied by Rimell et al. (2009) and Nivre et al. (2010) over a variety of constructions: object extraction from relative clauses (Obj RC), object extraction from reduced relative clauses (Obj Red), subject extraction from relative clauses (Sbj RC), free relatives (Free), object wh-questions (Obj Q), right node raising (RNR), and subject extraction from embedded clauses (Sbj Embed). Evaluated parsers are C&C (Clark and Curran, 2007), Enju (Miyao and Tsujii, 2005), DCU (Cahill et al., 2004), Rasp (Briscoe et al., 2006), Stanford (Klein and Manning, 2003), MST (McDonald, 2006), Malt (Nivre et al., 2006a,b). This system used the Berkley parser (Petrov and Klein, 2007) run on the reannotated categorial grammar.

Chapter 8

Conclusion and Discussion

This thesis has described a Generalized Categorial Grammar (GCG) which, like other categorial grammars, imposes a small, uniform, and easily learnable set of semantic composition operations based on functor-argument relations, but like HPSG, is generalized to limit the number of categories used to those needed to enforce grammatical constraints.

The thesis has also described a system for automatically reannotating syntactically-annotated corpora for the purpose of refining linguistically-informed phrase structure analyses of various phenomena. In particular, it described a method for implementing syntactic analyses of various phenomena through automatic reannotation rules, which operate deterministically on a corpus like the Penn Treebank (Marcus et al., 1993) to produce a corpus with desired syntactic analyses. This reannotated corpus is then used to define a probabilistic grammar which is automatically annotated with additional latent variable values (Petrov and Klein, 2007) and used to parse the constituent and syntactic dependencies from input sentences of the Wall Street Journal and from a minimal but special corpus introduced by (Rimell et al., 2009) that contains only sentences having Object extraction from a relative clause, Object extraction from a reduced relative clause, Subject extraction from a relative clause, Free relatives, Object wh-questions, Right node raising, and Subject extraction from an embedded clause. This corpus was designed specifically to test various parsers on their capability to recover these unbounded dependencies as studied by (Rimell et al., 2009, Nivre et al., 2010).

This system achieves significantly better result on syntax and semantic dependencies parsing compared to the main stream Combinatorial Categorial Grammar (CCG) system from (Steedman, 2000, Clark and Curran, 2007). It also scores the best result on the unbounded dependency

parsing accuracy favorably comparable to all the 7 major systems recently studied by Rimell et al. (2009) and Nivre et al. (2010) on this same task.

References

Bach, E. (1981). Discontinuous constituents in generalized categorial grammars. *Proceedings of the Annual Meeting of the Northeast Linguistic Society (NELS)*, 11:1–12.

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58.

Briscoe, T., Carroll, J., and Watson, R. (2006). The second release of the RASP system. In *Proceedings of the Interactive Demo Session of COLING/ACL-06*, Sydney, Australia.

Cahill, A., Burke, M., Genabith, J. V., and Way, A. (2004). Long-distance dependency resolution in automatically acquired wide-coverage PCFG-based lfg approximations. In *Proceedings of the 42nd Meeting of the ACL*, pages 320–327, Barcelona, Spain.

Candito, M.-H. and Kahane, S. (1998). Can the TAG derivation tree represent a semantic graph? In *Proceedings of the TAG+4 Workshop*, University of Pennsylvania.

Charniak, E. (2000). A maximum-entropy inspired parser. In *Proceedings of the First Meeting of the North American Chapter of the Association for Computational Linguistics (ANLP-NAACL'00)*, pages 132–139, Seattle, Washington.

Clark, S. and Curran, J. R. (2007). Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33:493–552.

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL'97)*.

Dalrymple, M. (2001). *Lexical-Functional Grammar*. Wiley Online Library.

de Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC 2006*.

Fowler, T. A. D. and Penn, G. (2010). Accurate context-free parsing with combinatory categorial grammar. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 335–344.

Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985). *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA.

Hockenmaier, J. and Steedman, M. (2007). Ccgbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Joshi, A. K. (1985). How much context sensitivity is necessary for characterizing structural descriptions: Tree adjoining grammars. In D. Dowty, L. K. and Zwicky, A., editors, *Natural language parsing: Psychological, computational and theoretical perspectives*, pages 206–250. Cambridge University Press, Cambridge, U.K.

Kaplan, R. and Bresnan, J. (1982). Lexical functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge MA.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan.

Komagata, N. (2002). Coordination of unlike (?) categories: How not to distinguish categories.

Kubota, Y. and Levine, R. (2012). Gapping as like-category coordination. *Logical Aspects of Computational Linguistics*, (7351):Al135–150.

Lambek, J. (1958). The mathematics of sentence structure. *American mathematical monthly*, pages 154–170.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330.

McDonald, R. (2006). *Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing*. PhD thesis, University of Pennsylvania.

McDonald, R. T. and Pereira, F. C. (2006). Online learning of approximate dependency parsing algorithms. In *EACL*.

Mel'čuk, I. (1988). *Dependency syntax: theory and practice*. State University of NY Press, Albany.

Mihalicek, V. and Pollard, C. (2010). Distinguishing phenogrammar from tectogrammar simplifies the analysis of interrogatives. In *Formal Grammar*, pages 130–145.

Miyao, Y., Ninomiya, T., and Tsujii, J. (2004). Corpus-oriented grammar development for acquiring a head-driven phrase structure grammar from the Penn Treebank. In *Proceedings of the First International Joint Conference on Natural Language Processing (IJCNLP-04)*, pages 684–693, Hainan Island, China.

Miyao, Y. and Tsujii, J. (2005). Probabilistic disambiguation models for wide-coverage HPSG parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 83–90, Michigan, Ann Arbor.

Nguyen, L., van Schijndel, M., and Schuler, W. (2012). Accurate unbounded dependency recovery using generalized categorial grammars. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING '12)*, pages 2125–2140, Mumbai, India.

Nivre, J., Hall, J., and Nilsson, J. (2006a). Maltparser: A data-driven parser-generator for dependency parsing. In *In Proc. of LREC*, pages 2216–2219.

Nivre, J., Hall, J., Nilsson, J., Eryiğit, G., and Marinov, S. (2006b). Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 221–225, Stroudsburg, PA, USA. Association for Computational Linguistics.

Nivre, J., Rimell, L., McDonald, R., and Gómez-Rodríguez, C. (2010). Evaluation of dependency parsers on unbounded dependencies. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING'10)*, pages 833–841.

Oehrle, R. T. (1994). Term-labeled categorial type systems. *Linguistics and Philosophy*, 17(6):633–678.

Parsons, T. (1990). *Events in the Semantics of English*. MIT Press.

Petrov, S. and Klein, D. (2007). Improved inference for unlexicalized parsing. In *Proceedings of NAACL HLT 2007*, pages 404–411, Rochester, New York. Association for Computational Linguistics.

Pollard, C. and Sag, I. (1994). *Head-driven Phrase Structure Grammar*. University of Chicago Press, Chicago.

Rimell, L., Clark, S., and Steedman, M. (2009). Unbounded dependency recovery for parser evaluation. In *Proceedings of EMNLP 2009*, volume 2, pages 813–821.

Sag, I. A., Gazdar, G., Wasow, T., and Weisler, S. (1985). Coordination and how to distinguish categories. *Natural Language & Linguistic Theory*, 3:117–171.

Shieber, S. M., Schabes, Y., and Pereira, F. C. (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36.

Steedman, M. (2000). *The syntactic process*. MIT Press/Bradford Books, Cambridge, MA.

Appendix A

Additional Reannotation Rules

Other than the rules categorized and mentioned in Chapter 4, there are more rules that we used in the system that are considered miscellaneous or just other variations of the ones categorized. These rules will be listed here for reference.

A.1 Other reannotation rules for initial and final argument attachment

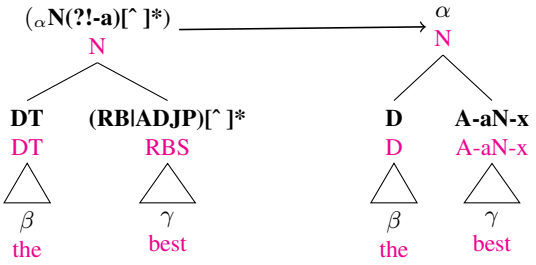


Figure A.1: Branch **N** -> **D A-aN-x**: 'the best' construction. This rule consists of a type changing rule to change an **A-aN** to an **N-aD** and an initial argument attachment rule Aa.

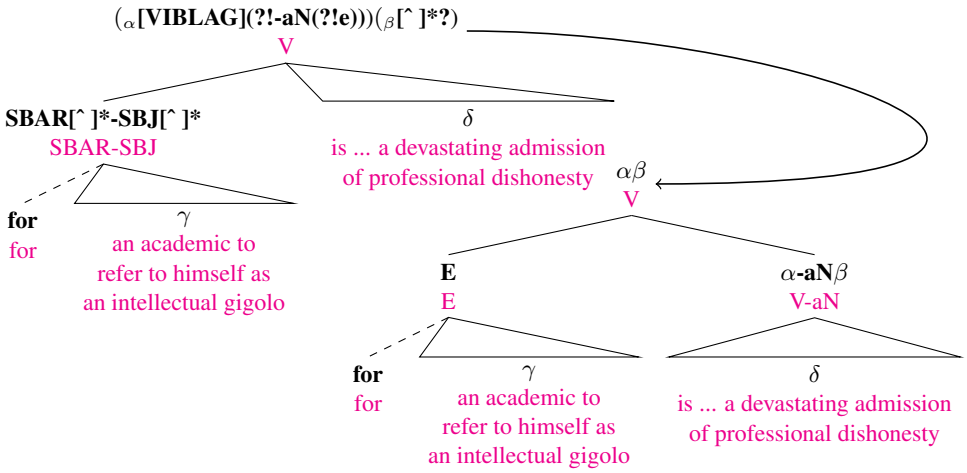


Figure A.2: **[VIBLAG]** sentence: branch off initial **E** subject. This rule embedded a type changing rule to change an **E** to an **N** to enable a final argument attachment rule Ae.

A.2 Other reannotation rules for initial and final modifier attachment

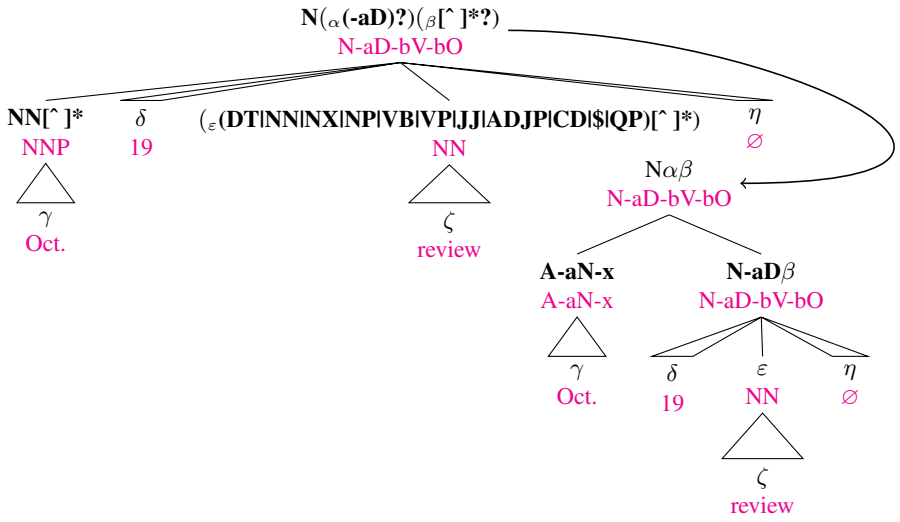


Figure A.3: Branch off initial modifier **A-aN-x**. If ε=**QP** then its leftmost child in ζ must be of category **$**. This example has α=-aD, β=-bV-bO. This is an initial modifier attachment rule Ma.

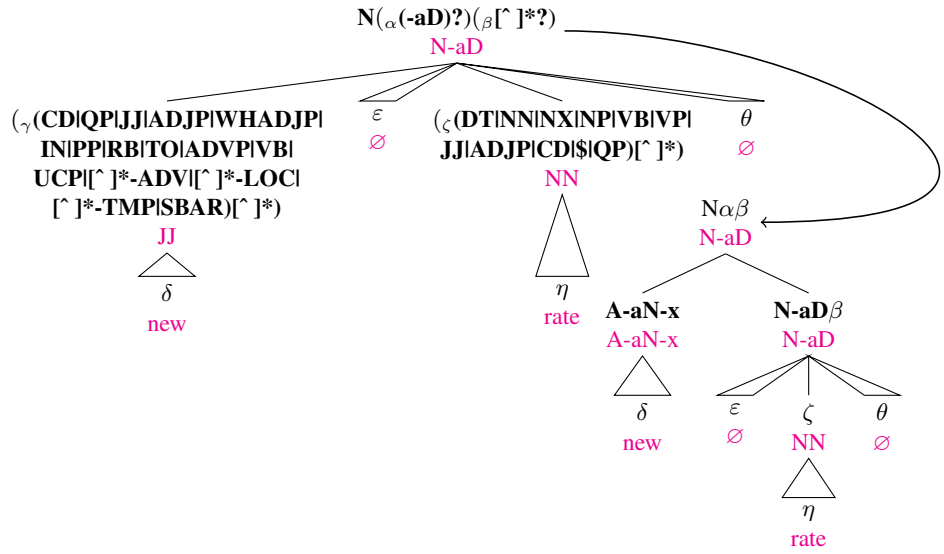


Figure A.4: Branch off initial modifier **A-aN-x**. If γ=**SBAR[^]*** then its left-most child in δ must be of category **IN** having child not **that**. If ζ=**QP** then its leftmost child in η must be of category **$**. This example has α=-aD, β=-bV-bO. This is an initial modifier attachment rule Ma.

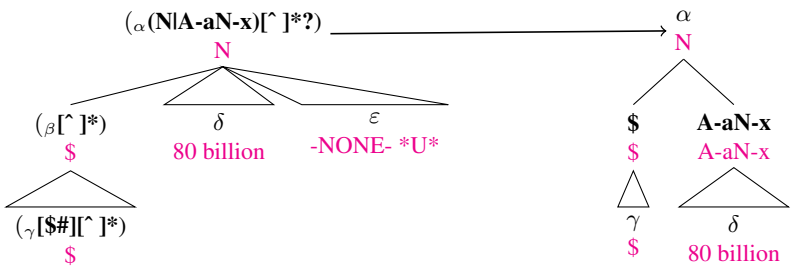


Figure A.5: Rebinarize currency unit followed by **QP**. If $\varepsilon \neq \varnothing$ then ε=**(-NONE- *U*)**. This rule embedded a type changing rule to change a **$** to an **N** that will enable a final modifier attachment rule Me.

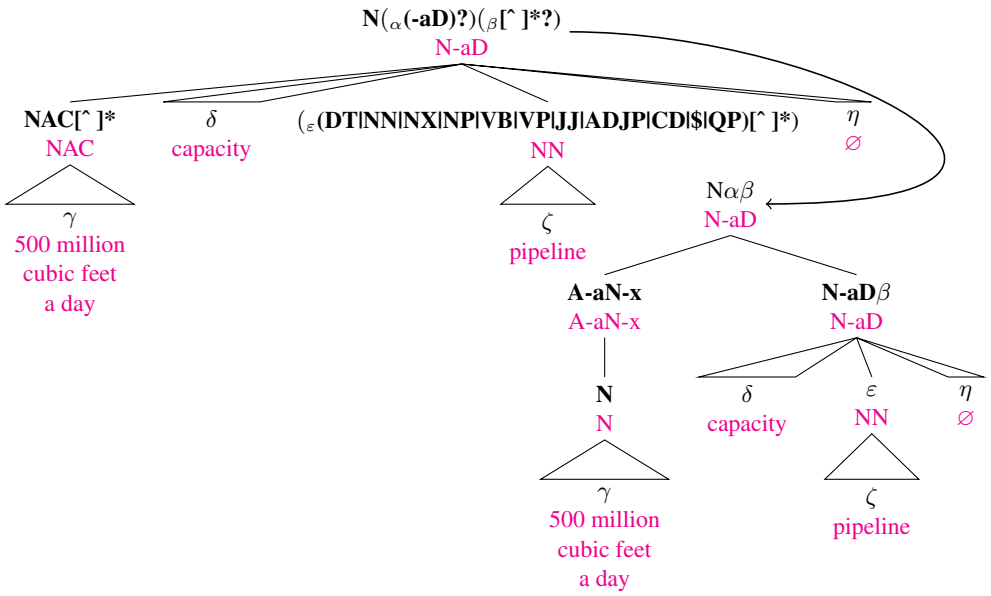


Figure A.6: Branch off initial modifier **A-aN-x**. If ε=**QP** then its leftmost child in ζ must be of category **$**. This example has α=-aD, β=$\varnothing$. This rule embedded a type changing rule to change an **N** to an **A-aN** to enable an initial modifier attachment rule Ma.

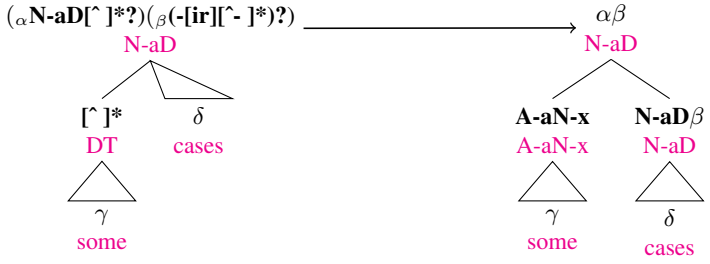


Figure A.7: Branch off initial modifier **A-aN-x**. Both γ and δ are not $\varnothing$. This example has α=N-aD and β=$\varnothing$. This is an initial modifier attachment rule Ma.

A.3 Other reannotation rules for filler attachment

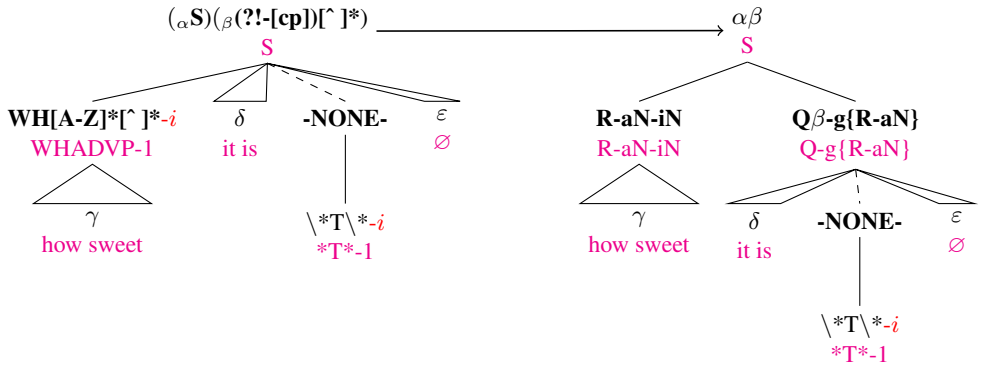


Figure A.8: Content question: branch off initial interrogative **R-aN**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fc rules **R-aN-iN** + **Q-g{R-aN}** = **Q-iN** that is followed by a type changing rule to change a **Q-iN** to an **S**.

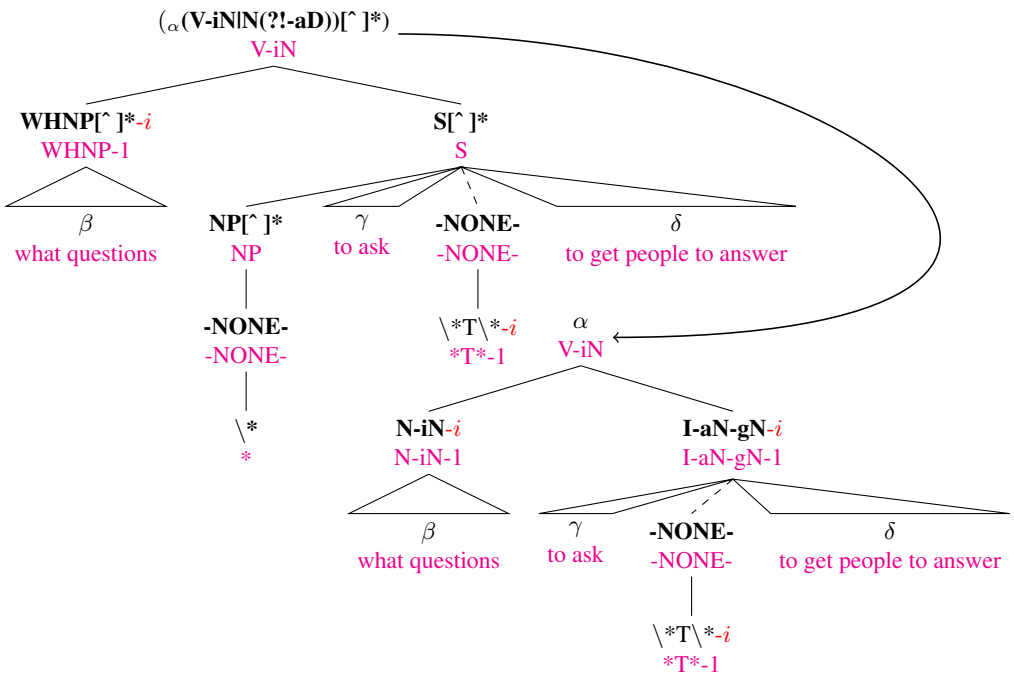


Figure A.9: Embedded question / nom clause: branch off initial interrogative **N** and final modifier **I-aN** with **N** gap. This rule combines an Fc rule **N-iN** + **I-aN-gN** = **I-aN-iN** that is followed by a type changing rule to change an **I-aN** to a **V**.

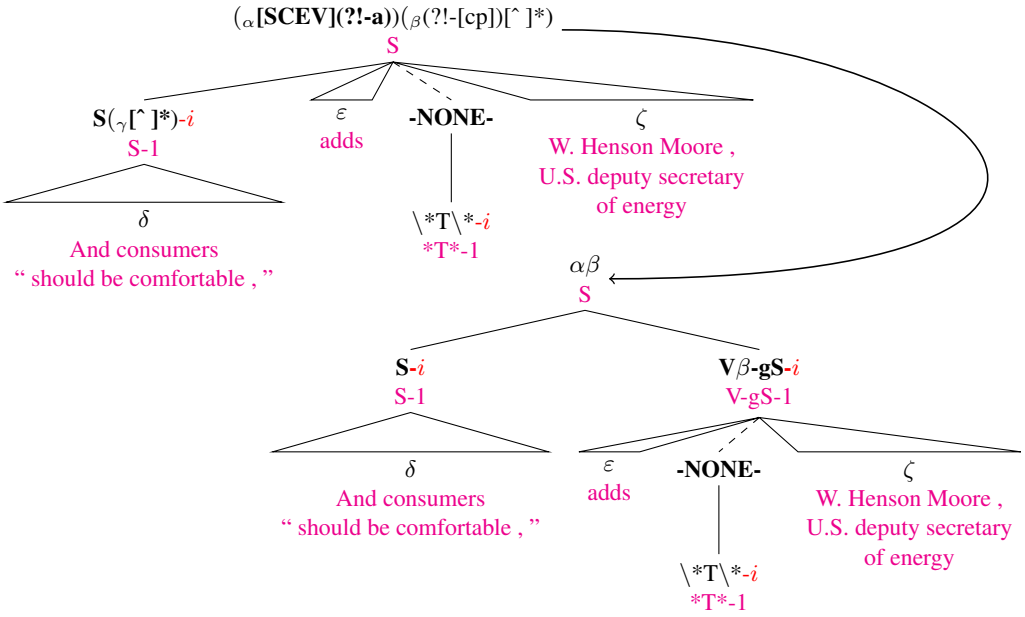


Figure A.10: Topicalized sentence: branch off initial topic **S** (possibly quoted). γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules.

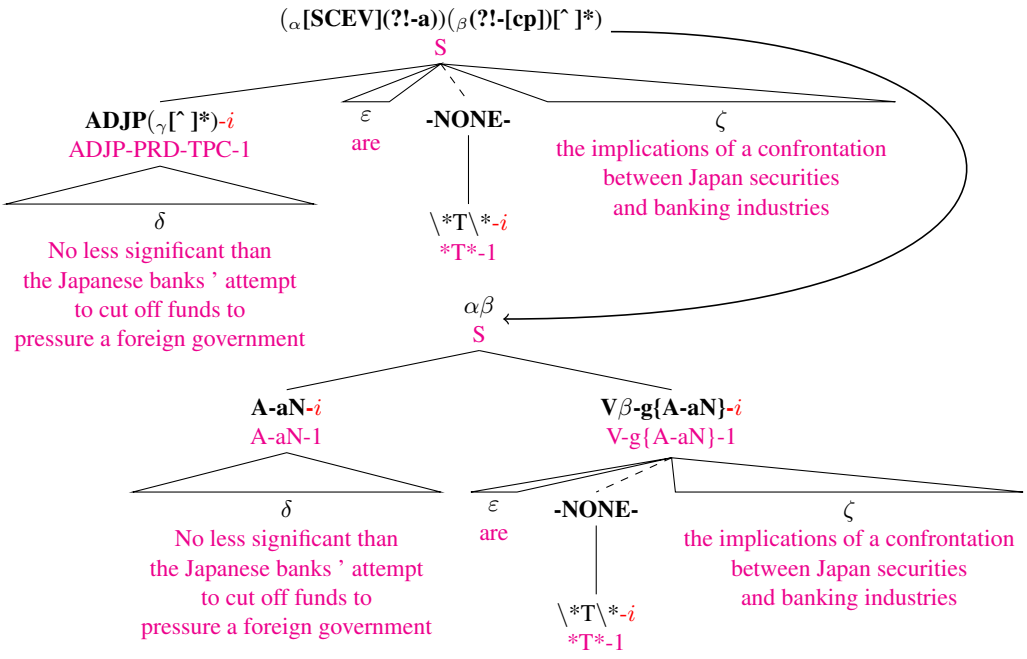


Figure A.11: Topicalized sentence: branch off initial topic **A-aN**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules.

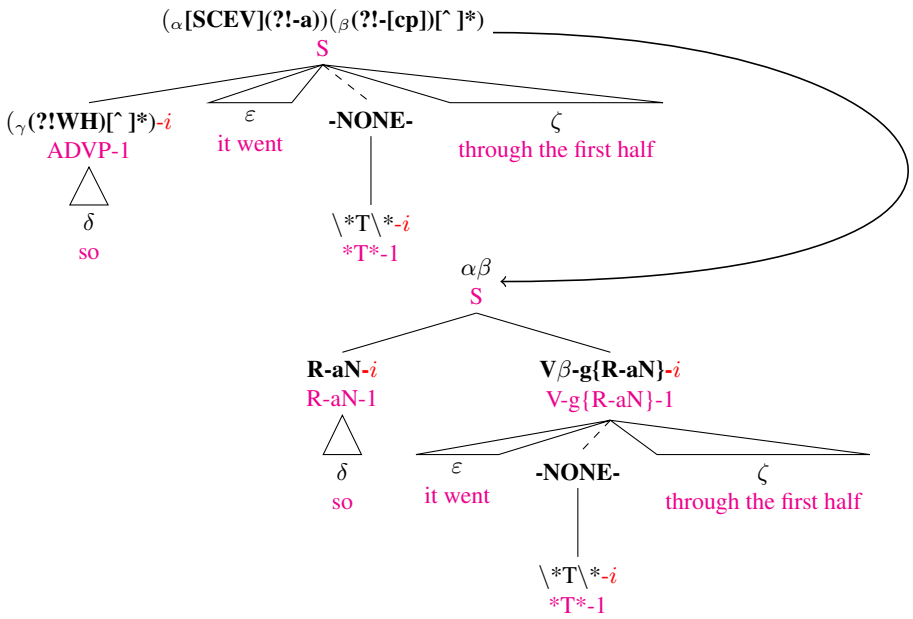


Figure A.12: Topicalized sentence: branch off initial topic **R-aN**. γ should not contain **-SBJ**. This example has α=S, β=$\varnothing$, and i=1. This is one of the Fd rules.

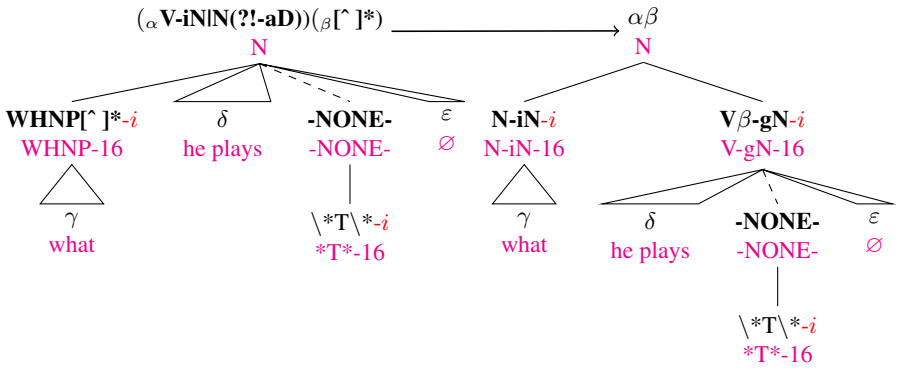


Figure A.13: Embedded question / nom clause: branch off initial interrogative **N**. This is an Fe rule.

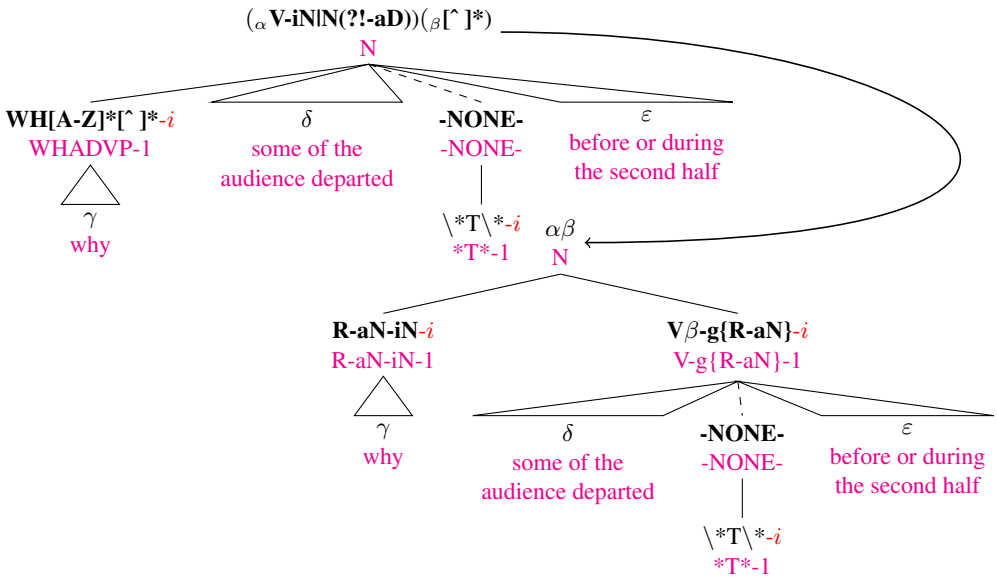


Figure A.14: Embedded question / nom clause / nom clause modifier: branch off initial interrogative **R-aN**. This is an Fe rule

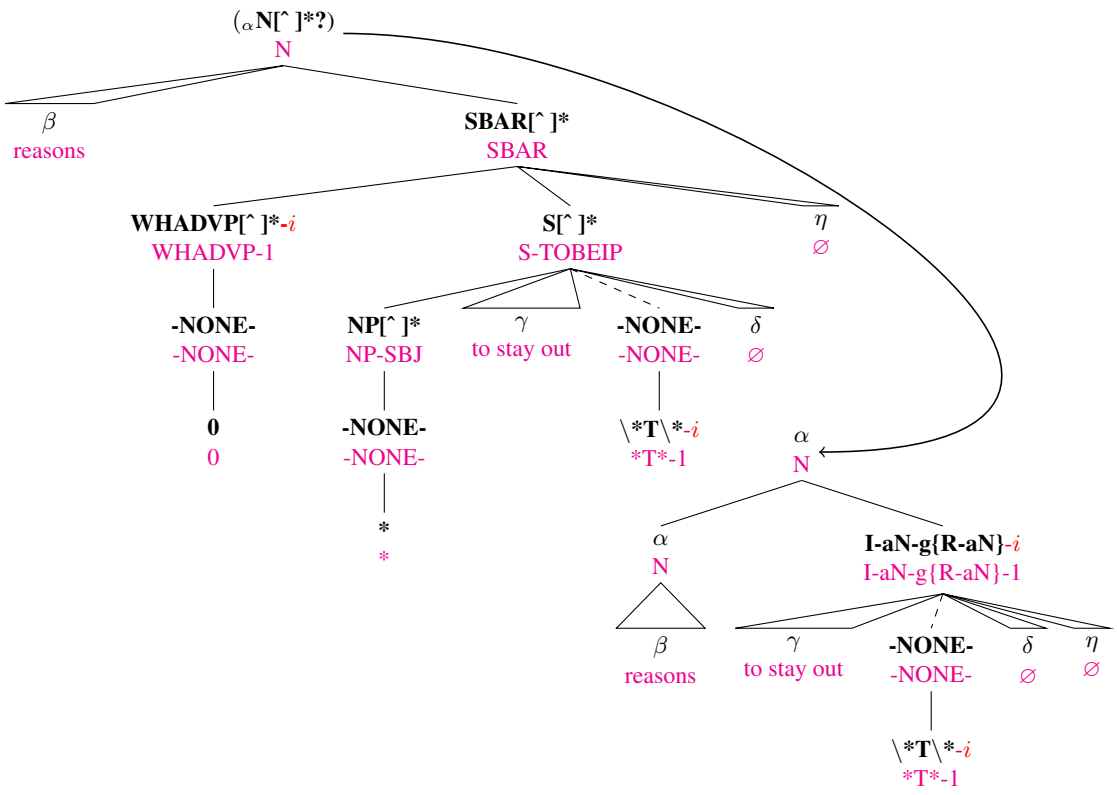


Figure A.15: Branch off final **SBAR** as modifier **I-aN-g{R-aN}**. This is an Fa rule **N + I-aN-gN = N**.

A.4 Other reannotation rules for type changing

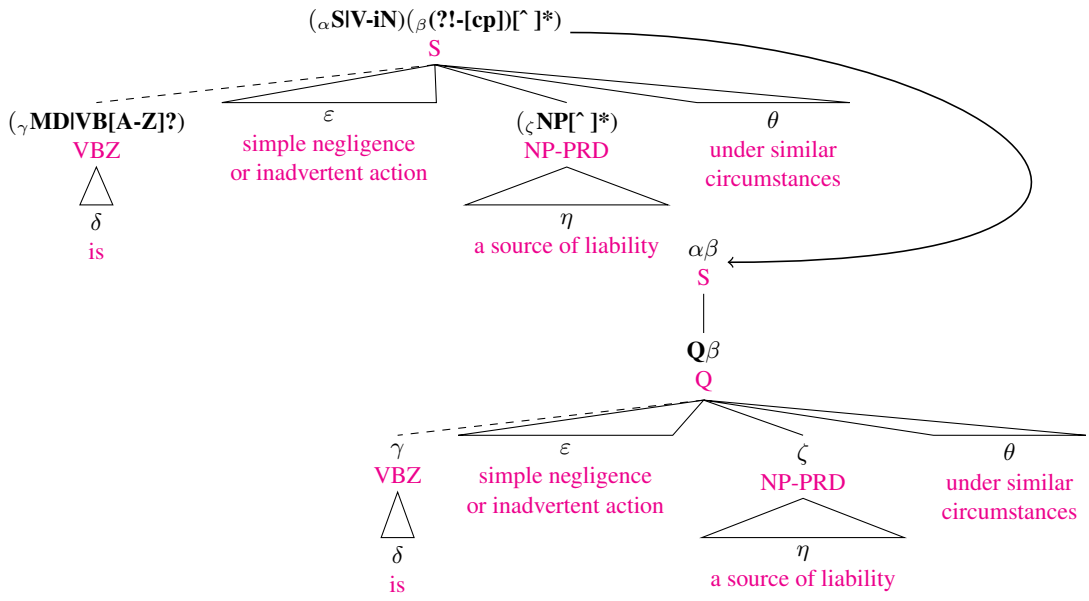


Figure A.16: Polar question: unary expand to **Q**. γ is the left-most pre-terminal under $\alpha\beta$. If γ=**VB[A-Z]*** then δ=**[Dd]oes|[Dd]o|[Dd]id|[Ii]s|[Aa]re|[Ww]as|[Ww]ere** or **[Hh]as|[Hh]ave|[Hh]ad**. This example has α=S, β=$\varnothing$, γ=VBD, and ε=NP-SBJ.

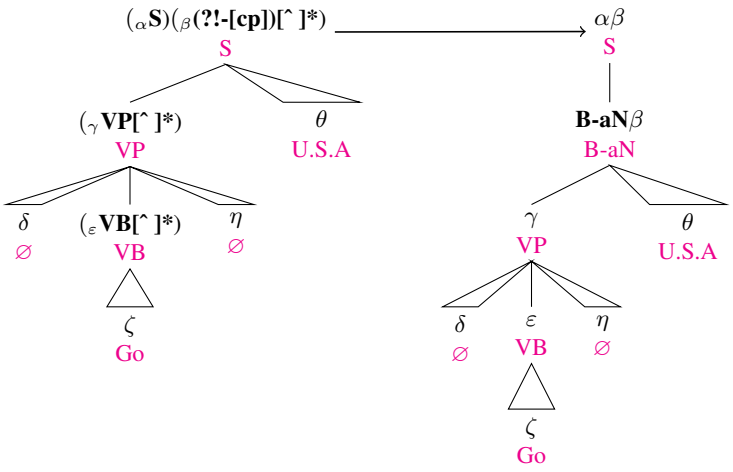


Figure A.17: Imperative sentence: unary expand to **B-aN**. In this rule, top level nodes in δ must not be any of **VB**, **JJ**, **MD**, or **TO**. This means ε is the first **VP**-head child of γ. This type changing rule changes a **B-aN** to an **S**.

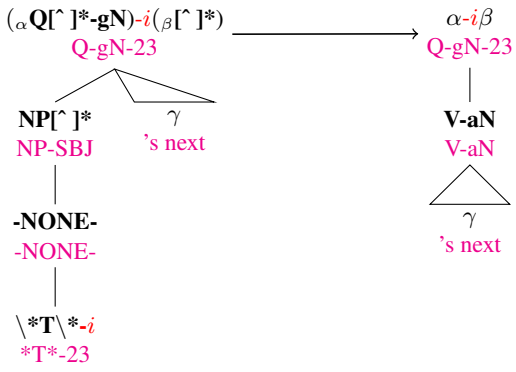


Figure A.18: Polar question: allow subject gap without inversion. This type changing rule changes a **V-aN** to a **Q-gN**. This example has α=Q-gN, i=23, and β=$\varnothing$.

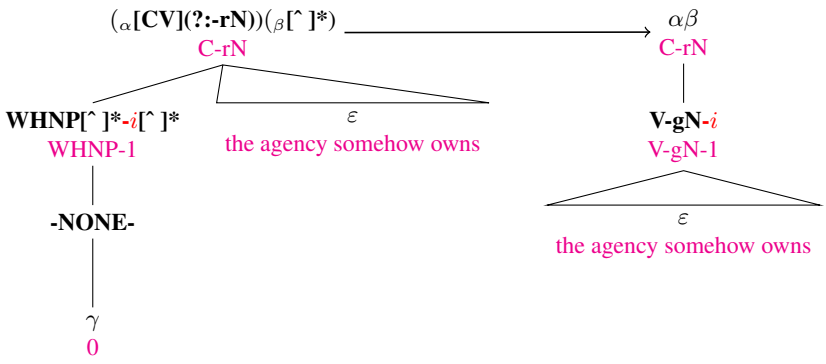


Figure A.19: Implicit-pronoun relative: delete initial empty interrogative phrase. This type changing rule changes a **V-gN** to a **C-rN**. This example has α=C, β=-rN and i=1.

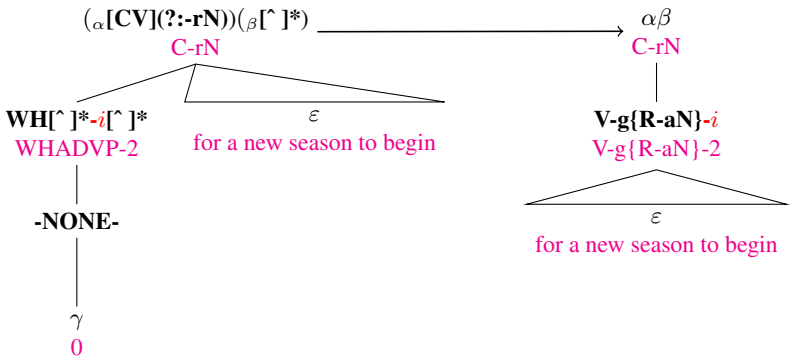


Figure A.20: Implicit-pronoun relative: delete initial empty interrogative phrase as adverbial. This type changing rule changes a **V-g{R-aN}** to a **C-rN**. This example has α=C, β=-rN and i=2.

A.5 Miscellaneous Rules

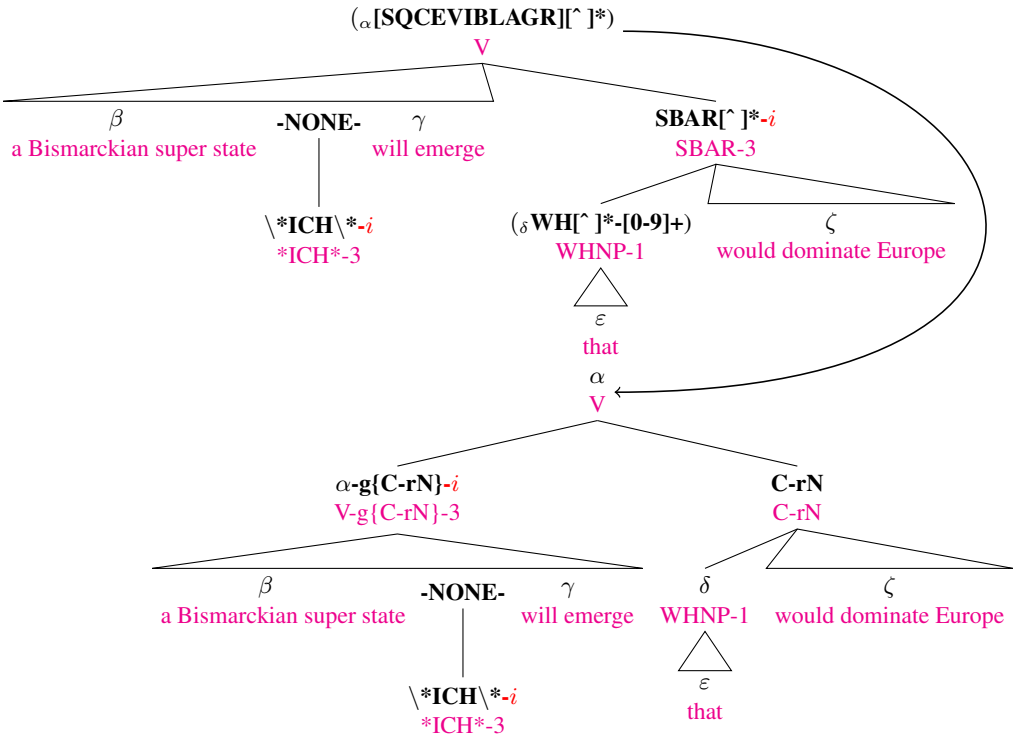


Figure A.21: Branch off final **SBAR** as extraposed modifier **C** relative clause. Subtree ε under δ must not contain **-NONE-** or $what$. This example has i=3.

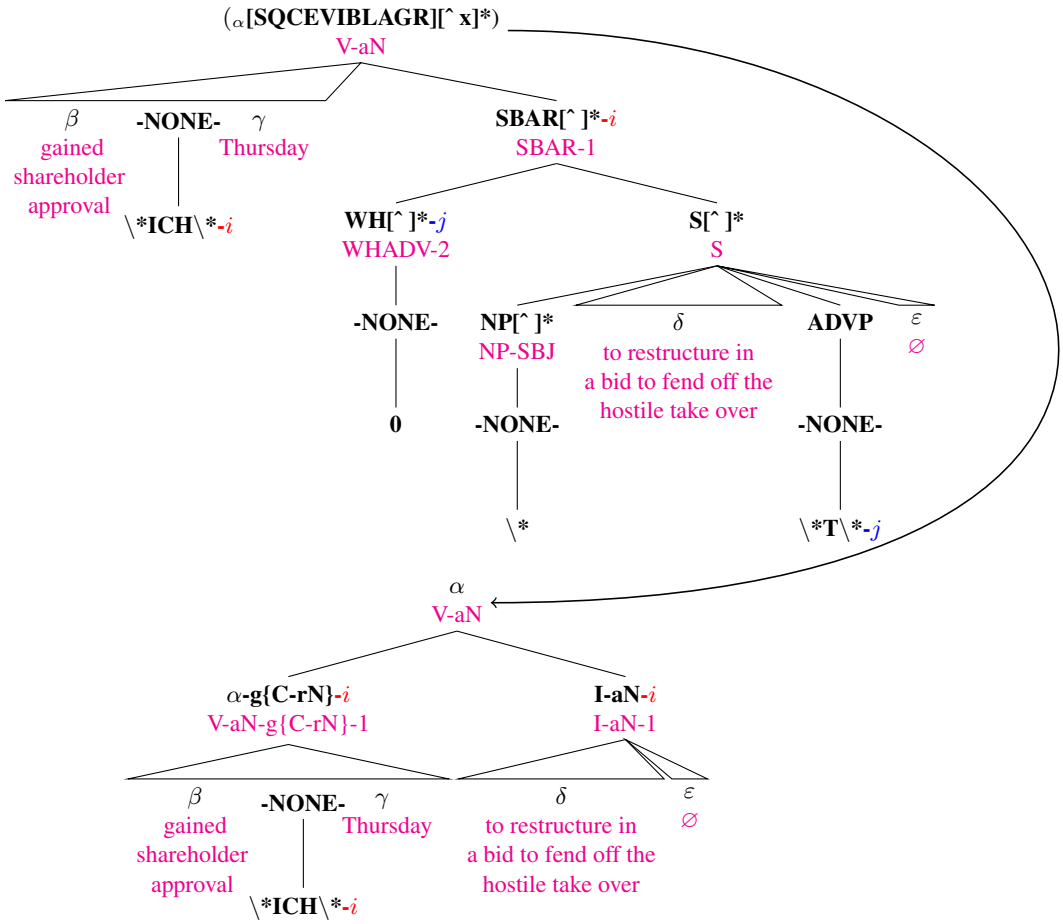


Figure A.22: Branch off final SBAR as extraposed modifier **I-aN**. The part **[^ x]*** in α is to ensure it won't match with intransitive categories which are ending in **-aN-x**. This example has α=V-aN, i=1 and j=2.

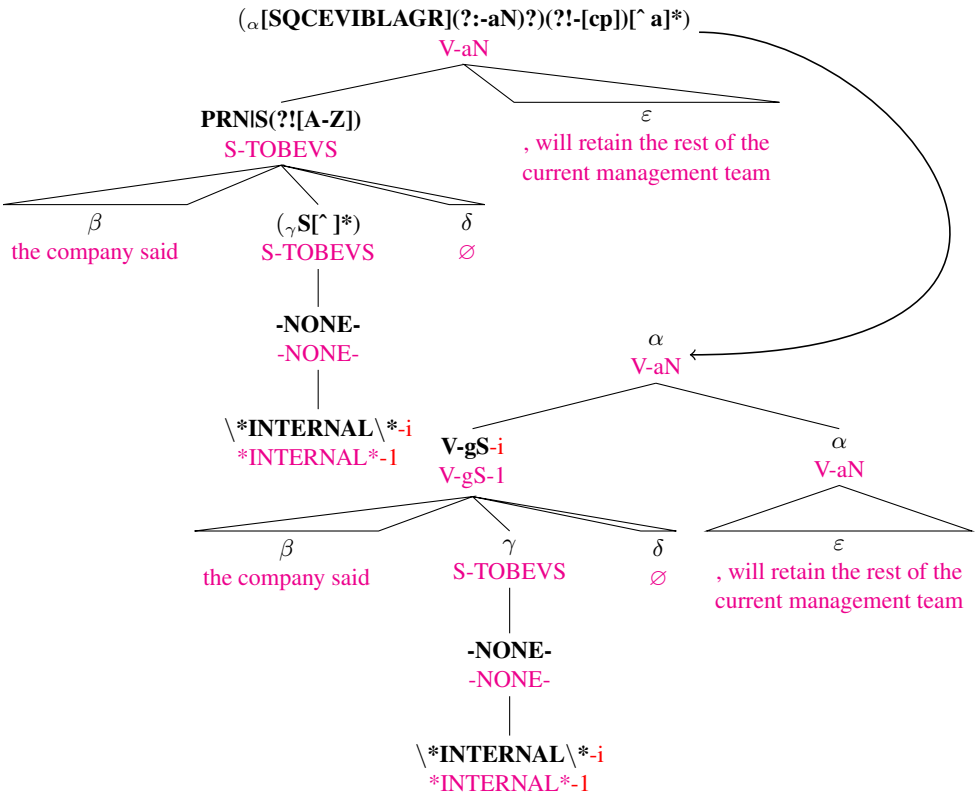


Figure A.23: Branch off initial parenthetical sentence with extraction.

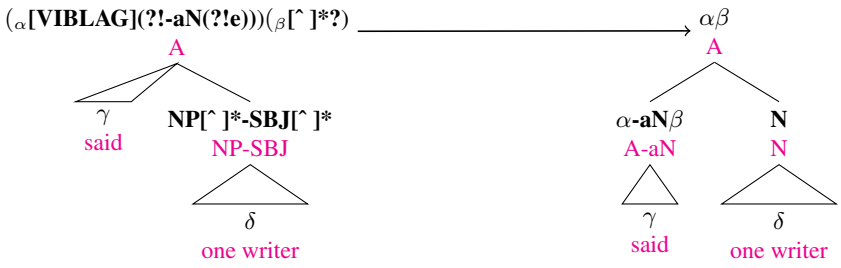


Figure A.24: Inverted declarative sentence: branch off final subject.

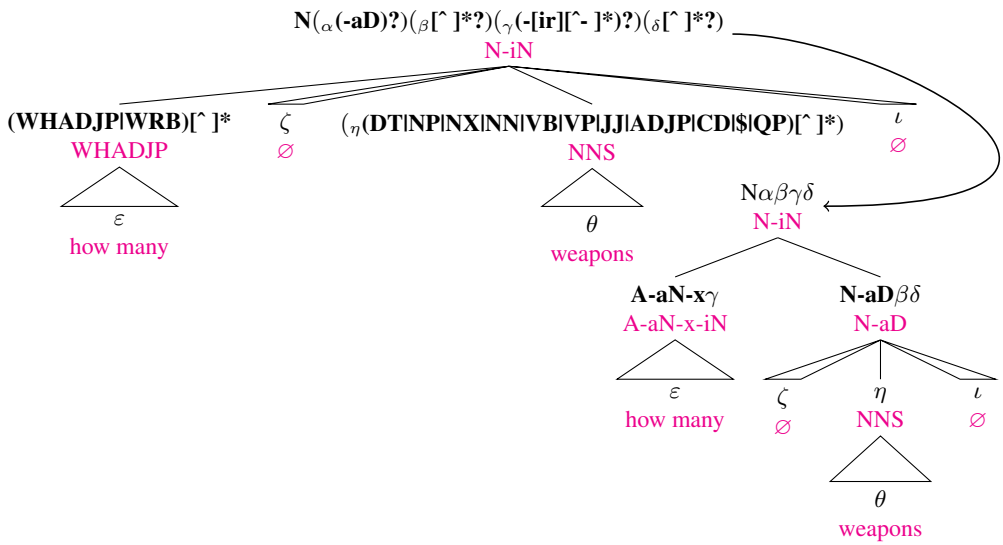


Figure A.25: Branch off initial modifier **A-aN-x**. if η=**QP** then its leftmost child in θ must be of category **$**. This example has α=β=δ=∅ and γ=-iN.

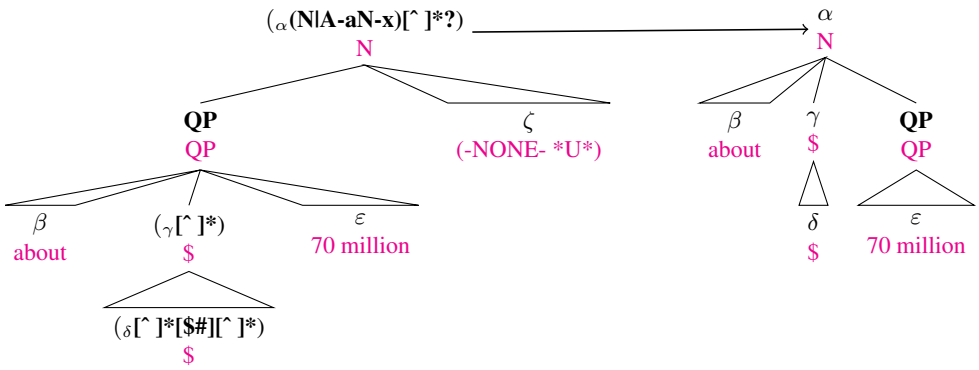


Figure A.26: Rebinarize **QP** containing dollar sign followed by ***U***, and continue. If $\zeta \neq$∅ then ζ=**(-NONE- *U*)**.

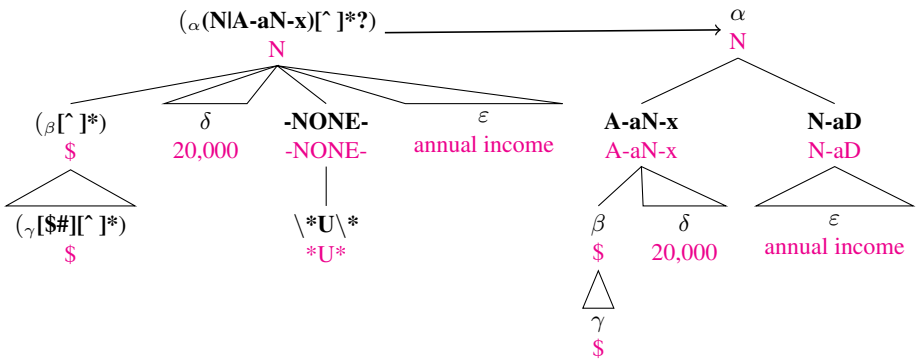


Figure A.27: Branch off currency unit followed by non-final ***U***.