Comparative Analysis of DeepBank and the Penn Treebank

Megan Schneider

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2013

Reading Committee:

Emily M. Bender, Chair

Gina-Anne Levow

Program Authorized to Offer Degree:
Department of Linguistics

University of Washington

Abstract

Comparative Analysis of DeepBank and the Penn Treebank

Megan Schneider

Chair of the Supervisory Committee:
Dr. Emily M. Bender
Linguistics

I examined the differences between the DeepBank and Penn Treebank and the effect of hand-created and grammar-derived annotations on dependency representations. The dependencies comparison involved transforming the DeepBank trees into Penn Treebank format, training the Stanford parser on the resulting output, and testing the trained parser vs known dependencies data; this task yielded a null result. A detailed analysis of the remaining differences between the Penn Treebank and modified DeepBank was done after the transformation process, showing many differences including parse selection, clause and phrase attachment, labeling of modifiers, and the treatment of proper noun phrases like movie titles. This yielded useful information for future work in this area.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

This thesis examines the differences between the DeepBank and the Penn Treebank, as well as looking at whether hand-created annotations or grammar-derived annotations provide more consistent dependency representations. The latter is tested by whether or not the Stanford parser (22) trained on DeepBank-derived (18) phrase structure trees which have been transformed into the Penn Treebank (PTB) format (25) is more or less accurate at parsing deep dependencies than the Stanford parser (22) trained on the Wall Street Journal (WSJ) portions of the Penn Treebank. The accuracy determination is made with the Deep Dependency Evaluation Corpus and the methodologies laid out by Bender et al. (3). Additionally, given the same deep dependency task and multiple versions of the altered DeepBank-derived trees which vary in their degree of similarity to the Penn Treebank format, I also seek to examine which variations produce the best parser. Certain aspects of the dependencies task received a null result, for reasons discussed in Chapters 5 and 6. In addition to the dependencies task, I also scrutinize the accuracy of the intermediate steps and the remaining transformational differences between the altered DeepBank parses and the original Penn Treebank. This is both to provide insight into remaining gaps for future work and to illustrate some of the basic format differences between the PTB and DeepBank.

The Penn Treebank is largely manually annotated and possesses a limited number of node labels in part to make that annotation task feasible. It does not provide information beyond the phrase structure trees and node labels, and therefore some types of deep dependencies may be difficult to recover (3). Development of the Penn Treebank has ceased, though it and its output format are still commonly used in a variety of research. Notably, the Penn Treebank format is one of the accepted input formats for training the Stanford parser.

The DeepBank project has the aim of annotating the same Wall Street Journal sentences

which are present in the Penn Treebank with the English Resource Grammar (ERG; 16) and a robust probabilistic context free grammar (PCFG) derived from ERG parses (18). This project is ongoing. The English Resource Grammar is an HPSG-based grammar which has been under continuous development since 1993 (16). Its native representations use feature-value pairs to express very detailed linguistic information and it uses a set of node labels to abbreviate those large feature structures. This set of node labels is roughly as large as the set of node labels used in the Penn Treebank.

There are basic structural differences between the Penn Treebank trees and the DeepBank trees; as a result a translation layer is required in order to produce trees sufficiently similar to the Penn Treebank for the Stanford parser. In addition to the basic tree structure and node label modifications it is also of interest to determine what, if any, additional information that the exported DeepBank parses possess is useful to the Stanford parser in outputting dependency structures.

The translation layer should prove to be generally useful for anyone who wishes to move from using Penn Treebank trees to the DeepBank parses in particular or English Resource Grammar parses in general. A variety of utilities were also written during the course of creating the translation layer; it is hoped that these utilities will save time in future work involving manipulation of parse trees. Additionally, researchers training parsers may find the general architecture of the translation layer useful for adapting other inputs. The code developed for this thesis is available from `https://bitbucket.org/caelum/thesis-transform`.

The next sections of this thesis delve into background information and previous work in this area. This is followed by descriptions of the general methodology used and implementation details. I then discuss expected mismatches between the original PTB and the modified ERG output. Finally, I detail the deep dependency results, analyze the remaining tree differences, and discuss future paths.

Chapter 2

AN OVERVIEW OF PREVIOUS WORK

This chapter discusses relevant background information and the various tools and representations relevant to the execution of this thesis.

2.1 Deep Dependencies

The Deep Dependencies Evaluation Corpus was developed in 2010 and described in Bender et al. 3. It consists of 100 examples each of ten linguistic phenomena intended to cover a range of deep and subtle linguistic dependency relations which occur with reasonably high frequency and are not usually highlighted by parseval (6) or other evaluation metrics for parsing. The sentences are sourced from the English Wikipedia.

Bender et al. (3) evaluated seven parsers for their ability to recover the dependencies in each sentence, among them the Stanford parser (22) trained on the English factored model. This model uses the preferences to provide both unlexicalized PCFG phrase structures and typed dependencies (grammatical relations) and is trained on sections 02-21 of the Wall Street Journal portion of the Penn Treebank. To survey the ten linguistic phenomena: three of them represent long-distance dependencies, two concern the absence of dependencies, two involve modifier phrases, and three involve verbal arguments outside of ordinary finite clauses. Examples are shown in Figure 2.1. The long-distance dependencies are finite *that*-less relative clauses, tough adjectives, and right-node raising. The absence of dependencies phenomena involve *it* expletives and verb-particle constructions. The first of the modifier phrases involves a noun taking the *-ed* ending and then being used as a predicate or modifier in conjunction with a noun or adjective. The second modifier phrase is the absolutive, which consists of a noun phrase preceding a non-finite predicate. Raising/control constructions, the arguments of verbal gerunds, and the interleaving of arguments and adjuncts make up the final three dependency types. The corpus contains the manually annotated expected

Finite thatless relative clauses
The maximum points a single team can earn is 775.

Right node raising items
With inertial guidance, a computer control system directs and assigns tasks to the vehicles.

Interleaved arguments and adjuncts
The story shows, through flashbacks, the different histories of the characters.

Adj-or-N + N-ed
Light colored glazes also have softening effects when painted over dark or bright images.

Absolutives
His diary clear for the rest of the day, Hacker sits down with Bernard to go through his brief.

Verbal gerunds
Even when a construction is offered, it is usually applied towards proving the axioms of the real numbers, which then support the above proofs.

Raising and Control VPs
In these capacities, he helped to reform education and to establish the first university in Kazakhstan.

Tough adjectives
The long-term effect on the viewing habits of the general public will be difficult to gauge.

Expletive it
This arrow affected how the unit could move, as it cost a movement factor each time the unit was moved.

Verb-particles
However, the sound has changed in recent times to take on a darker and more mysterious tone.

Figure 2.1: Deep Dependency Types & Examples

dependencies and the original script used in the paper for comparison of parser dependency output to the gold standard is available[1]. Neither of these requires modification for the purposes of this thesis.

2.2 Penn Treebank

The Penn Treebank was largely developed between 1989 and 1996; the first phase of the project ended in 1992 (25). It was created via manual annotation with some automated preprocessing and utilizes text from the Air Traffic Information System, the Wall Street Journal (WSJ), the Brown Corpus, Switchboard, and a variety of other sources. The relevant portion for this thesis is the Wall Street Journal texts. A great deal of research uses the Penn Treebank in general and the WSJ portion in particular. No further development is being done on the PTB, however, and the complexity and detail of the annotations is limited in part to allow for better inter-annotator agreement (25). The most recent version of the Penn Treebank uses 36 part of speech tags for words and approximately 12 other part of speech tags for symbols and punctuation. It also has 26 phrase or clause level node labels.

In addition to the node labels, the PTB representations have many other aspects of note. The output tree structure is not constrained to a binary (or any other n-ary) form. Non-ascii symbols such as curved quotes and long hyphens are converted to ascii equivalents. Punctuation and contractions are tagged as separate words, while hyphenated compounds are not split apart. There are some inconsistencies between the input sentences and the resulting output; for example, an extra period is often inserted in the output when *U.S.* is the final word in the sentence (15). Beyond the tokenization, node labels, and tree structure, there is no additional typing or information given in the annotated trees. This thesis requires altering the DeepBank analyses, which are in the output format of the English Resource Grammar, to mimic the Penn Treebank format. This is for both the dependencies comparison task, where the Stanford parser expects to receive information in the PTB format, and the detailed analysis of the remaining differences between the altered DeepBank parses and the Penn Treebank. By transforming the DeepBank trees to comply

[1] `http://www.delph-in.net/ddec/`

with certain PTB guidelines, other differences become highlighted.

2.3 DELPH-IN

This thesis uses several different tools and resources produced by the the Deep Linguistic Processing with HPSG Initiative (DELPH-IN)[2], which is an international collaborative effort of computational linguistics research groups. The partners make use of two formal linguistics models: Minimal Recursion Semantics (MRS) (12) and Head-Driven Phrase Structure Grammar (HPSG) (33). MRS is a framework for semantic representation consisting of flat structures which allow underspecification. It can be integrated with HPSG-based grammars and is integrated with the English Resource Grammar which is discussed further below. HPSG is a framework for natural language syntax and semantics; it benefits from an active community of theoretical linguists and has analysed a variety of phenomena in multiple languages (34). A basic understanding of MRS is required for parsing some of the output formats described in the next chapter. DELPH-IN has produced an extensive set of tools, grammars, treebanks, architectures, and frameworks, several of which are referenced throughout this thesis.

LOGON is an infrastructure which was originally developed to work on quality Norwegian-English translation by the Norwegian LOGON and HandOn research projects and has since expanded to include other languages; many of its core aspects are discussed in further detail in Oepen et al. 32. The infrastructure ties together a set of tools designed for experimentation with machine translation, parsing, and generation. It is intended to provide some degree of uniformity, interoperability, and ease of installation. LOGON is part of the base infrastructure required for the code created through this thesis.

The LKB is primarily documented in (10); it is a development environment for grammar and lexicons. It is not strictly limited to HPSG but does make use of typed feature structures, and is intended for interactive development over efficiency. PET is aimed at being much more efficient for batch-processing and takes the same inputs as the LKB and provides the same outputs (7). The LKB has proven particularly useful during node label

[2] `http://www.delph-in.net`

rule creation and debugging.

[incr tsdb()] is a package for system and grammar development that focuses on profiling the performance and competence of those systems and grammars (30). It enables comparison across different versions of a system with relative ease. The snapshots for each version are referred to as profiles and they are stored in a database and can be exported; these profiles can then be examined with varying degrees of granularity to provide insight into the performance of the system across multiple versions.

Two further DELPH-IN resources that are at the core of this project are described in the next two sections: the English Resource Grammar and the DeepBank.

2.4 The English Resource Grammar

The English Resource Grammar (ERG; 16; 17) has been under continuous development since 1993. The code and documentation for the ERG are available through DELPH-IN, both separately and as part of the LOGON distribution. The ERG is HPSG-based and was developed against a broad base of sources including Wikipedia, appointment scheduling and travel dialogues, prose fiction, and electronic commerce emails (2; 11; 38). It uses Minimal Recursion Semantics (MRS), described in Copestake et al. 12, for its semantic representations and the ERG release 1212 defines more than 200 features, almost 9000 types, 29 node labels, and more than 300 lexical and grammatical rules. The ERG includes more than 38,000 lexical items and unknown-word handling facilities.

All of the rules, types, features, and labels of the ERG are defined in type description language (TDL) (9) files and designed to be easily maintainable; new features and rules are designed to be developed and maintained over time and modifying node labels is very simple. The analyses the ERG produces for strings are trees with feature structures at the nodes. These feature structures are represented in attribute value matrices (AVMs). The basic tree structure consists of binary trees and each node contains a great deal of information as to its features, which may be identified with other nodes' values or provided directly by a rule or lexical entry. Processing software which uses the ERG, such as the LKB and PET, provides output formats which abbreviate large feature structures with simple node labels. The label application is done through AVM-matching against the ordered entries in

the TDL labels file.

The tokenization assumed by the ERG and effectuated by its preprocessor is not directly in line with the Penn Treebank: punctuation is left attached to words and there is broader character support than simple ASCII. Contractions are attached to words while possessives and hyphenated compounds are separated. The Regular Expression Pre-Processor (REPP) described by Dridan and Oepen (15) is able to reproduce PTB tokenization with reasonably high accuracy, in order to provide a standalone tokenizer to NLP tools which require PTB-compliant tokenization. The ERG uses a REPP tokenizer in conjunction with token mapping rules to prepare input for parsing; REPP implementations are also available in several of the systems which work with the ERG and are included in LOGON (15). I make use of REPP tokenization to alter the ERG tokenization into more PTB-compliant tokenization.

The Redwoods Treebank, which is described in Oepen et al. (31), is a grammar-based treebank and makes use of the ERG and an HPSG framework. The grammar provides the parse forest and the treebanker uses minimal discriminants (8) to select the tree that represents the intended meaning of the sentence in context or reject all of the trees if no such tree is available. The treebanking tool stores the discriminants in addition to the selected parse; after updates or modifications the grammar can use these discriminants to reproduce the treebank while requiring minimal additional disambiguation. There is a script in LOGON for Redwoods which allows for exporting parses in various text formats including MRS, derivation trees, phrase structure trees with abbreviated node labels, AVMs, input tokenization, and elementary dependency structures, which are a reduced MRS form. I make use of this script to produce relabeled DeepBank parses. The formats available include plain text and XML. From the export formats it is possible to determine the rules and lexical entries used at each level of the tree, the ERG representations, the assigned node labels, and a variety of other details about how the parse tree is constructed. Additionally, the derivation tree can be used by the ERG to reconstruct the full analysis.

2.5 DeepBank

The DeepBank project began development in 2008 and was first partially released in late 2012 and is still undergoing development; the first public release was made in January 2013. As described in Flickinger et al. 18, it annotates the 1989 Wall Street Journal text with the English Resource Grammar and an approximating PCFG based on ERG parses; this is the same set of Wall Street Journal sentences previously annotated by the Penn Treebank. Instead of being derived from the Penn Treebank, the DeepBank annotations are determined independently. It adopts the dynamic treebanking methodology laid out by Redwoods, using the ERG and PCFG to produce candidate parses and human annotators to disambiguate parses. At this time the DeepBank sentences have undergone at least two rounds of human annotation.

DeepBank provides as one of its annotation formats the raw [incr tsdb()] profiles. These profiles allow for directly examining the MRS structures and ERG derivations and provide more detailed reconstructions when used with the appropriate version of the ERG. Roughly 15% of the annotations are generated by the approximating PCFG where the ERG failed to parse the WSJ sentence.[3] These cases largely consist of incomplete linguistic coverage in the grammar, computing resource limitations imposed on sentence analysis, or more rarely ill-formed text as the parser input. The DeepBank ERG-derived analyses are the input for the work contained in this thesis; no PCFG-derived analyses were used.

While the DeepBank parses are selected manually, as stated above the initial parsing is done via the ERG, which is rule-based. It is hoped that this initial step plus the addition of [incr tsdb()] and its utilities for discriminant selection allow for a more reliable and consistent process than was available at the time of the creation of the PTB, which is manually annotated. It is expected that the initial node label assignment via the ERG will be more consistent than manual node label assignment. In part this is because manual assignment becomes less accurate when applied to more complex structures; grammar-derived structures do not have this limitation (31).

---

[3]From `http://svn.delph-in.net/erg/tags/1212/etc/redwoods.xls`, accessed 31-July-2013.

2.6 Summary

The discussion of the tools, grammars, and representations above provides the necessary background information for the more detailed examination of the various aspects of this thesis in the next chapters. The ERG, LOGON, and DeepBank provide the main base for the transformation of DeepBank parses with ERG node labels and tree structure into DeepBank parses with PTB node labels and tree structure. The PTB provides the comparison point for the extent that transformation, and the DDEC provides the comparison point for the effectiveness of deep dependency parsing on both the original PTB and the transformed DeepBank parses.

Chapter 3

METHODOLOGY

This chapter is intended to provide a high-level overview of the methodology used in this thesis. A more detailed description is in the next chapter. The initial input for this work is the DeepBank and the final output makes use of the Deep Dependency Evaluation Corpus to examine the Stanford parser output. The intermediate steps involve changing the node labels the DeepBank analyses use from ERG node labels to PTB node labels (Section 3.1), exporting the altered DeepBank parses to text files which contain the tree structure and rules and other ERG information (Section 3.2), parsing those text files into modifiable trees (Section 3.3), and modifying those trees to resemble the PTB tree structure to varying degrees (Section 3.3). These trees are the training input to the Stanford parser. The trained parser is then tested on the DDEC sentences and the result is analyzed with the DDEC toolset versus the gold standard (Section 3.4). The details of each step are explained below. Figure 3.1 and Figure 3.2 show the original PTB format and the starting point of the raw DeepBank result for the same sentence.

```
( (S
    (NP-SBJ (NNS Investors) )
    (VP (VBD welcomed)
      (NP (DT the) (NN move) ))
    (. .) ))
```

Figure 3.1: PTB native format (PTB ID 20376004)

20376004@0@-1@-1@-1@-1@-1@-1@-1@(root_strict (441 sb-hd_mc_c 5.70235 0 4 (435 hdn_bnp_c
1.74967 0 1 (434 n_pl_olr 1.77268 0 1 (73 investor_n1 1.1102 0 1 ("investors" 63 "token
[+CARG "Investors" +CLASS alphabetic [+CASE capitalized+lower +INITIAL +] +FORM
"investors" +FROM "0" +ID *diff-list* [LAST #1=*top* LIST *cons* [FIRST "1" REST #1]
] +PRED predsort +TICK bool +TNT null_tnt [+MAIN tnt_main [+PRB "0.47845729999999997"
+TAG "NNP"] +PRBS *null* +TAGS *null*] +TO "9" +TRAIT token_trait [+HD token_head +IT
italics +LB bracket_null +RB bracket_null +UW -]]")))) (440 hd-cmp_u_c 2.71741 1 4 (436
v_pst_olr 0.216849 1 2 (80 welcome_v1 0 1 2 ("welcomed" 53 "token [+CARG #1="welcomed"
+CLASS alphabetic [+CASE non_capitalized+lower +INITIAL -] +FORM #1 +FROM "10" +ID
diff-list [LAST #2=*top* LIST *cons* [FIRST "2" REST #2]] +PRED predsort +TICK bool
+TNT null_tnt [+MAIN tnt_main [+PRB "0.95325110000000002" +TAG "VBD"] +PRBS *null* +TAGS
null] +TO "18" +TRAIT token_trait [+HD token_head +IT italics +LB bracket_null +RB
bracket_null +UW -]]"))) (439 sp-hd_n_c 1.9369 2 4 (133 the_1 0.46344 2 3 ("the" 51 "token
[+CARG #1="the" +CLASS alphabetic [+CASE non_capitalized+lower +INITIAL -] +FORM #1 +FROM
"19" +ID *diff-list* [LAST #2=*top* LIST *cons* [FIRST "3" REST #2]] +PRED predsort
+TICK bool +TNT null_tnt [+MAIN tnt_main [+PRB "1" +TAG "DT"] +PRBS *null* +TAGS *null*
] +TO "22" +TRAIT token_trait [+HD token_head +IT italics +LB bracket_null +RB bracket_null
+UW -]]")) (438 w_period_plr 1.25344 3 4 (437 n_sg_ilr 0.989387 3 4 (146 move_n1 0.253198 3
4 ("move." 55 "token [+CARG "move" +CLASS alphabetic [+CASE non_capitalized+lower +INITIAL
-] +FORM "move." +FROM "23" +ID *diff-list* [LAST #1=*list* LIST *cons* [FIRST "4"
REST *cons* [FIRST "5" REST #1]]] +PRED predsort +TICK bool +TNT null_tnt [+MAIN
tnt_main [+PRB "0.98965190000000003" +TAG "NN"] +PRBS *null* +TAGS *null*] +TO "28"
+TRAIT token_trait [+HD token_head +IT italics +LB bracket_null +RB bracket_null +UW -]
]"))))))))@@("S" ("NP" ("N" ("N" ("investors")))) ("VP" ("V" ("V" ("welcomed"))) ("NP"
("DET" ("the")) ("N" ("N" ("N" ("move."))))))))@[LTOP: h1 INDEX: e3 [e SF: PROP TENSE:
PAST MOOD: INDICATIVE PROG: - PERF: -] RELS: < [udef_q_rel<0:9> LBL: h4 ARG0: x6 [x
PERS: 3 NUM: PL IND: +] RSTR: h5 BODY: h7] ["_investor_n_1_rel"<0:9> LBL: h8 ARG0: x6]
["_welcome_v_1_rel"<10:18> LBL: h2 ARG0: e3 ARG1: x6 ARG2: x9 [x PERS: 3 NUM: SG IND:
+]] [_the_q_rel<19:22> LBL: h10 ARG0: x9 RSTR: h12 BODY: h11] ["_move_n_1_rel"<23:28>
LBL: h13 ARG0: x9] > HCONS: < h1 qeq h2 h5 qeq h8 h12 qeq h13 >]@((:ASCORE . 5.702))

Figure 3.2: DeepBank native format (PTB ID 20376004)

3.1 Node Re-labeling

The node labels in the PTB and ERG are different; in order to make the DeepBank-selected parses look more like the PTB, they must be reassigned the labels of the PTB. The DeepBank contains the [incr tsdb()] profiles, which allow for reconstructing the selected parses with a slightly altered ERG. The node labels are defined in an ERG file called parse-nodes.tdl, which is applied to a completed parse. Assignment of labels is done by matching against rules which specify orthography and feature structures; the first rule which matches will be the assigned node label. An example of one such rule I created for this thesis is in Figure 3.3.

The flexible structure of the ERG allows for simple modification of this file and those

```
verb-base-form := label &
      [ SYNSEM [ LOCAL.CAT.HEAD verb & [ VFORM bse ],
                 LEX + ],
        LNAME "VB" ].
```

Figure 3.3: Example Node Label Rule

that define the feature structures and rules; the [incr tsdb()] profiles can make use of these modifications when used with LKB (10), PET (7), or the redwoods script for exporting parses. The difficulty then is in mapping the ERG feature structures to the PTB node labels; the underlying structures of the English Resource Grammar and the Penn Treebank are different. A complete one-to-one mapping for the labels is not possible, although this can be compensated for with later tree processing. The node label mappings are further discussed with more examples in the next chapter. In order to allow for PTB function tags like -SBJ, I added an additional feature to the grammar to track the rule applied to a node's parent node; this is further described in the next chapter. The node labels for both the PTB and ERG are listed in Table 3.1.

	Node Labels					
Penn Treebank	S	FW	NNPS	RBS	VP	,
	SBAR	IN	NP	RP	WDT	:
	SBARQ	INTJ	NX	RRC	WHADJP	(
	SINV	JJ	PDT	SYM	WHAVP	)
	SQ	JJR	POS	TO	WHNP	''
	ADJP	JJS	PP	UCP	WHPP	‘
	ADVP	LS	PRN	UH	WP	“
	CC	LST	PRP	VB	WP$	’
	CD	MD	PRP$	VBD	WRB	”
	CONJP	NAC	PRT	VBG	X	
	DT	NN	QP	VBN	#	
	EX	NNS	RB	VBP	$	
	FRAG	NNP	RBR	VBZ	.	
English Resource Grammar	S	COMP	N	NX	V	
	SC	CONJ	NP	P	VP	
	ADJ	DET	NP-CJ	PNCT	VP-C	
	ADV	FP	NP-R	PP	VP-CR	
	ADV-C	FRG	NP-X	PRDP	XP	
	AP	LADV	NP-WH	STEM		

Table 3.1: PTB and ERG Node Labels

3.2 Exporting of ERG Parses

The DeepBank selected parses are exported via the redwoods script which is part of the LOGON distribution. The version of LOGON used here is obtained from the subversion repository at `http://svn.emmtee.net/trunk`; the specific revision is 13301. The redwoods script has a variety of output formats, including Minimal Recursion Semantics, derivation trees, phrase structure trees with node labels, AVMs, and dependencies. All of these outputs are to a text file with basic header information and then the formatted output in text or xml. If multiple output formats are requested they are added to a single file, separated by empty lines. Many of the output formats are represented as trees, whereas others like MRS are lists of structures. The redwoods script uses the derivation tree and English Resource Grammar to reconstruct the parse; any changes made to the grammar since the parse selection are reflected in the output.[1] I use this script to take the DeepBank as distributed and export its parses with altered node labels.

3.3 Tree Transformation

For the next step of the process, I parse the exported text representations into editable trees and then transform them to conform to the PTB tree format. One part of this process is correcting for node labels that cannot be accounted for with editing the ERG parse-nodes.tdl file. These labels include words such as *to*, which is always tagged as TO in the Penn Treebank (35), and labels associated with symbols.

Another part involves retokenizing the sentence to match PTB tokenization. The ERG allows for multi-word leaf nodes like *New Jersey* where the PTB does not. The Penn Treebank also separates punctuation from the words in the sentence, and the ERG leaves the punctuation attached. Hyphenation is also handled differently; hyphenated compounds are combined in the PTB but generally separated by the ERG. There are many other nuances, most of which have been covered in Dridan and Oepen 15. The Regular Expression Pre-Processor (REPP) has been part of PET in the LOGON distribution since August 2011

[1] If the changes result in a sufficiently different structure, the system may be unable to reconstruct an analysis according to the stored derivation tree. The changes made in this work (to the node labels, and the addition of the MRNAME feature) do not lead to any failures of reconstruction.

and can be used to output tokenization largely conforming to Penn Treebank standards when given a sentence as input. This output still possesses some characters not allowed by the PTB, such as curved quotes, which must also be transformed appropriately. REPP tokenization is not given as part of DeepBank; I generate it separately to determine the new tokenization that the DeepBank parse should be altered to.

There are many additional transformations required for the basic tree structure; all trees from the ERG are binary and the PTB does not enforce this behavior. Additionally, the PTB uses flattened noun phrases and differs from the ERG in the level of attachment used for various phrase structures. The ERG-generated trees have rules which do not correspond to PTB constituents; this results in many extra nodes that are not present in the matching PTB tree and these nodes must be collapsed. The PTB additionally requires that non-punctuation/symbol labeled nodes have an additional node with a phrasal node label above the word-level node label when it attaches to a clausal node or to a phrasal node under certain circumstances. For example, while the ERG may allow an adverb to attach directly to the uppermost S-node, the PTB requires that the adverb be included in an adverb phrase before attaching to the S. Note that in many cases similar non-branching nodes exist, but they attach at different levels or require additional parent nodes in either the PTB or ERG. In most of the cases where phrasal nodes are required the ERG and PTB conventions match; for example, in prepositional phrases the phrase after the preposition is contained under a node with a phrasal label in both the ERG and PTB. Each transformation is written to allow sequential application in any order. It should be noted that even with all of the transformations applied, the final altered DeepBank trees are still not expected to fully agree with those of the PTB. The reasons for this are further discussed in Chapter 5.

3.4 Deep Dependencies

The Deep Dependencies Evaluation Corpus (DDEC), described by Bender et al. (3), is used for the evaluation of the effect of the above transformations on the Stanford parser's dependency output. The transformed trees are outputted to text files and the Stanford parser is trained on this output. The trained parser is then run on the DDEC sentences in order to output the selected trees and the dependency output. This output is then examined

with the DDEC evaluation script against the gold standard also included with the DDEC. I expect these results to be different from the results obtained with the same process and the original Penn Treebank due not only to differences in parse selection and the effects of the transformation rules but because the DeepBank parses are generated with the ERG, which is able to parse and represent all of the dependency information in the DDEC sentences. The goal here is to see if any part of the ERG representations can be used by the Stanford parser to improve its dependency parsing, while still remaining close enough to the Penn Treebank format to be acceptable input for the parser.

3.5 Summary

This chapter has laid out the basic steps required to get from the original DeepBank parses to testing the trained Stanford parser against the DDEC. The ERG is altered to have an additional feature for the mother's rule name and the labels file is changed to have PTB node labels. The DeepBank parses are exported via the redwoods script to contain the new node labels, and these parse trees are then modified to more closely resemble the PTB structure. The trees are then used to train the Stanford parser, which is then tested against the DDEC sentences. The next chapter describes each step in more detail.

Chapter 4

SPECIFIC IMPLEMENTATION

This chapter describes in further detail the methodology laid out in Chapter 3. First I more closely examine the modifications made to the ERG: the labels file and how it is altered to map to PTB labels, and the additional feature added to the AVM structure (Section 4.1). I then go on to describe how the trees are exported with the new labels (Section 4.2) and parsed into a transformable format (Section 4.3). Each transformation rule is then described in detail with specific examples in Section 4.4. Last, I explain how the initial comparisons of the modified DeepBank trees and the PTB were performed (Section 4.5), as well as the dependency parsing task (Section 4.6).

4.1 ERG Modifications

For this work, I modified the ERG in two ways: the parse-nodes.tdl file is replaced with a file containing the PTB node labels and how they map to ERG structures, and an additional feature is added to the basic AVM structure to populate the rule applied to a given node's parent on the node itself. Each of these are described in turn below.

Some PTB node labels map to ERG node labels with only minimal changes to the original ERG labels file entry; the PTB and ERG contain the same basic lexical categories but differ in how broadly or narrowly they separate those categories into labels. The ERG node labels, for instance, do not differentiate between any type of adjective and the PTB node labels divide adjectives into three categories: basic, comparative, and superlative. Similar behavior occurs with adverbs. These and a few other cases were very straightforward for translating from ERG to PTB, due to the relevant feature structures already being encoded. Examples of the parse-nodes.tdl entries for the original ERG and modified ERG are in Figure 4.1.

Verb node labels in the PTB are differentiated by tense and person. Verb node labels in the ERG are not differentiated along either of those criteria; at the word level all of the

Original ERG	PTB-like ERG

numadj := label &
  [SYNSEM.LOCAL.CAT.HEAD intadj,
  INFLECTD +,
  LNAME "ADJ"].
	adjective-comparative := label &
	  [SYNSEM.LOCAL.CAT.HEAD compar_adj,
	  INFLECTD +,
	  LNAME "JJR"].

supadj := label &
  [SYNSEM.LOCAL.CAT.HEAD superl_adj,
  INFLECTD +,
  LNAME "ADJ"].
	adjective-superlative := label &
	  [SYNSEM.LOCAL.CAT.HEAD superl_adj,
	  INFLECTD +,
	  LNAME "JJS"].

compadj := label &
  [SYNSEM.LOCAL.CAT.HEAD compar_adj,
  INFLECTD +,
  LNAME "ADJ"].
	numadj := label &
	  [SYNSEM.LOCAL.CAT.HEAD intadj,
	  INFLECTD +,
	  LNAME "JJ"].

Figure 4.1: Adjective Node Label Examples in ERG and PTB-like ERG

labels are left to a single verbal category. Noun node labels are similarly treated. Examples of the parse-nodes.tdl entries for the original ERG and modified ERG are in Figure 4.2.

The differentiation of node labels in the ERG comes at the AVM structure level and in the rules applied to form the parse tree. The majority of the mappings from ERG structures to PTB node labels are defined by the features set by specific rules; the AVM contains tense, number, and person information which is used to differentiate the VBP and VBZ labels, which respectively correspond to non-3rd person singular present and 3rd person singular present tense verbs. Other labels are more difficult to map; TO in the PTB is applied to the word *to*, regardless of how it is used in the sentence. Due to restrictions matching to both fully specified and underspecified attributes on the AVM and the inability to perform strict string matching in the labels file, it is difficult to formulate a rule or set of rules which correctly applies TO to all *to* occurrences. Another label that proves difficult is VBG; since orthographical regular expressions are also not an option in the labels file, *-ing* verbs cannot be differentiated by them and end up falling into other verbal categories. The application of these node labels must therefore be enforced during the tree transformation steps. Labels relating to punctuation must also be applied during that stage of the process; since the ERG does not separate the majority of punctuation from the attached words it is not possible to produce most of the separate punctuation node labels before exporting the trees.

In order to allow the addition of function tags like -SBJ, the rule applied to the parent

Original ERG:

```
coord-v := label &
  [ SYNSEM.LOCAL [ CAT [ HEAD verb,
                         VAL.COMPS *cons* ],
                   CONT.HOOK.INDEX conj_event ],
    LNAME "V" ].
v := label &
  [ SYNSEM.LOCAL.CAT [ HEAD verb,
           VAL.COMPS *cons* ],
    INFLECTD +,
    LNAME "V" ].
tagaux := label &
  [ SYNSEM.LOCAL.CAT [ HEAD tagaux,
           VAL.COMPS *cons* ],
    INFLECTD +,
    LNAME "V" ].
```

PTB-like ERG:

```
verb-past-tense := label &
  [ SYNSEM.LOCAL.CAT.HEAD verb & [ TAM.TENSE past ],
    LNAME "VBD" ].
verb-3rd-singular-present := label &
  [ SYNSEM [ LOCAL.CAT [ HEAD verb & [ TAM.TENSE present ],
                         VAL [ SUBJ [ REST *null*,
                                      FIRST [ --SIND [ PNG.PN 3s ] ] ] ] ] ],
             LEX + ],
    LNAME "VBZ" ].
verb-non-3rd-singular-present := label &
  [ SYNSEM [ LOCAL.CAT [ HEAD verb & [ TAM.TENSE present ],
                         VAL [ SUBJ [ REST *null*,
                                      FIRST [ --SIND [ PNG.PN -3s ] ] ] ] ] ],
             LEX + ],
    LNAME "VBP" ].
```

Figure 4.2: Verb Node Label Examples for ERG and PTB-like ERG

(mother) of a given node must be known as part of that node's AVM. NPs which should have the -SBJ tag are identified as such because they are the daughter of a subject-head rule; the mother's rule name must therefore be accessible from the daughter in the labels file. In the original ERG, there is no indication of the rule licensing the mother present in the daughters' AVMs. I added a feature called MRNAME to the type *sign* in the ERG's fundamentals.tdl, which file lays out the basic structures used for all AVMs produced by the ERG, in order to indicate the mother's licensing rule. There is an existing feature called RNAME which lists the rule responsible for licensing the node itself; MRNAME

is added alongside this feature and is populated by altering the basic phrase types like basic_unary_phrase and basic_binary_phrase to apply the value of RNAME to MRNAME on each of the ARGS (daughters) listed in the structure.

After the above modifications, the ERG will produce parses with Penn Treebank node labels and structures including the mother's rule name, which allow for Penn Treebank function tags on the node labels. The node labels at this point are only expected to be partially correct. Punctuation labels will be completely missing from the tree because punctuation is not separated from the words in the ERG. Node labels which are difficult to fully apply for previously discussed reasons will also not be fully accurate at this stage.

4.2 Exporting Tree Information

The parses are exported via the redwoods script in the LOGON distribution. This script takes the location of the altered ERG, the name of the DeepBank selected parse directory, a list of what outputs are requested, and the name of an output directory.[1] For our purposes we want the derivation and tree output formats. The derivation output format includes the rule names applied at each node of the tree as well as the word tokenization and other detailed information at the word level. The tree output format includes the node labels and the word tokenization. The tree structures in each format contain the same number of nodes, except where DeepBank was unable to parse the sentence with the ERG and was forced to fall back to the PCFG. In such cases one or both output formats may consist of empty trees.

4.3 Tree Parsing

Where the Redwoods export contains both a derivation tree and a phrase structure tree with PTB-like node labels, the trees can be combined into a single tree containing the node labels and rule names as part of the "tag" portion of each node. Examples of the derivation tree, phrase structure tree, and combined tree are in Figure 4.4, Figure 4.5, and Figure 4.6 respectively. The rule names are included to allow the transformation rules

[1] Example usage: `./redwoods --binary --terg --target /Thesis/exported-trees/ --export derivation,tree wsj01c.1`

```
(ROOT_STRICT
  (441 SB-HD_MC_C 5.70235 0 4
   (435 HDN_BNP_C 1.74967 0 1
    (434 N_PL_OLR 1.77268 0 1
     (73 investor_n1/n_-_c_le 1.1102 0 1
      ("investors" 63
       "token [ +CARG \"Investors\" +CLASS alphabetic [ +CASE capitalized+lower +INITIAL
+ ] +FORM \"investors\" +FROM \"0\" +ID *diff-list* [ LAST #1=*top* LIST *cons* [
FIRST \"1\" REST #1 ] ] +PRED predsort +TICK bool +TNT null_tnt [ +MAIN tnt_main [ +PRB
\"0.47845729999999997\" +TAG \"NNP\" ] +PRBS *null* +TAGS *null* ] +TO \"9\" +TRAIT
token_trait [ +HD token_head +IT italics +LB bracket_null +RB bracket_null +UW - ] ]"))))
  (440 HD-CMP_U_C 2.71741 1 4
   (436 V_PST_OLR 0.216849 1 2
    (80 welcome_v1/v_np_le 0 1 2
     ("welcomed" 53
      "token [ +CARG #1=\"welcomed\" +CLASS alphabetic [ +CASE non_capitalized+lower
+INITIAL - ] +FORM #1 +FROM \"10\" +ID *diff-list* [ LAST #2=*top* LIST *cons* [ FIRST
\"2\" REST #2 ] ] +PRED predsort +TICK bool +TNT null_tnt [ +MAIN tnt_main [ +PRB
\"0.95325110000000002\" +TAG \"VBD\" ] +PRBS *null* +TAGS *null* ] +TO \"18\" +TRAIT
token_trait [ +HD token_head +IT italics +LB bracket_null +RB bracket_null +UW - ] ]")))
  (439 SP-HD_N_C 1.9369 2 4
   (133 the_1/d_-_the_le 0.46344 2 3
    ("the" 51
     "token [ +CARG #1=\"the\" +CLASS alphabetic [ +CASE non_capitalized+lower +INITIAL
- ] +FORM #1 +FROM \"19\" +ID *diff-list* [ LAST #2=*top* LIST *cons* [ FIRST \"3\" REST
#2 ] ] +PRED predsort +TICK bool +TNT null_tnt [ +MAIN tnt_main [ +PRB \"1\" +TAG \"DT\" ]
+PRBS *null* +TAGS *null* ] +TO \"22\" +TRAIT token_trait [ +HD token_head +IT italics +LB
bracket_null +RB bracket_null +UW - ] ]"))
  (438 W_PERIOD_PLR 1.25344 3 4
   (437 N_SG_ILR 0.989387 3 4
    (146 move_n1/n_-_c_le 0.253198 3 4
     ("move." 55
      "token [ +CARG \"move\" +CLASS alphabetic [ +CASE non_capitalized+lower +INITIAL - ]
+FORM \"move.\" +FROM \"23\" +ID *diff-list* [ LAST #1=*list* LIST *cons* [ FIRST \"4\" REST
*cons* [ FIRST \"5\" REST #1 ] ] ] +PRED predsort +TICK bool +TNT null_tnt [ +MAIN tnt_main
[ +PRB \"0.98965190000000003\" +TAG \"NN\" ] +PRBS *null* +TAGS *null* ] +TO \"28\" +TRAIT
token_trait [ +HD token_head +IT italics +LB bracket_null +RB bracket_null +UW - ] ]"))))))))
(S (NP-SBJ (NNS (NNS (investors)))) (VP (VBD (VBD (welcomed))) (NP (DT (the)) (NN (NN (NN
(move.)))))))
```

Figure 4.3: Redwoods Export Example (20632006) 'Investors welcomed the move.'

described in Section 4.4 below to reference the phrase structure rule during processing. The initial parsing of each tree is done via regular expressions and then loaded into tree objects with the Natural Language Toolkit (NLTK) ParentedTree class (5). In cases where both trees are missing or empty, this is not done and no further processing with that sentence is attempted. Approximately 15% of the sentences are unparseable with the ERG and fall into this category. The failures are largely due to the lack of linguistic coverage in the ERG, computational constraints, or malformed text as input. It should be noted that the DeepBank itself makes use of a PCFG to fill most of the coverage gap above what the ERG is capable of parsing. The redwoods script makes use of the ERG when exporting parses, so the export functionality used in this thesis is by nature limited to only ERG-derived

DeepBank parses.

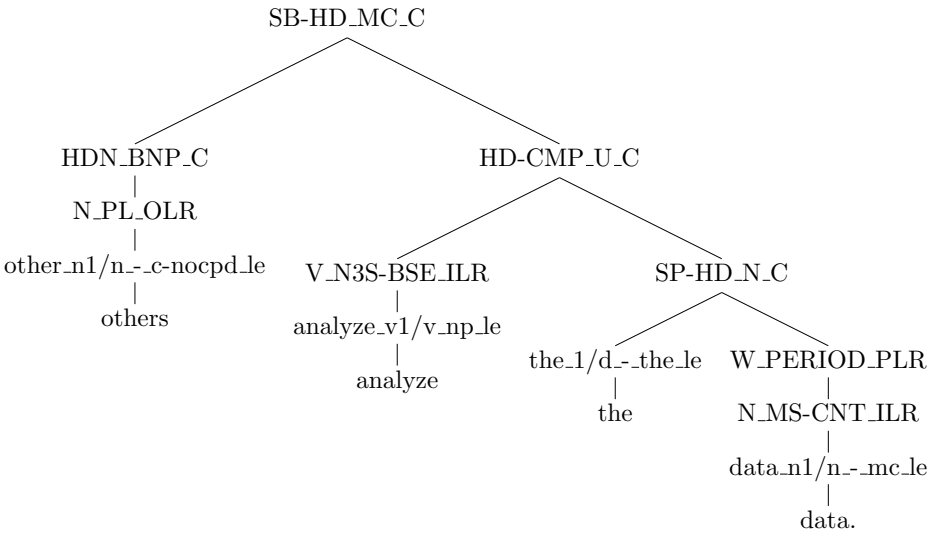

Figure 4.4: Derivation Tree Example 1

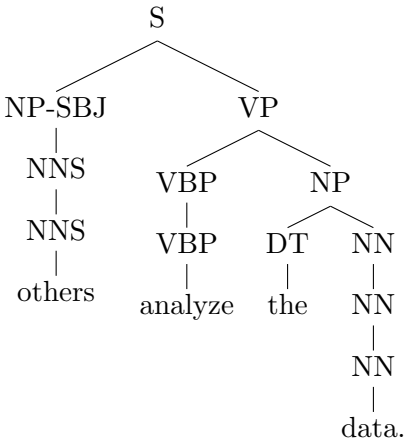

Figure 4.5: Node Labeled Tree Example 1

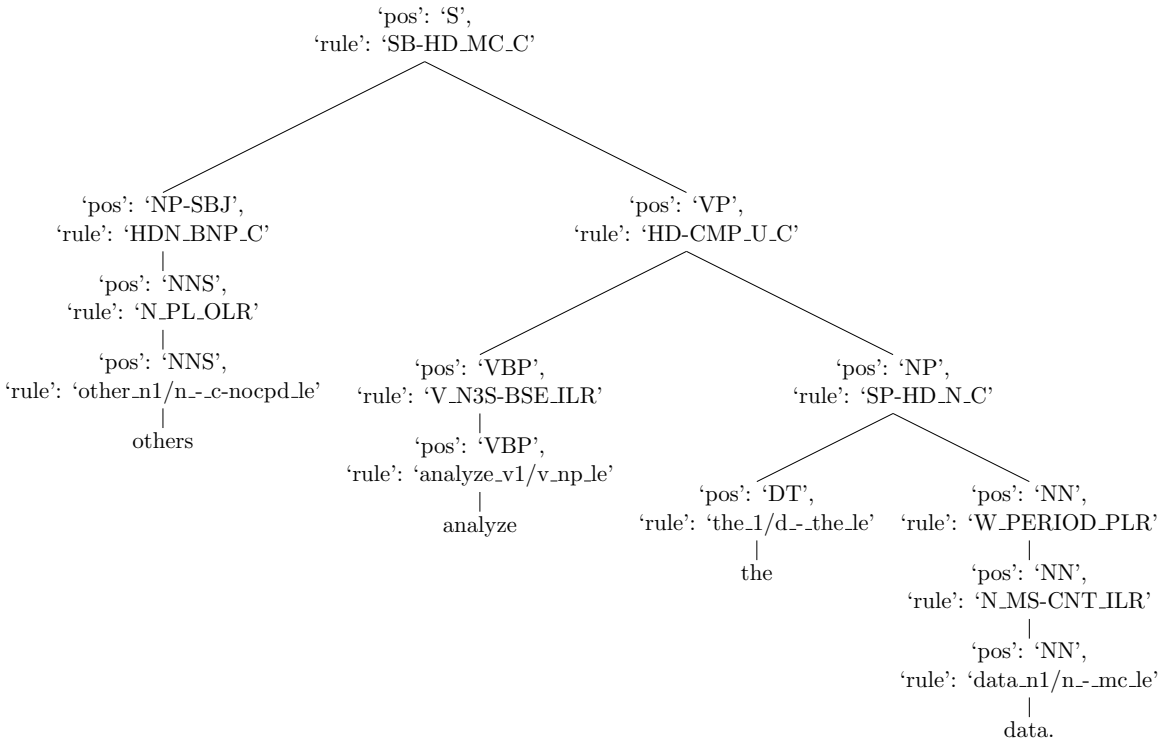

Figure 4.6: Combined Tree Example 1

4.4 Tree Transformation

There are many transformations required to get from a tree provided by the ERG to a tree similar or identical to the one provided by the PTB for the same sentence. The simplest transformation is that of retokenization; the ERG and PTB tokenize sentences differently and this must be reconciled. I do this by using REPP to generate the PTB tokenization for the sentence and then walk the leaf nodes of the ERG tree, splitting or combining nodes as required. An example output of the retokenization rule (B) is in Figure 4.7. I split nodes, such as a node which contains both a word and punctuation, at the lowest node above the word; the parent node which contains node label information is copied and the parent of that node has a new child inserted. I then modify the two nodes' words are to contain the REPP tokenized versions up to that point. This splitting continues for as long as the word on the tree node covers more than one REPP token, and then moves to the next leaf node in the tree. I combine nodes in a similar manner; this occurs with hyphenated

compounds—the ERG usually separates these compounds and the PTB usually combines them. For recombining nodes, I take the lowest common ancestor node as the new parent of the combined words and its label becomes their new label. The new label for the combined words is that of the rightmost word node. Depending on which rules are applied before this rule, in 62.5-71.7% of cases the labels in question are different. Neither label is correct in comparison to the PTB in 48.2% of cases and choosing the rightmost node is a 0.2-0.3% improvement over choosing the leftmost node. Many of the cases where neither label is correct involve hard-coded behavior laid out in the PTB guidelines, such as labeling all hyphenated modifiers JJ.

In addition to splitting on the words themselves and duplicating their label information, there is another form of splitting which must occur due to the ERG allowing words to contain whitespace; this behavior is not allowed with the PTB, which treats those instances as multiple words. The resolution for this is very similar to the rest of the splitting code, the only particular note is that instead of being concerned about REPP tokenization versus the tree, I duplicate the original parent labeled node as many times as there are words in its list and then edit each node to include only the nth word in order. An example output of this rule (A) is in Figure 4.9.

The tokens the combined tree contains are largely lowercase regardless of the original sentence; in order to enforce the capitalization used in the original sentence, I check that the only differences are capitalization or unicode and re-apply the original sentence formatting with a rename tokens rule (C), an example of which is in Figure 4.10. This rule does not require first applying the rules enforcing REPP tokenization.

The final tokenization difference is that of character sets; the PTB does not allow for unicode characters such as curved quotes, em dashes (—; U+2014), en dashes (–; U+2013), or ellipsis (...; U+2026). I replace these characters with their ascii equivalents. Double quotes in the PTB are also represented as doubled single quotes (``or '') and brackets are represented by special three character sequences (-LRB-, -RRB-, -LSB-, -RSB-, -LCB-, -RCB-) which indicate left or right and whether the brackets are round, square, or curly. I carry out these replacements with the replace leaf character rule; an example output for this rule (D) is in Figure 4.12. Most punctuation marks and the bracket character sequences

have their node label match the text in question; I created an additional rule to both enforce this behavior and enforce node labels in problematic mapping situations, like *to* and *-ing* verbs. An example output of this rule (E) is in Figure 4.14.

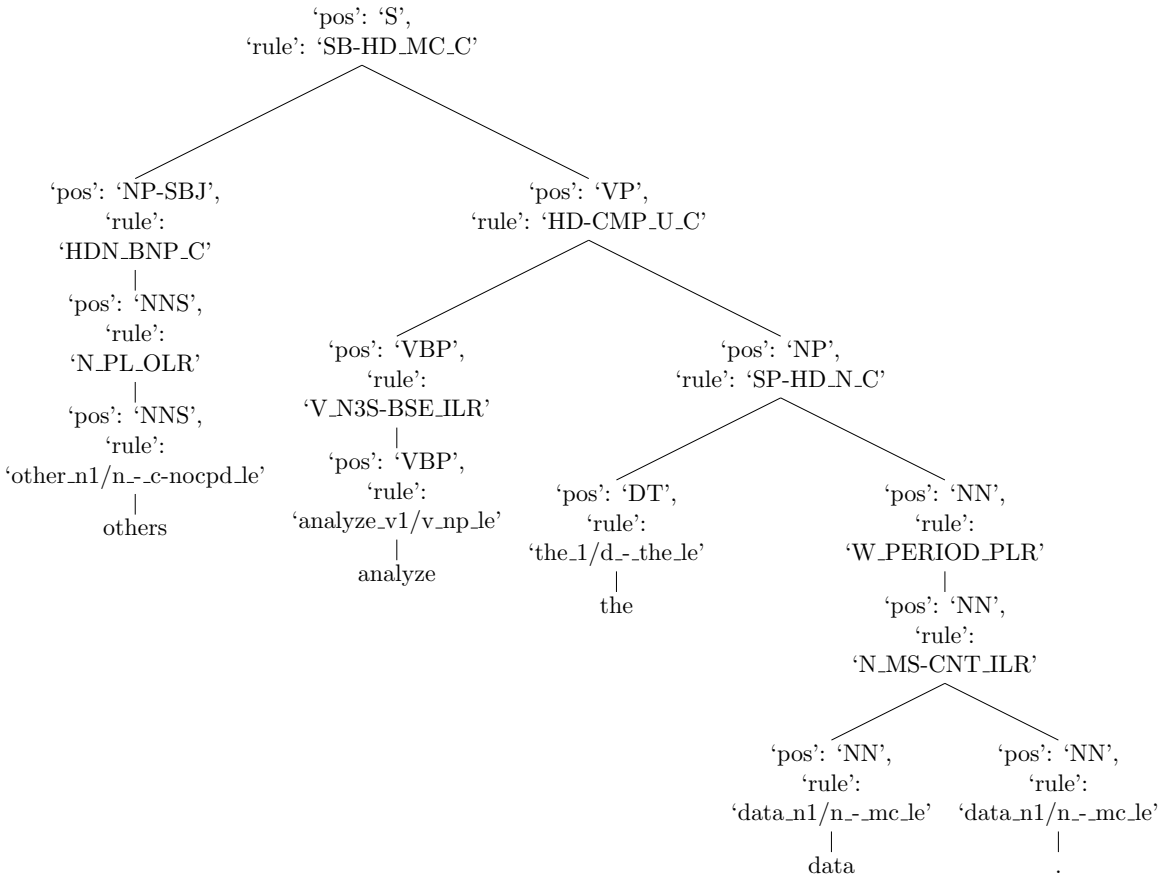

Figure 4.7: Tree 1 with Retokenize Rule (B) Applied

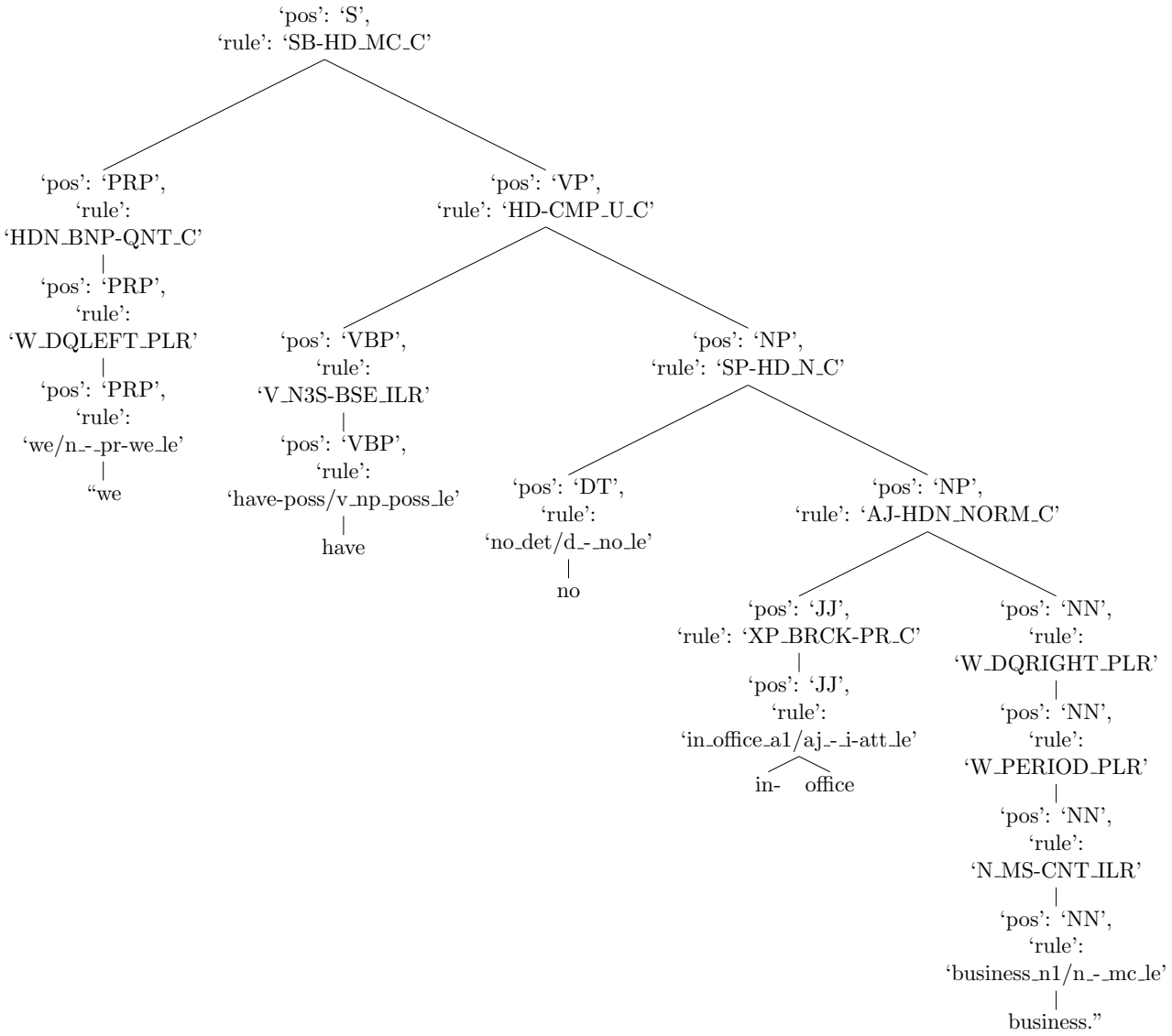

Figure 4.8: Example Tree 2 (No Rules Applied)

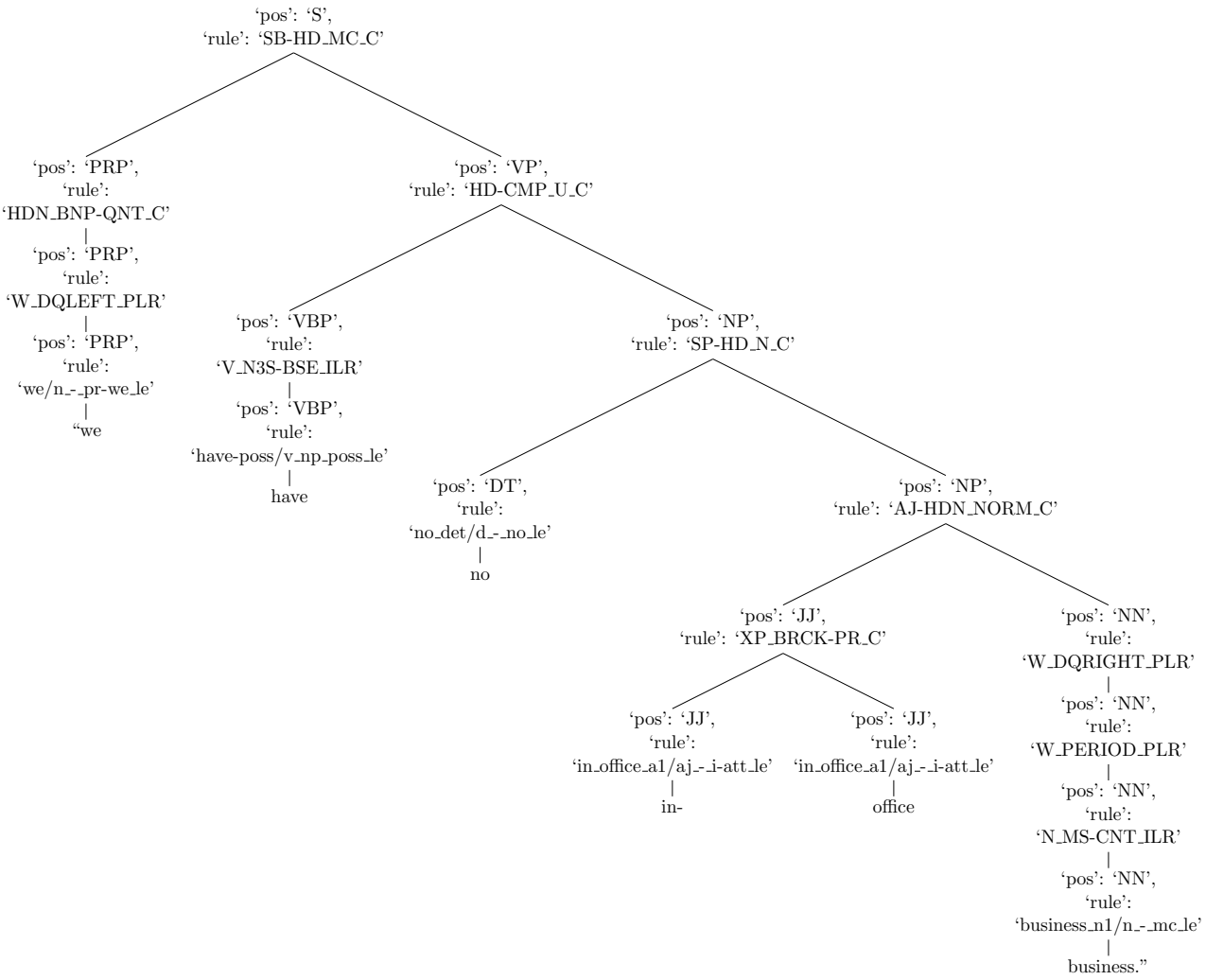

Figure 4.9: Tree 2 with Split Multiple Words Rule (A) Applied

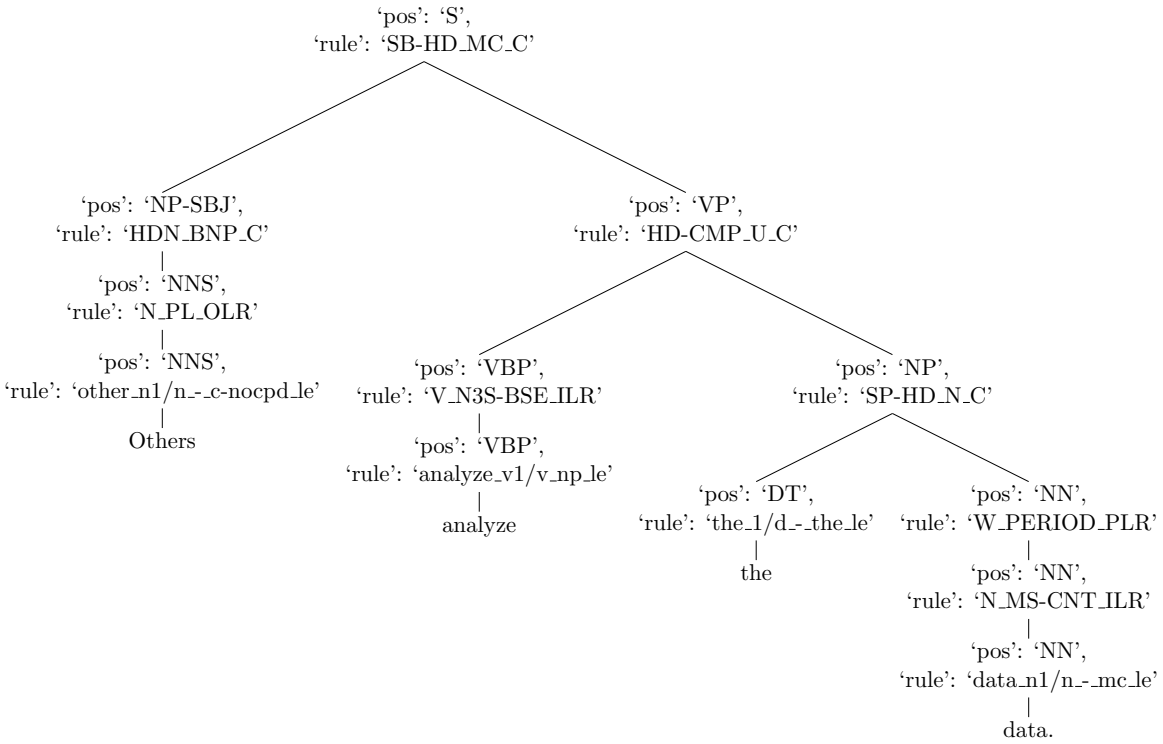

Figure 4.10: Tree 1 with Rename Tokens Rule (C) Applied

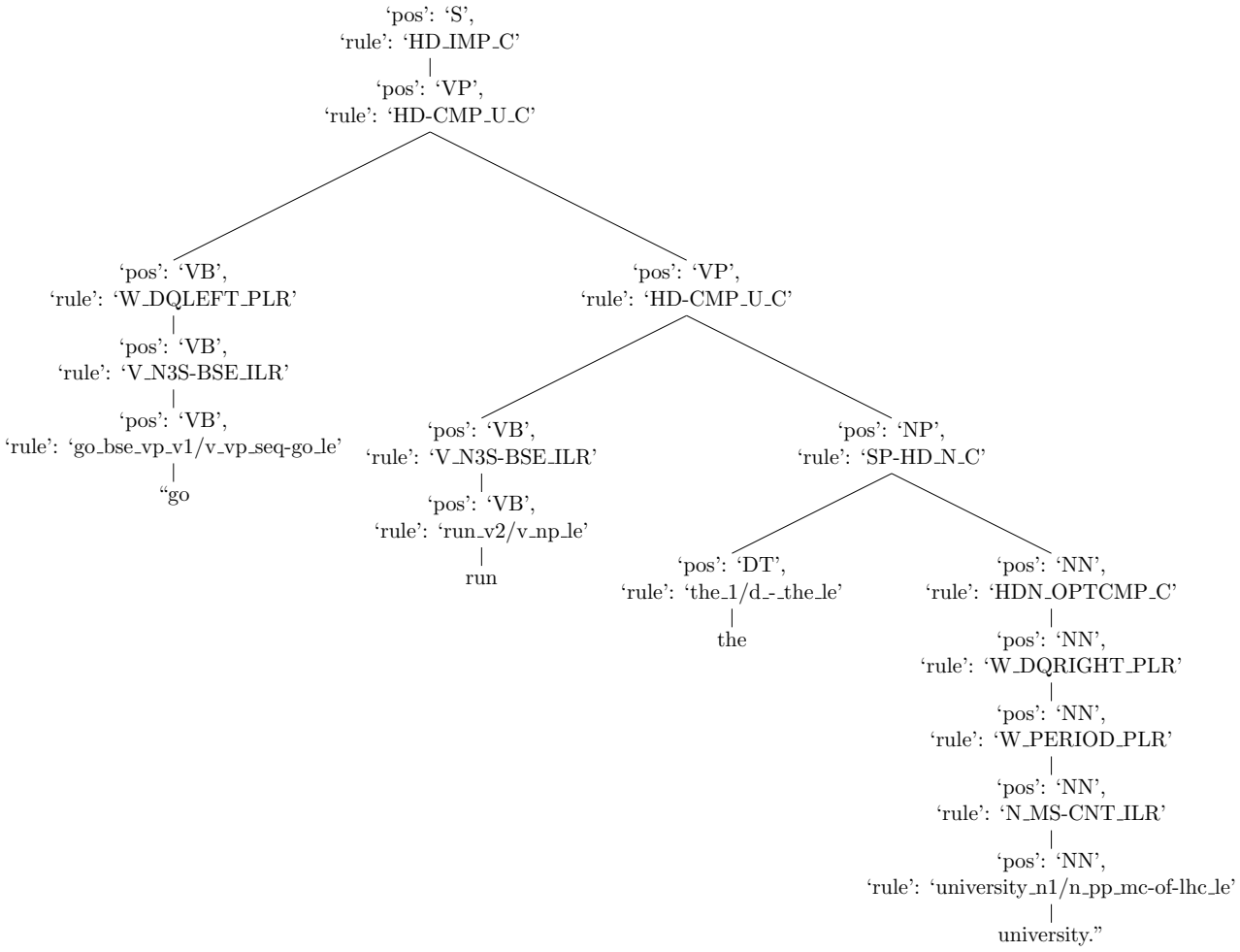

Figure 4.11: Example Tree 3 (No Rules Applied)

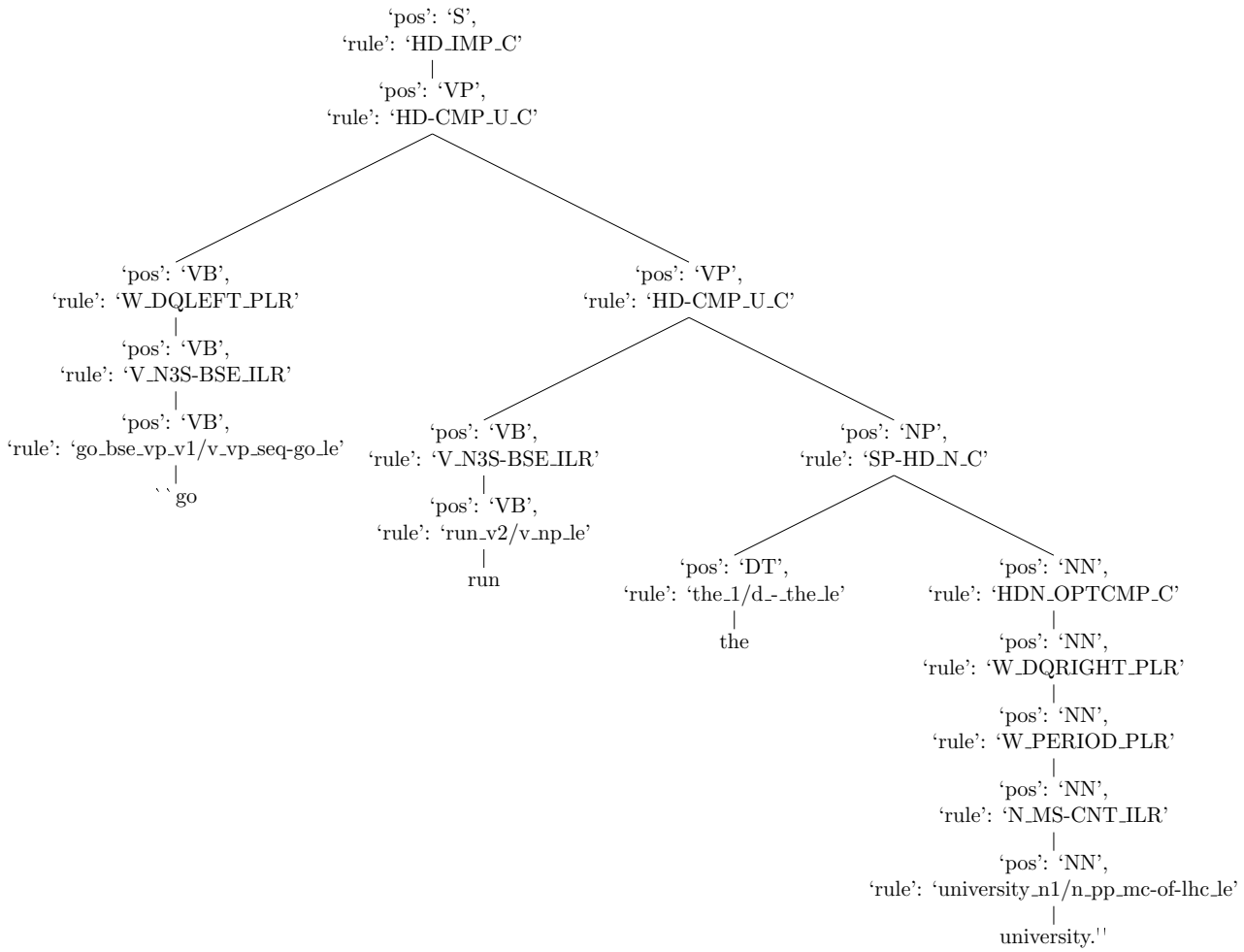

Figure 4.12: Tree 3 with Rename Tokens and Replace Leaf Characters Rules (C, D) Applied

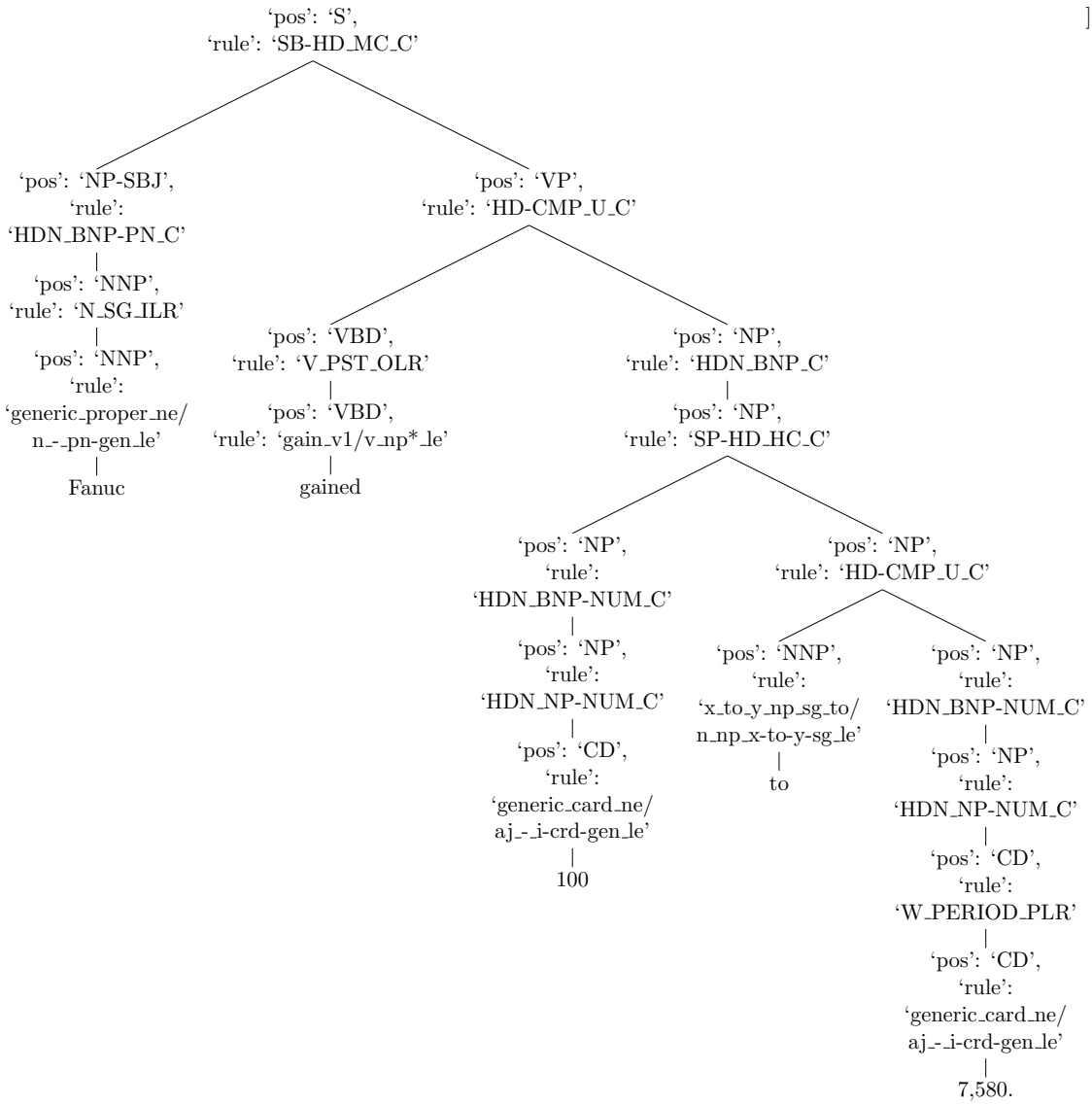

Figure 4.13: Example Tree 4 (No Rules Applied)

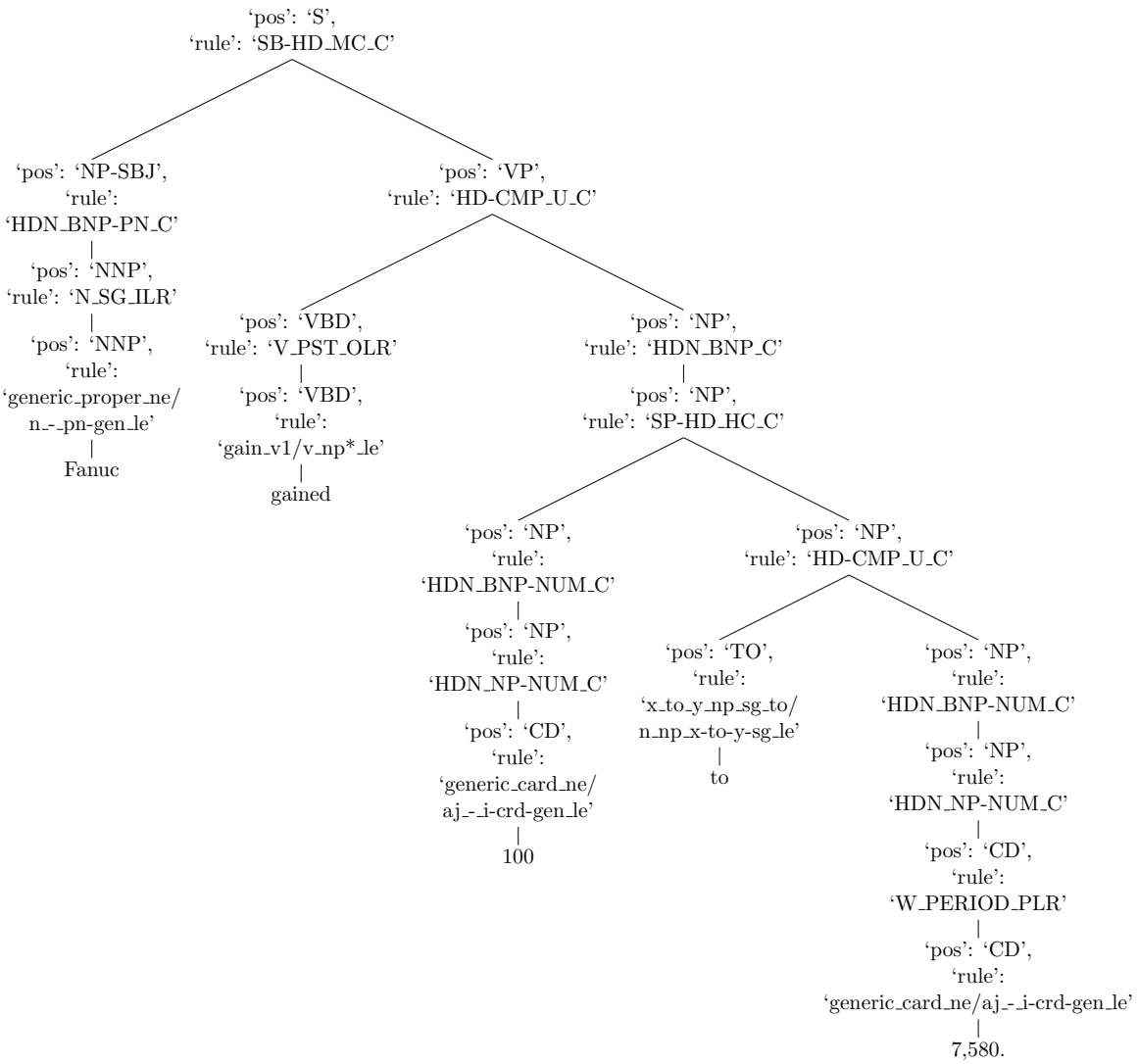

Figure 4.14: Tree 4 with Correct Node Label Rule (E) Applied

The ERG's lexical rules present as non-branching productions at the bottom of the tree and the ERG also contains non-branching phrase structure rules which can appear higher in the tree. In order to match the PTB structures, many of these unary series must be collapsed to single nodes. The exception for this collapsing behavior is when the leaf node node is a sibling of a phrasal node, or is attaching directly to a clausal node. At the leaf nodes, a collapse is done by selecting the uppermost node which has an applied lexical rule as the parent node and collapsing any higher or lower unary productions. In the ERG,

syntactic rules end in *-_c* by convention and lexical rules end in *-_ilr*, *-_olr*, *-_dlr*, *-_odlr*, or *-_plr*. Lexical entries make up the lowest entity above the word form itself; these all end in *-_le*. An example output of the unary collapse rule (K) is in Figure 4.15. After collapse, instead of a unary series above a leaf position, the leaf node should consist of the uppermost node in the series which had a lexical rule applied to it. The other nodes in the unary series are removed, except in the cases described above where a single phrasal node should also be preserved. These exceptions are largely due to the PTB requiring that most nodes with word-level labels can only adjoin to phrasal nodes through a phrasal parent (37). This rule also collapses unary series higher in the tree, though this is not as common an occurrence as unary series above leaf nodes.

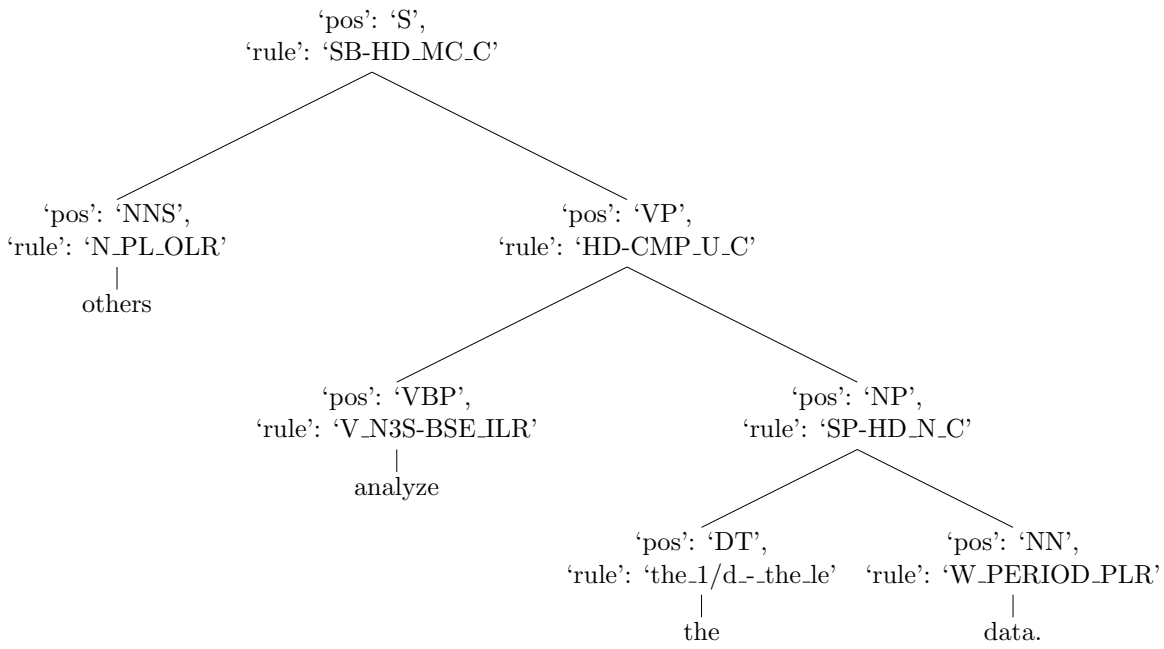

Figure 4.15: Tree 1 with Unary Collapse Rule (K) Applied

After the above steps the node labels on the tree should be largely correct. The remaining structural differences between the modified ERG-produced tree and the PTB tree largely consist of differing levels of attachment for various nodes. I have created rules to deal with the various attachment differences; I describe these rules now. Note that they can be applied

in any combination or order and that different orders of the same combinations will yield different results. All of the rules in this chapter are further detailed in Appendix A.

The PTB has sentence initial and sentence final punctuation attached directly to the uppermost S node; the ERG before retokenization had the punctuation attached to the nearest word and the retokenized trees have the punctuation as siblings of those nodes and not attached to the uppermost S. Paired mid-sentence punctuation like brackets, quotes, or commas around phrases are generally attached in the PTB as siblings to the phrases they surround. Single mid-sentence punctuation is usually attached at the highest possible dividing level in the PTB. In order to reproduce this behavior in the ERG-derived trees, each punctuation node and its label is moved higher in the tree, becoming a left or right sibling to the highest phrase in which it is, respectively, the leftmost or rightmost node in the modified ERG tree. An example output of the moving punctuation rule (F) is in Figure 4.16.

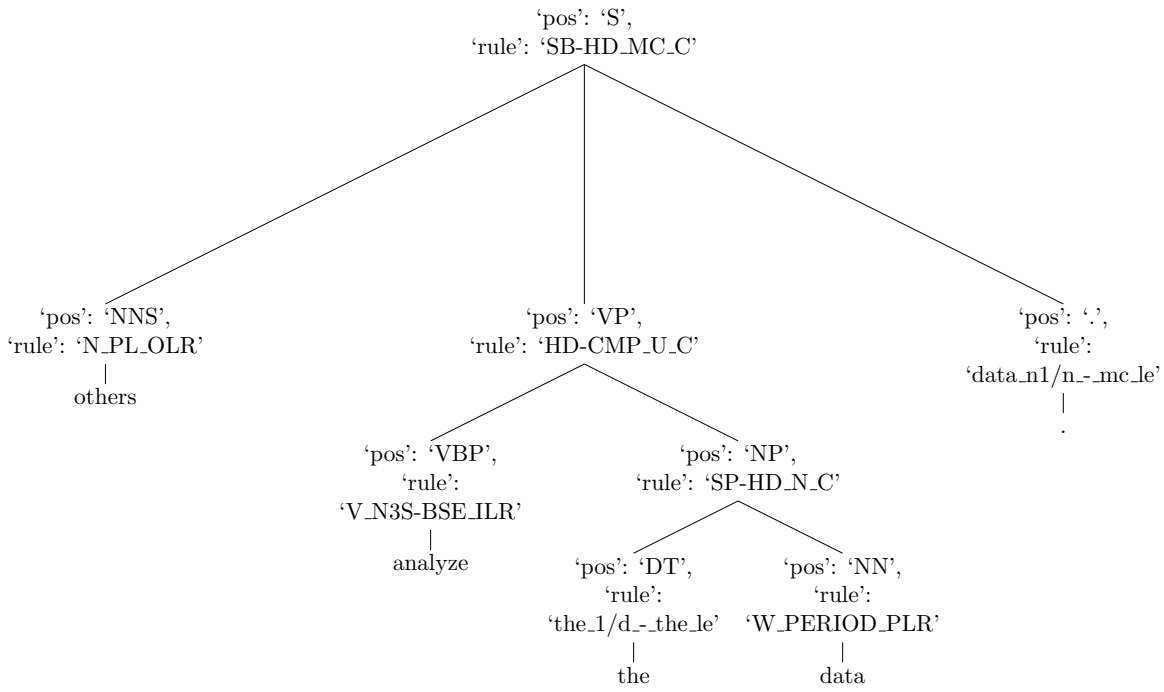

Figure 4.16: Tree 1 with Retokenize, Correct Node Label, Move Punctuation, and Collapse Unary Rules (B, E, F, K) Applied

In addition to sentence initial and sentence final punctuation, the PTB attaches the subject NP, highest VP, any modifiers preceding the verb, and fronted constituents to the uppermost S node. Subjects of sentences are contained within an NP-SBJ node; in order to comply with this I created a rule (M) to add an NP-SBJ parent node to any subjects which were not already labeled NP-SBJ by the labels file, which has further restrictions than the MRNAME on its NP-SBJ label entries. This is most commonly applied with pronouns; an example output is in Figure 4.18. In the ERG, the subject NP or fronted constituents and the highest VP are often also attached to the S, thus the main rules created here are for the pre-verb modifiers and for determining if both a subject NP and fronted constituents are present in the tree and moving any that are attached to nodes below the S. Verbal postmodifiers remain inside the VP; this is their expected position in the PTB. An example output of the verbal premodifier movement rule (J) is in Figure 4.19. Additionally, because the ERG licenses trees with at most two daughters per node, nested S structures may be present which must first be flattened before applying the above rules. S and S-like (SQ, SINV, etc) nodes which are children of another S node are removed and their children moved up; this continues until the chain of S nodes is broken by an intervening node. An example output of this rule (H) is in Figure 4.21. The difference between the verbal premodifier rule and the sentence flattening rule is the recursion in the sentence flattening rule; the verbal premodifier rule applies only to children of the VP whereas the sentence flattening rule can apply to both the children and levels further down.

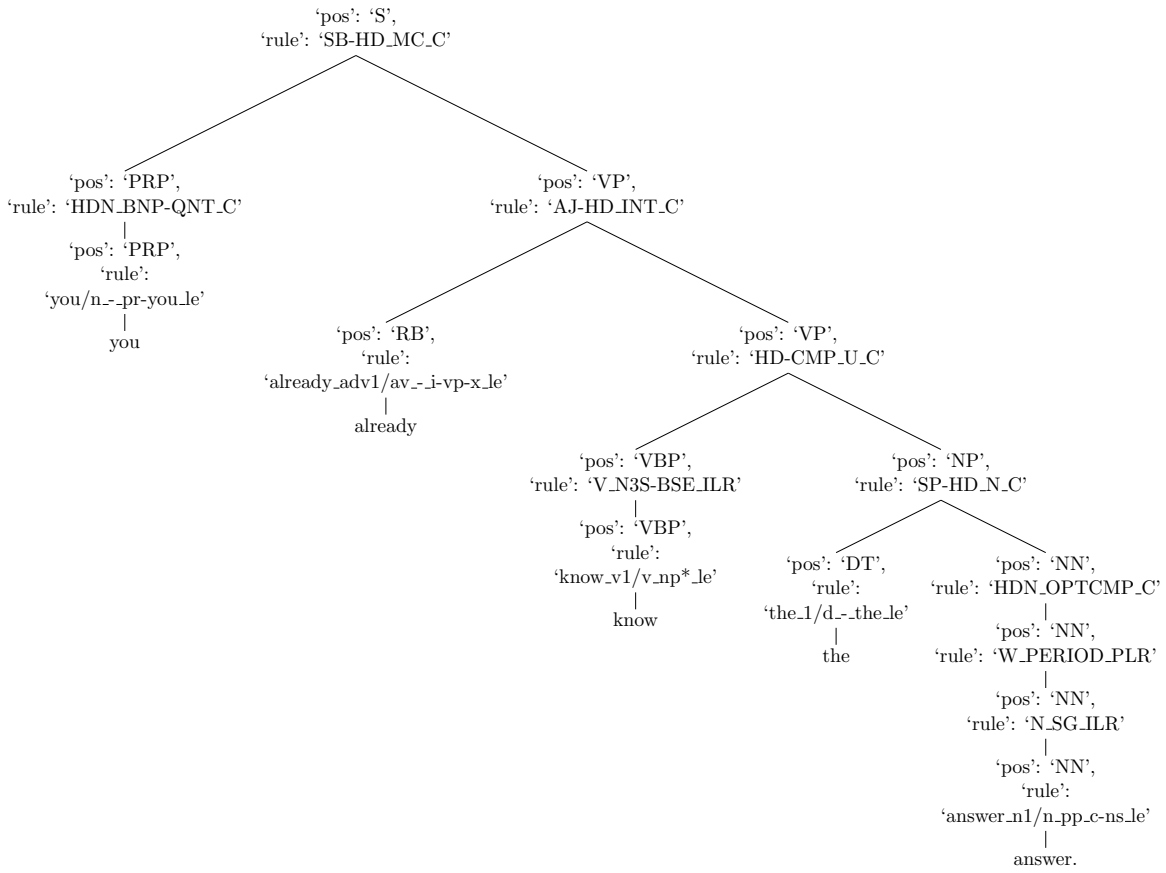

Figure 4.17: Tree Example 5 (No Rules Applied)

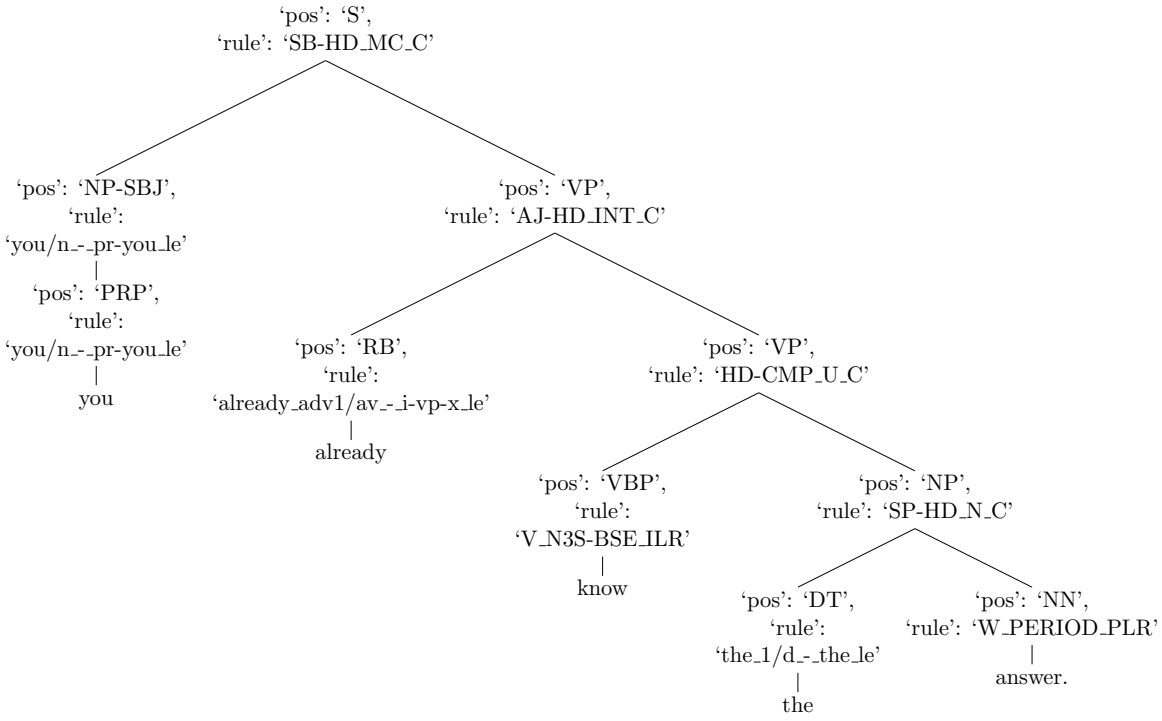

Figure 4.18: Tree 5 with Collapse Unary and Add NP-SBJ Rules (K, M) Applied

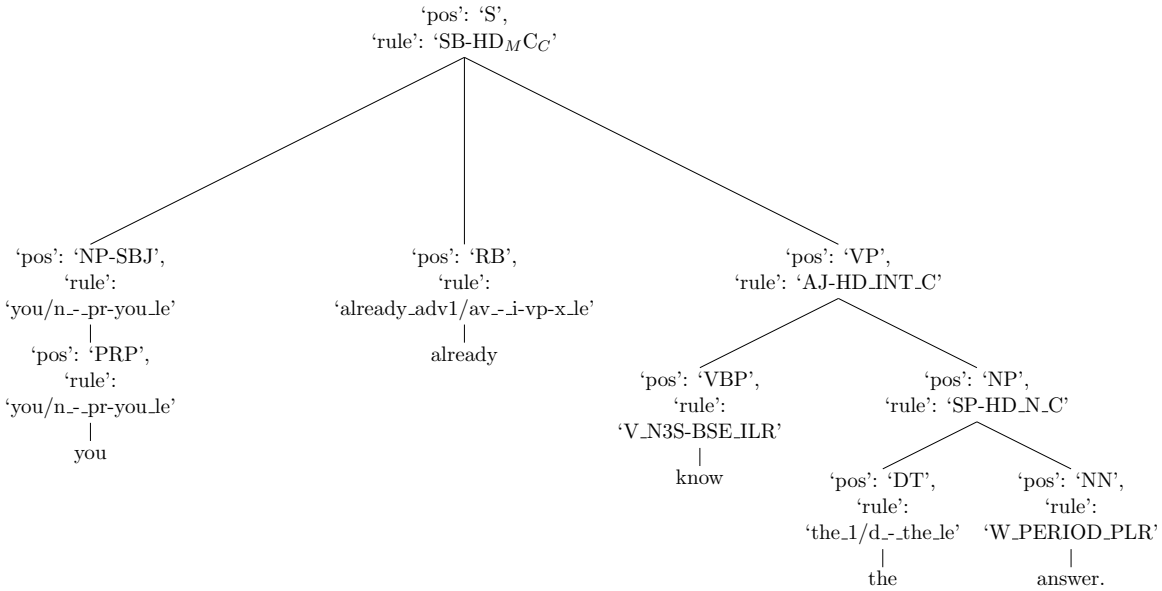

Figure 4.19: Tree 5 with Move Verbal Premodifiers, Collapse Unary, and Add NP-SBJ Rules (J, K, M) Applied

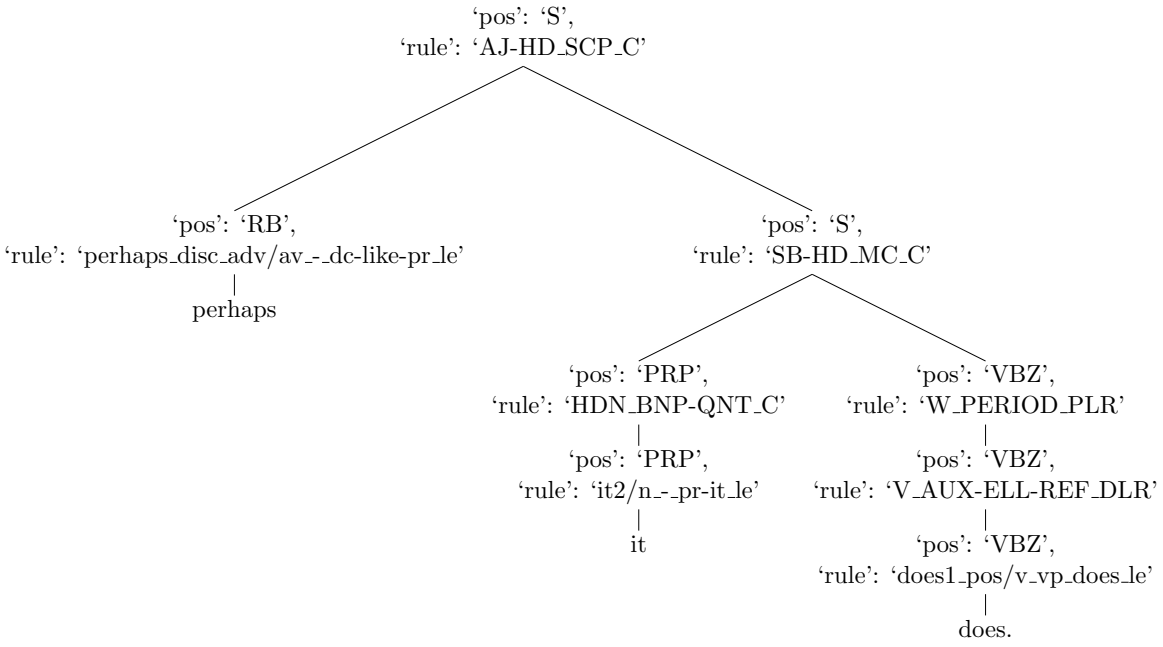

Figure 4.20: Tree Example 6 (No Rules Applied)

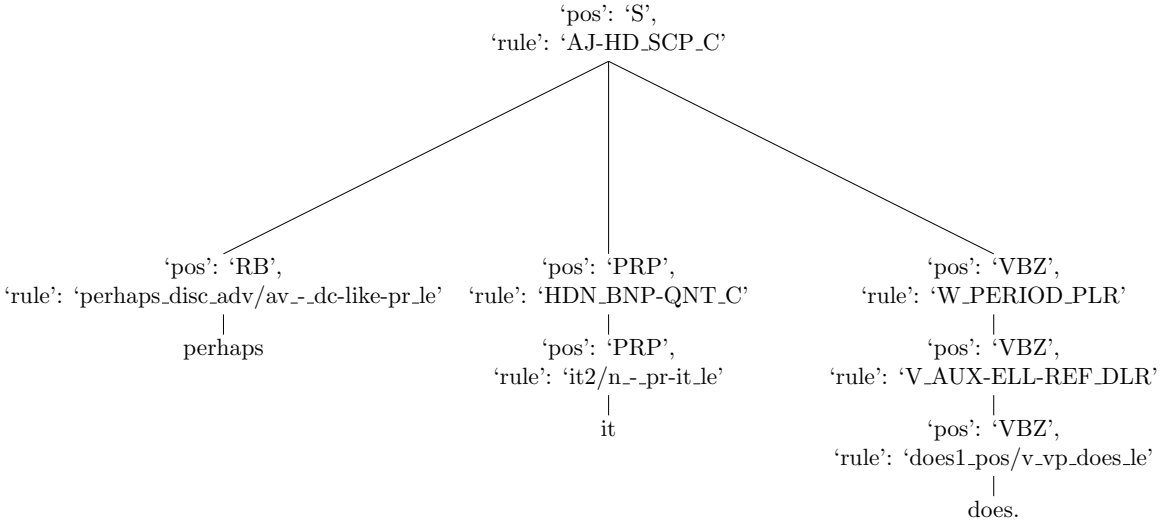

Figure 4.21: Tree 6 with Flatten Sentences Rule (H) Applied

In the PTB, premodifiers for non-verbs are usually placed within the phrase they are modifying; the ERG usually places many premodifiers as left siblings of the phrase they

are modifying. Moving these premodifiers often requires reapplying the unary-collapsing rule; for example, many NPs in the ERG are constructed of determiner and another phrasal nominal constituent (labeled NP with the new parse-nodes.tdl file), into which the determiner would be moved. An example output of the rule for moving determiners (G) is in Figure 4.25. An additional rule exists for postmodifiers of non-verbs, which are adjoined to the phrase in the PTB rather than attaching under the phrase like premodifiers. Much like the premodifier rule, the unary-collapsing rule (N) should be applied after this rule because moving child nodes out of the original parent node can result in the old parent and sibling becoming a unary node series which should be collapsed. An example of the output for this rule is in Figure 4.23.

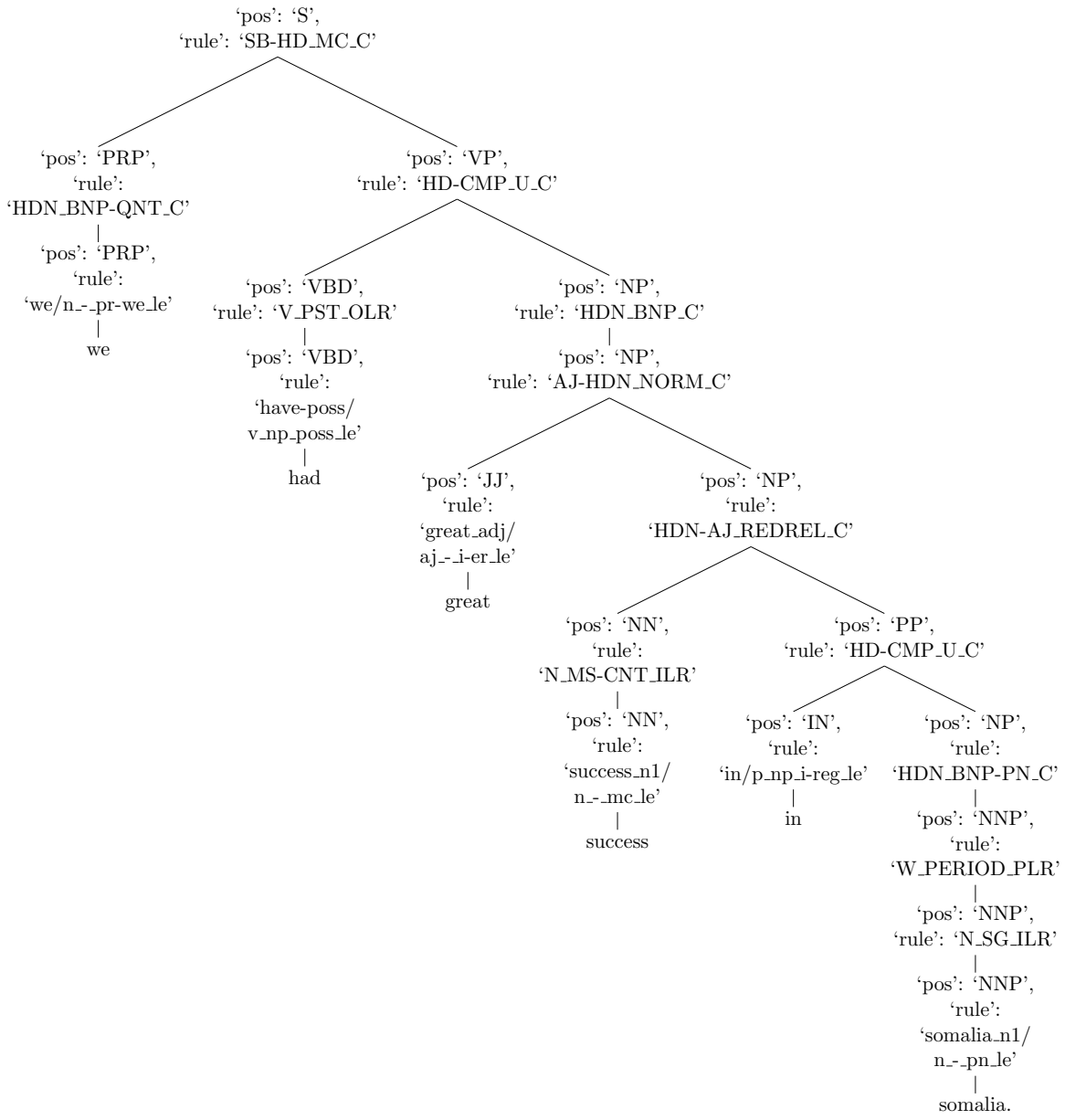

Figure 4.22: Tree Example 7 (No Rules Applied)

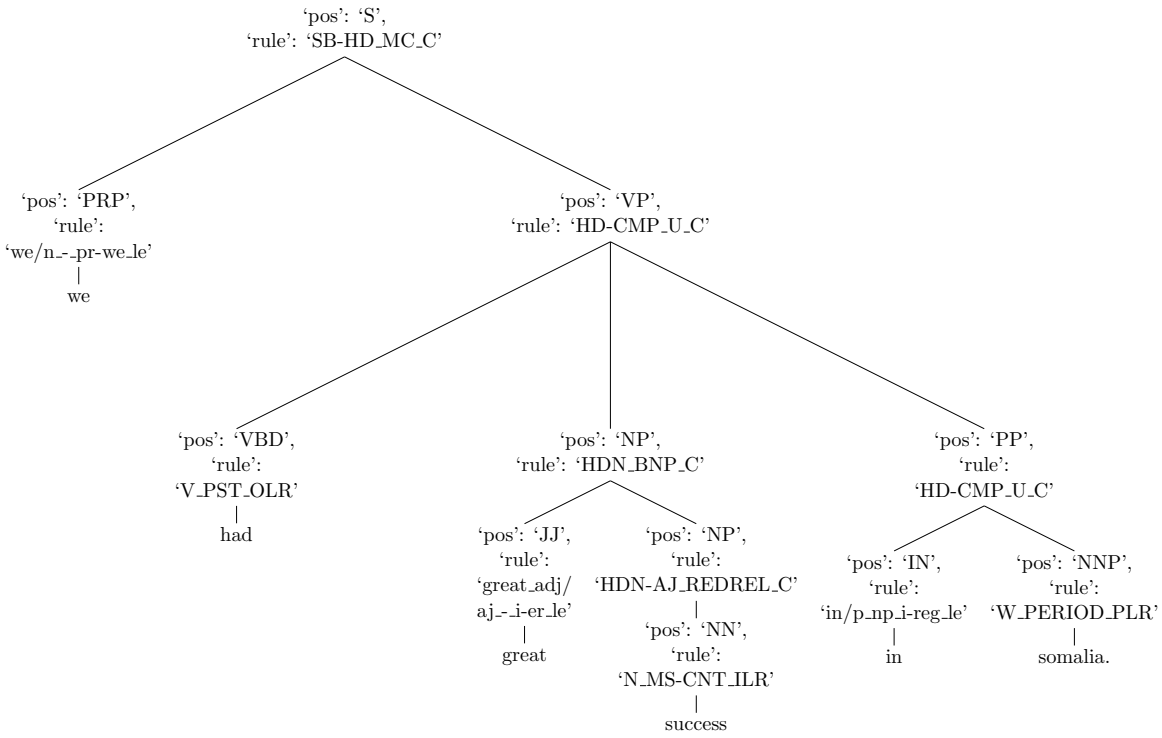

Figure 4.23: Tree Example 7 with Collapse Unary and Move Prepositional Phrases Rules (K, N) Applied

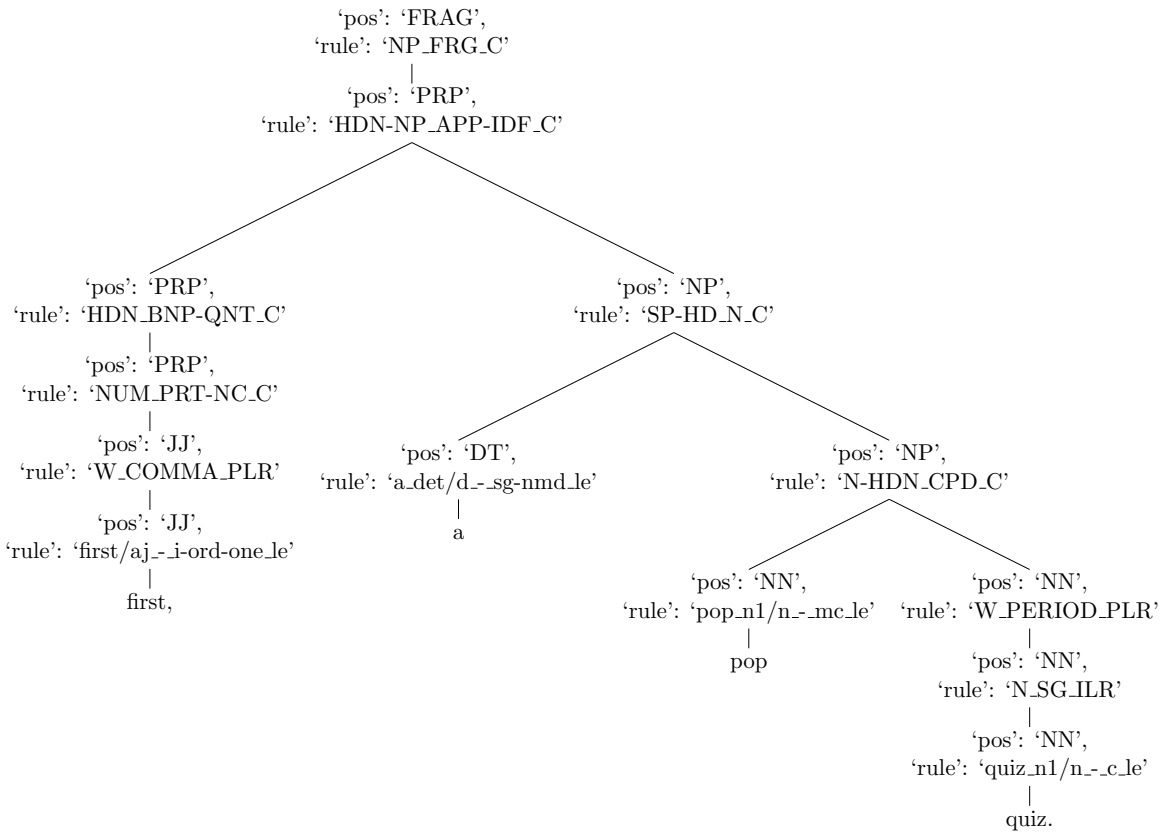

Figure 4.24: Tree Example 8 (No Rules Applied)

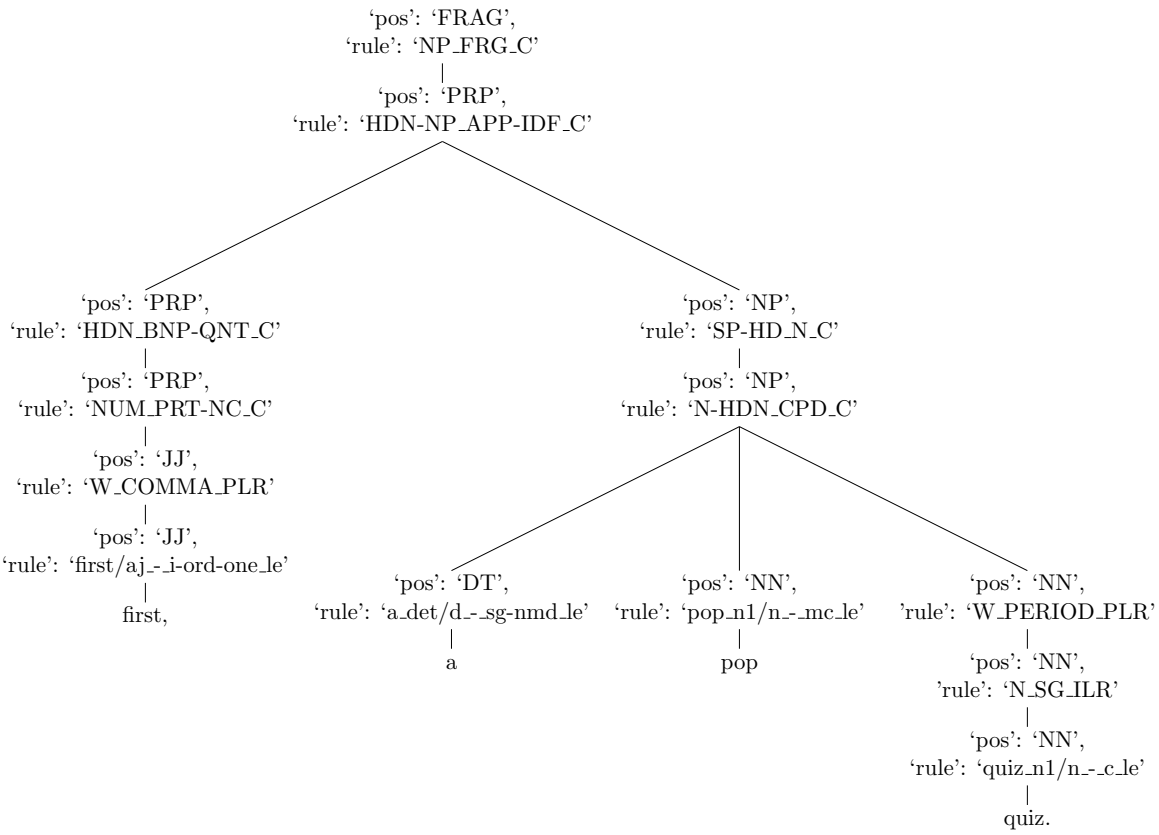

Figure 4.25: Tree 8 with Move Determiners Rule (G) Applied

In the ERG, node labels which are word-level node labels in the PTB are allowed to be siblings of phrase-level labels; this behavior is not allowed in the PTB for most node labels and parent phrasal nodes must be added before conjoining the node to other phrasal nodes. Due to what the PTB allows, this rule is not applied across all node labels; for example it is applied to RB nodes but not to IN nodes, which are allowed to adjoin to phrasal nodes. An example output for the RB parent rule (L) is in Figure 4.26. In order to enforce the requirement that non-leaf nodes must have phrasal node labels, an additional rule is created to convert word nodes' labels to phrasal node labels in phrasal positions. This rule is specifically intended to cover any gaps that the correct node labels (E) and RB parent (L) rules do not address, because training or testing the Stanford parser will fail if any sentence has disallowed combinations of word and phrasal node labels at the same level

of the tree. It applies to any non-preterminal node in the tree. The mapping of word labels to phrasal labels is shown in Table 4.1 and an example output of the correct tree labels rule (O) is in Figure 4.28.

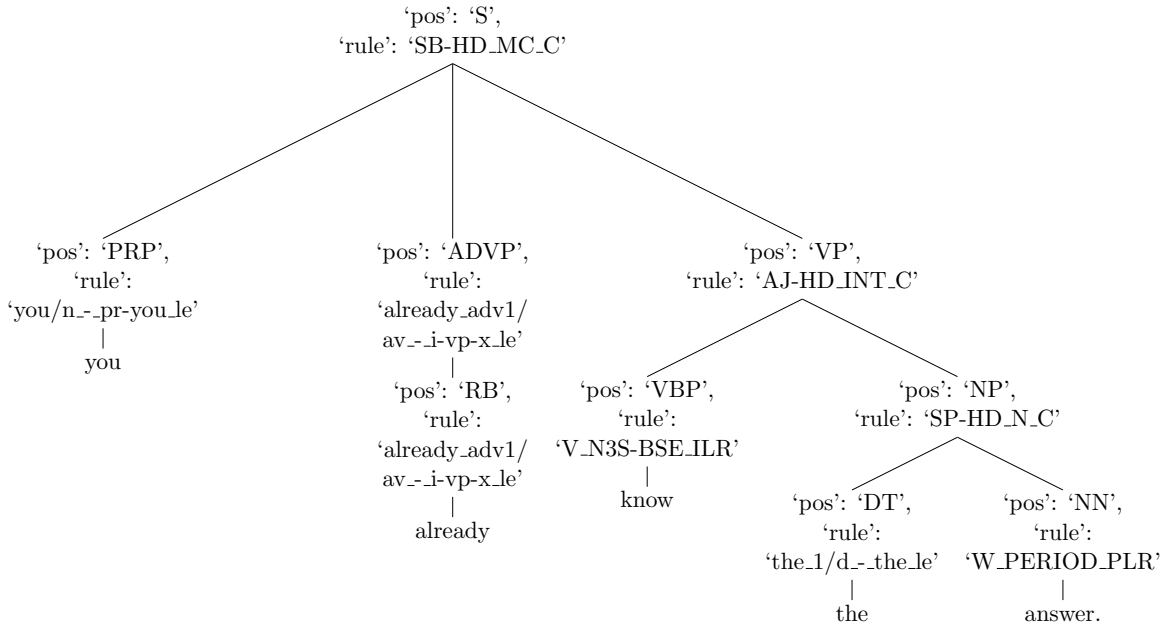

Figure 4.26: Tree 5 with Move Verbal Premodifiers, Collapse Unary, and Add RB-Parent Rules (J, K, L) Applied

Original Node Label	New Phrasal Node Label
NN	NP
NNS	NP
NNP	NP
NNPS	NP
PRP	NP
PRP$	NP
CD	NP
POS	NP
DT	NP
$ [1]	NP
. [1]	NP
`` [1]	NP
'' [1]	NP
VB	VP
VBD	VP
VBG	VP
VBN	VP
VBP	VP
VBZ	VP
MD	VP
IN	PP
RB	ADVP
RBR	ADVP
RBS	ADVP
TO	ADVP
JJ	ADJP
JJR	ADJP
JJS	ADJP
WDT	WHNP
WP	WHNP
WP$	WHNP
WRB	WHAVP
CC	CONJP
UH	INTJ
LS	LST
WRB	WHAVP
EX	S

Table 4.1: Remapping of Word Node Labels to Phrasal Node Labels (Rule O)[1]

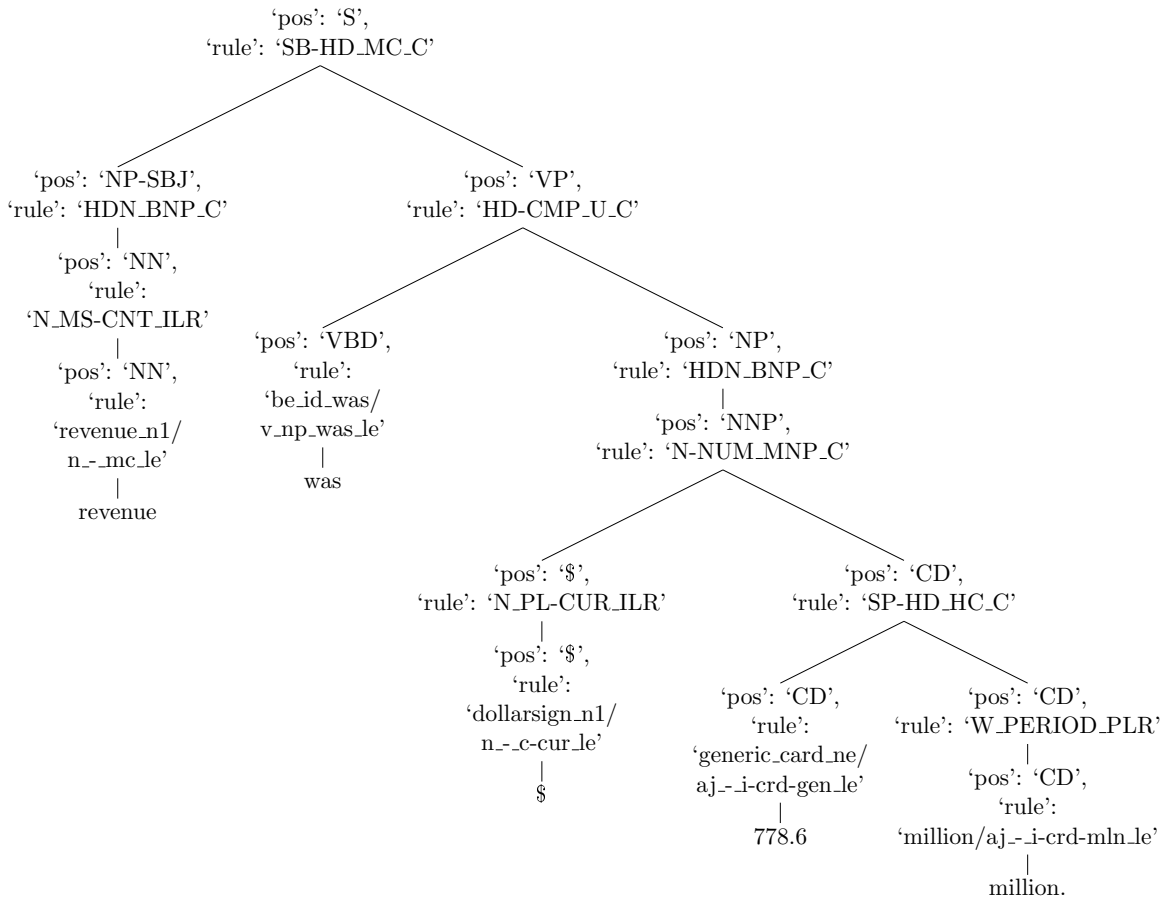

Figure 4.27: Tree Example 9 (No Rules Applied)

[1]Some node labels, such as quotes, are included purely in case the correct tree labels rule (O) is applied without the punctuation raising rule (F).

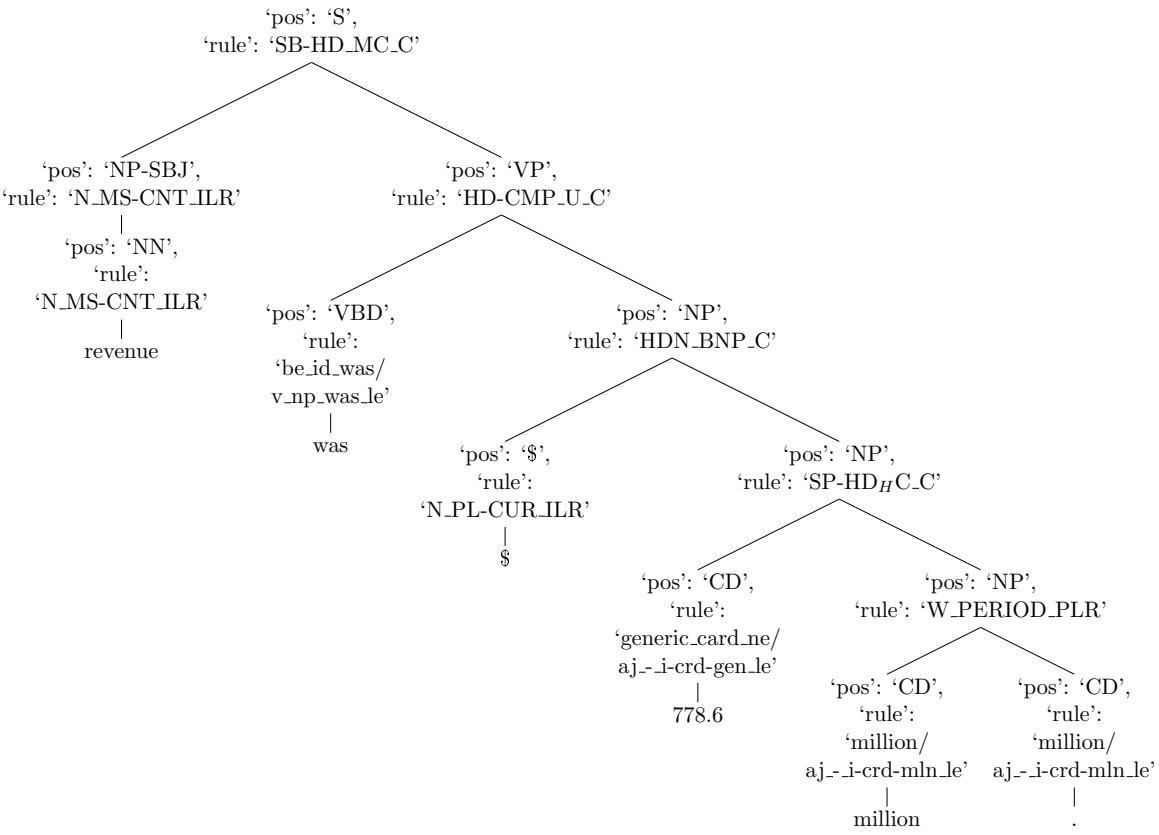

Figure 4.28: Tree 9 with Retokenize, Add NP-SBJ, and Correct Tree Node Labels Rules (B, M, O) Applied

Noun phrases also require flattening from the ERG binary structure; which is to say that NPs consisting of other NPs and word-level noun labels, modifiers, coordinating conjunctions, and punctuation should have each of those children moved to be at the same level. Child NPs are preserved when combining with non-noun phrasal nodes. These NPs largely adjoin adjectival or adverbial phrasal nodes. In the PTB these branchings are off of a single node whereas in the ERG this structure is represented in a series of binary joinings tailing off to the right. Similar behavior occurs within other non-verb phrasal nodes, such as adjective phrases. An example output of the noun flattening rule (I) is in Figure 4.30.

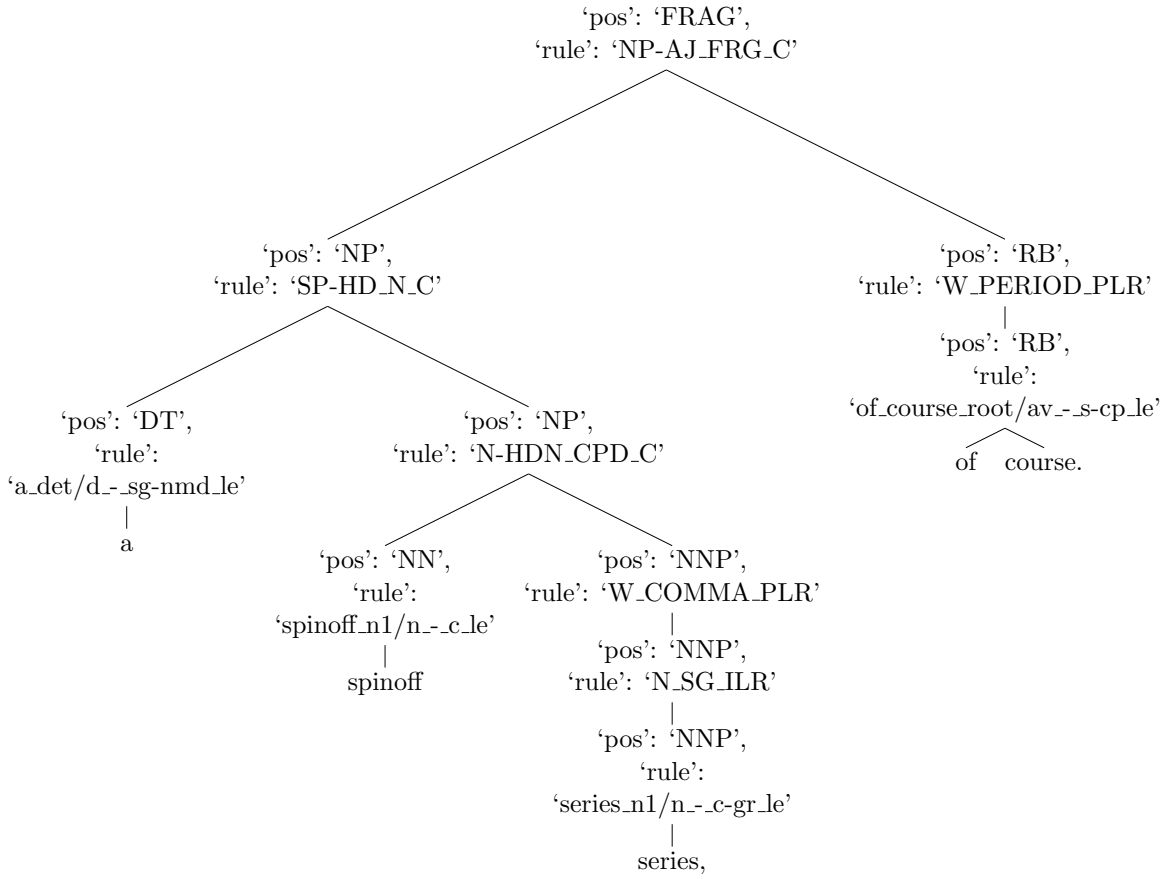

Figure 4.29: Tree Example 10 (No Rules Applied)

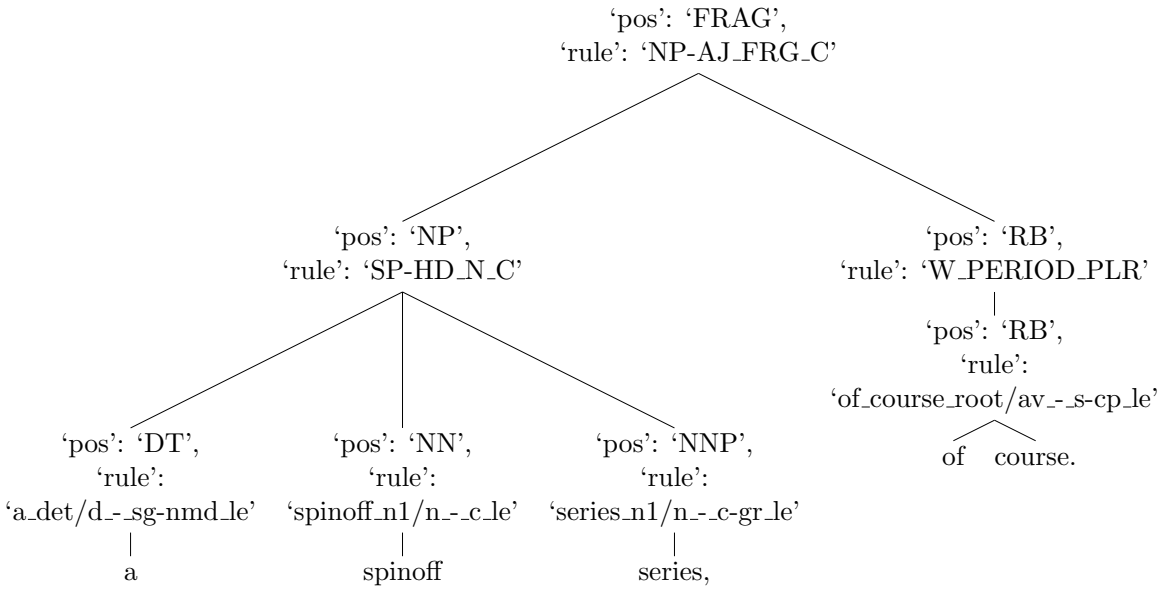

Figure 4.30: Tree 10 with Flatten Noun Phrase Rule (I) Applied

It should be noted that the PTB also creates NULL elements. These elements appear to be ignored by the Stanford parser and thus I make no effort to introduce them here. Adding these elements would require examining the rules and additional features for behavior similar to that defining the addition of NULL elements.

This section introduced fifteen rules for transforming DeepBank trees into more PTB-like trees. While this set of rules is intended to cover the largest differences between the ERG and PTB formats, it is expected that a number of differences will still remain. Further discussion of the differences between the altered DeepBank trees and the Penn Treebank is in Section 5.5.

4.5 Comparison of PTB-like DeepBank and PTB

Initial comparison of the modified DeepBank trees and the PTB trees involved removing the NULL elements from the PTB tree. I extended the NLTK ParentedTree (5) class to allow variations on outputting the trees; this enabled outputting only the node labels or additional information. I did comparisons during creation of the ruleset to determine agreement for leaf-level node labeling, tokenization agreement, level-by-level comparison for

tree structure, level-by-level comparison for node labeling, and equality of all subtrees of each tree. The combinations of these metrics allow for examining the tagging agreement as well as structural agreement while providing insight into the similarity of subtrees which may be off by one or more tree levels. These results are further described in the next chapter.

4.6 Deep Dependencies

In order to provide a good reference point for deep dependency parsing performance, the DDEC sentences were tested against the Stanford parser trained on PTB sections 02-21 as in Bender et al. 3 and against the Stanford parser trained on only those sentences also in the DeepBank. The outputs of these test runs required some trivial modification to strip out the initial processing outputs from the Stanford parser before being passed to the dependency evaluation script, depeval.pl. The rules described in previous sections were run in various combinations against the Redwoods-exported DeepBank parses and the results were output via the altered tree-printing functions to yield an .mrg file similar in format to the PTB and varying in degree of similarity to the PTB phrase structure trees. This provided the input for training the Stanford parser and then the DDEC sentences were used as the test data to provide the final results for the varyingly PTB-like DeepBank parse trees. The combinations of rules are described in further detail in the next chapter.

The detailed rules and methodologies described in this chapter are intended to take in DeepBank parses and output trees whose node labels and structures closely resemble the PTB in a format acceptable to the Stanford parser. The parser is then trained on this output and tested against the DDEC sentences to determine how the parsing of deep dependencies differs between the original PTB and the PTB-like DeepBank parse trees.

4.7 Summary

This chapter laid out the detailed methodology outlined in the previous chapter. I discussed the modifications to the ERG and how the DeepBank parses were exported. I then described how the exported parses were parsed into mutable tree structures, to which a series of transformation rules can be applied. I examined each of the fifteen transformation rules

and then touched on what types of metrics were used to gauge the effectiveness of those rules and how the resulting modified trees could be used to train the Stanford parser and test its dependency parsing.

Chapter 5

ANALYSIS OF MISMATCHES AND REMAINING DIFFERENCES

This chapter takes a look at the mismatches and remaining differences between the altered DeepBank parses and the PTB. Some degree of mismatch is expected simply due to the DeepBank parses being automatically generated by the ERG and the PTB parses being largely manually annotated. As a result of its being manually annotated, the Penn Treebank is not completely consistent with its guidelines; for example, when *U.S.* is the final token in a sentence many (but not all) parses add an additional period that does not exist in the original sentence. As another example, determination of whether or not to label a phrase as parenthetical (PRN) is left in large part up to the annotator's intuition according to Bies et al. (4). Thus, even if the DeepBank exported parses were transformed to have 100% agreement with the Penn Treebank bracketing and node label guidelines, the results are still expected to not fully match with the Penn Treebank itself. Additionally, I did not attempt to obtain 100% agreement with the PTB guidelines. I instead focused on creating rules which cause the greatest reductions in the number of tree differences per rule as possible. This was largely determined through iterative rule creation using the metrics below, with some additional rules being required for Stanford parser input acceptability as described in Chapter 4. The remaining differences between the modified DeepBank parses and the PTB are discussed further below in Section 5.5.

5.1 Transformation Rule Analysis

The implemented transformation rules are not comprehensive; they are intended to cover the majority of differences between the ERG and PTB tree structures and node labeling but do not completely close the gap. Each rule also has a varying effect on the final output. A brief description of each rule is in Table 5.1 for reference.

Initial evaluation of the rules and combinations of rules for transforming the exported

ID	Rule Name	Description
A	split_multiword_leaves	When the leaf nodes contain multiple words, duplicate the leaf node and change the original and duplicate(s) to contain one word each.
B	retokenize_tree	Alter the leaf nodes of the tree to conform to REPP tokenization.
C	rename_tokens	Alter the leaf node tokens to match the REPP characters and capitalization, from the original sentence.
D	replace_leaf_chars	Replace unicode characters with ascii equivalents, escape slashes, change bracket characters to PTB bracket character sequences (-LRB-, etc).
E	correct_node_pos	Alter leaf node labels; enforces *to* behavior, punctuation label names, etc.
F	move_punctuation_on_erg_tree	Move punctuation higher in the tree.
G	move_determiners_on_erg_tree	Move determiners to be left siblings of their original parent.
H	flatten_sentential_subtrees	Move children of non-root S nodes which are in a direct line to the root S node up to flatten sentence structures.
I	flatten_noun_phrases	Flatten noun structures when they only contain non-verb labels.
J	move_verb_premodifier_phrases	Move verbal premodifiers to be left siblings of their original parent node.
K	modified_collapse_unary	Collapse unary productions.
L	add_rb_parent_phrases	Add an ADVP parent node to RB nodes which have phrasal siblings.
M	add_initial_npsubj_trees	Add an NP-SBJ parent node where one does not already exist for non-verbal/non-punctuation nodes which are to the left of the main verb phrase (if one exists).
N	move_prepositional_phrases	Move prepositional phrases (postmodifiers) in NP/ADJP phrases to be right siblings of their original parent.
O	correct_tree_node_pos	Modify non-leaf word-level node labels to phrase-level node labels.

Table 5.1: Basic Rule Descriptions and Identifiers

DeepBank-selected parses involved a variety of comparisons aimed at determining the tree structure and node label accuracy. I calculated the baseline accuracy by comparing the raw exported DeepBank parse output against the Penn Treebank through several metrics.[1] The baseline numbers are expected to be quite low, as the exported parses are still in ERG structure. Each individual metric is further described below.

5.2 Leaf-level Comparisons

The first two comparisons are the leaf-level (word-level) tokenization and the leaf-level node label accuracy between the trees. The tokenization metric provides insight into the accuracy of REPP tokenization and the application of that tokenization to the tree (rule

[1] All of the metric calculations are in `https://bitbucket.org/caelum/thesis-transform` at the end of the main transformation script, new_parse_export_files.py.

B). Some differences between the REPP tokenization and PTB tokenization are expected; for example, the PTB occasionally adds an extra period to the end of acronyms like *U.S.* when they occur at the end of a sentence, while REPP does not add an extra period (15). Tokenization results are in the Sentence Tokenization % column in Table 5.2 and Table 5.3. They topped out at 98.805% for both combinations and individual rules, almost entirely provided by rule B. The baseline was 7.186%, which is the percentage of sentences which already had identical tokenization to the corresponding parse in the PTB.

The leaf-level node label accuracy metric does not provide much insight into overall tree structure but does provide information about the accuracy of the modified ERG labels file and the effectiveness of rules which collapse nodes (K), split nodes (A, B), or otherwise alter node labels (E, O). The results for this are in column Leaf Node Labels % in Table 5.2 and Table 5.3. The baseline for this was 38.482%, with individual rules in the same column topping out at 74.316% and rule combinations at 91.312%. The calculation of this metric is heavily dependent on tokenization matching up between the PTB and modified DeepBank trees. In order to provide further insight for the effects of individual rules, the column Leaf Node Labels % With Rule B is in Table 5.2. The baseline here is the same as the result for rule B, 74.316%, and individual rules topped out at 86.061%.

5.3 Tree Structure Comparisons

For the first tree structure comparison metric, level-by-level comparisons of numbers of nodes and their respective positions provide overall accuracy. The difficulty with this metric is that disagreements closer to the root node in the tree are weighted much more heavily than disagreements closer to the leaf nodes of the tree. This is simply due to the higher-level disagreements effectively causing entire subtrees to be discounted due to being off by one level. When applied to individual levels of the tree during development this metric proved very useful in determining rule effectiveness, although when applied to the entire tree it proved to have generally low numbers overall; the results for this are in column Layer-by-Layer Tree Structure % in Table 5.4 and Table 5.5. The baseline was 5.836% and individual rules topped out at 7.197% while rule combinations topped out at 14.317%.

I also combined the above metric with a node label comparison; as expected this had

ID	Rule	Sentence Tokenization %	Leaf Node Labels %	Leaf Node Labels % With Rule B
	baseline	7.186	38.482	74.316
A	split_multiword_leaves	7.186	38.482	74.316
B	retokenize_tree	98.805	74.316	74.316
C	rename_tokens	7.186	38.482	74.316
D	replace_leaf_chars	7.186	39.126	75.550
E	correct_node_pos	7.186	39.494	86.061
F	move_punctuation_on_erg_tree	7.186	38.482	74.316
G	move_determiners_on_erg_tree	7.186	38.482	74.316
H	flatten_sentential_subtrees	7.186	38.482	74.316
I	flatten_noun_phrases	7.186	38.482	74.316
J	move_verb_premodifier_phrases	7.186	38.482	74.316
K	modified_collapse_unary	7.186	39.397	76.143
L	add_rb_parent_phrases	7.186	38.482	74.316
M	add_initial_npsubj_trees	7.186	38.481	74.317
N	move_prepositional_phrases	7.186	38.482	74.316
O	correct_tree_node_pos	7.186	38.482	74.316

Table 5.2: Individual Rule Comparison vs Baseline: Leaf Node Comparisons

worse numbers but provided some insight into how many node level disagreements were unrelated to tree structure. In general it appeared to show that most node label disagreements were also structure-related. The baseline was 4.538%, with individual rules topping out at 6.257% and rule combinations at 12.808%; these results are in column All-Tree Node Labels % in Table 5.4 and Table 5.5.

In order to provide further insight into level-by-level disagreement caused by otherwise correct subtrees occurring at a lower or higher level of the parse tree than expected, I compared the full sets of subtrees in each tree. This metric takes into account the node labels in each subtree, so it is not a purely structural comparison. As an individual rule comparison this had little variation since no individual rule was designed to broadly correct both the tree structure and node labels. However, as a combined rule comparison this was one of the more useful metrics because it provided more insight into overall tree and node label accuracy while not harshly punishing level-of-attachment disagreements. This topped out at 56.513% for combinations of rules and 33.578% for individual rules from a baseline

Rule IDs Used	Sentence Tokenization %	Leaf Node Labels %
baseline	7.186	38.482
ABCDEFGHIJKLMEO	98.800	91.183
ABCDEFGHIJKLMNO	98.800	89.869
ABCDEFGHIJKLMO	98.800	89.869
ABCDEFGHIJNKLMEO	98.800	91.183
ABCDEFGJIHKLMEO	98.800	91.183
ABCDEFGJIHKLMO	98.800	89.869
ABCDEFGJKLMHIEO	98.805	91.311
ABCDEFGKHIJLMEO	98.800	91.312
ABCDEFGNJKLMHIEO	98.805	91.311
ABCDEKFGHIJLMEO	98.800	91.248
ABCDEKLFGHIJMEO	98.800	91.248
ABCDEKLFGHIJMNEO	98.800	91.248
ABCDFGHIJKLM	98.800	77.244
ABCDFGHIJKLMEO	98.800	91.117
ABCDFGHIJKLMO	98.800	77.244
BCDEFGHIJKLMEO	98.778	91.122
BCDEFGHIJNKLMEO	98.778	91.122
BCDEGHIJKME	98.800	91.109
BHGIEEKDJACFOML	98.800	87.468
BIGHKCJMDEEAFLO	98.800	91.040
BIKHGCJMDEEAFLO	98.800	91.040
CDEFGHIJKLMEO	7.183	41.045
DEFGHIJKLMEO	7.183	40.983

Table 5.3: Combined Rules Comparison vs Baseline: Leaf Node Comparisons

of 22.211%, in column Subtree % in Table 5.4 and Table 5.5.

To provide a more detailed tree distance metric I calculated the pq-gram distance, which is an approximation of tree edit distance described by Augsten et al. (1). The values range between 0 and 1, with lower values being better. The comparison between identical trees yields a 0; due to this being an approximation 0 can also be returned for very similar but not identical trees. The value returned for completely dissimilar trees is 1. For combinations of rules it provides good insight into tree differences while avoiding the performance problems that fully calculating tree edit distance would entail. With this metric rule combinations dipped to 0.877 and individual rules to 0.941 from a baseline of 0.970, shown in the column

Average pq-Gram Distance in Table 5.4 and Table 5.5. A contributing factor to the relatively high numbers here are the differences in parse selection and clause structures noted further below in Section 5.5.

As previously noted, the individual rule comparisons are intended solely for validation of a transformation rule's benefit against the baseline. Additionally, the expected final combination of these rules with the output is not expected to fully match the PTB output for reasons described in the next section.

The rule combinations in Table 5.5 and Table 5.3 were chosen through a mixture of experimentation and permutation. ABCDEFGHIJKLMEO was determined to be fairly successful through experimentation during the development process, for example, while BHGIEEKDJACFOML was a randomly selected permutation. Certain rules make sense to apply in order; for example rule D (replace leaf characters) before rule E (correct node labels) because rule E does not account for unicode characters or the replacement character combinations for brackets. Others, such as the unary collapse rule (K), are more flexible in their ordering relative to other rules but will yield different results when applied in different orders.

ID	Rule	All-Tree Node Labels %	Layer-by-Layer Tree Structure %	Subtree %	Average *pq*-Gram Distance
	baseline	4.538	5.836	22.211	0.970
A	split_multiword_leaves	4.534	5.831	22.263	0.970
B	retokenize_tree	4.466	5.847	24.043	0.970
C	rename_tokens	4.809	6.183	23.897	0.967
D	replace_leaf_chars	4.554	5.854	22.660	0.969
E	correct_node_pos	4.546	5.846	23.029	0.970
F	move_punctuation_on_erg_tree	4.538	5.836	22.211	0.970
G	move_determiners_on_erg_tree	4.499	5.807	22.212	0.971
H	flatten_sentential_subtrees	4.717	5.909	22.504	0.970
I	flatten_noun_phrases	5.763	6.999	28.843	0.955
J	move_verb_premodifier_phrases	4.509	5.797	22.212	0.970
K	modified_collapse_unary	6.257	7.197	33.578	0.941
L	add_rb_parent_phrases	4.478	5.760	22.248	0.970
M	add_initial_npsubj_trees	4.604	5.920	22.091	0.968
N	move_prepositional_phrases	4.341	5.308	22.221	0.970
O	correct_tree_node_pos	4.163	5.353	22.981	0.978

Table 5.4: Individual Rule Comparison vs Baseline: Tree Structure Comparisons

Rule IDs Used	All-Tree Node Labels %	Layer-by-Layer Tree Structure %	Subtree %	Average *pq*-Gram Distance
baseline	4.538	5.836	22.211	0.970
ABCDEFGHIJKLMEO	12.556	14.021	55.292	0.883
ABCDEFGHIJKLMNO	11.383	12.911	55.197	0.891
ABCDEFGHIJKLMO	12.536	13.998	54.525	0.888
ABCDEFGHIJNKLMEO	11.503	13.134	56.183	0.887
ABCDEFGJIHKLMEO	12.556	14.021	55.292	0.883
ABCDEFGJIHKLMO	12.536	13.997	54.525	0.888
ABCDEFGJKLMHIEO	12.808	14.317	55.673	0.877
ABCDEFGKHIJLMEO	11.931	13.343	55.212	0.891
ABCDEFGNJKLMHIEO	11.642	13.351	56.513	0.882
ABCDEKFGHIJLMEO	12.543	14.068	55.399	0.884
ABCDEKLFGHIJMEO	12.564	14.091	55.374	0.884
ABCDEKLFGHIJMNEO	11.444	12.787	56.083	0.886
ABCDFGHIJKLM	8.622	9.646	44.723	0.912
ABCDFGHIJKLMEO	8.733	9.774	52.432	0.897
ABCDFGHIJKLMO	8.641	9.669	44.849	0.911
BCDEFGHIJKLMEO	12.563	14.034	54.992	0.883
BCDEFGHIJNKLMEO	11.508	13.148	55.866	0.886
BCDEGHIJKME	8.497	9.509	51.412	0.904
BHGIEEKDJACFOML	11.511	12.811	53.195	0.905
BIGHKCJMDEEAFLO	12.155	13.499	55.490	0.889
BIKHGCJMDEEAFLO	12.480	13.995	55.024	0.886
CDEFGHIJKLMEO	8.669	9.727	40.421	0.905
DEFGHIJKLMEO	8.070	9.078	37.142	0.916

Table 5.5: Combined Rules Comparison vs Baseline: Tree Structure Comparisons

5.4 Additional Sentence Length/Accuracy Comparisons

In order to provide further insight into the transformation process and its results, I have also run several metrics to examine their results against the number and/or length of sentences. Figure 5.1 shows the spread of subtree accuracy across the sentences for several combinations, chosen as having the highest mean subtree accuracies. These graphs show similar narrow distributions with the majority of sentences having 45-55% subtree accuracy.

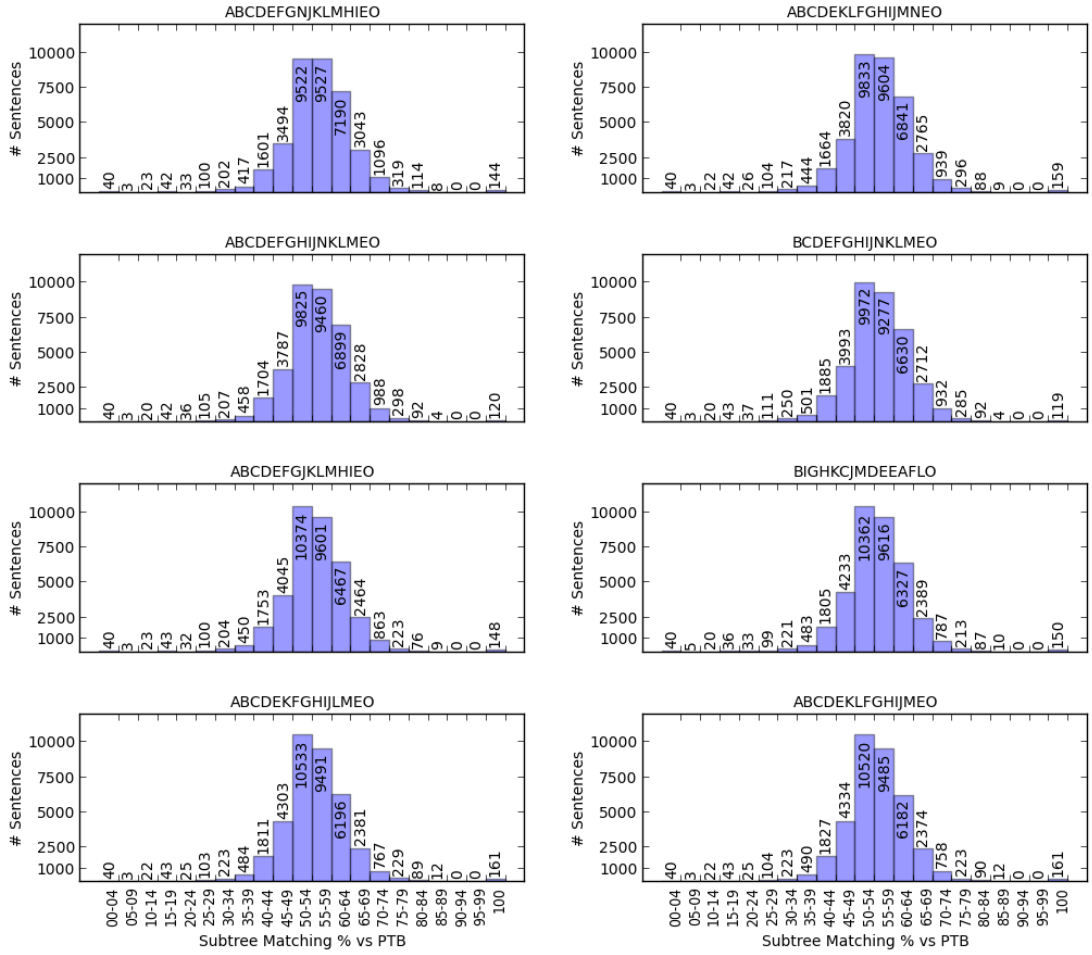

Figure 5.1: Overall Subtree Accuracy Breakdown

In order to show how sentence length affects edit distance, Figure 5.2 shows a scatter plot of the pq-Gram distance multiplied by 100 versus the sentence length. The rule combination chosen here is ABCDEFGHIJKLMO, which performed as well as ABCDEFGJIHKLMO in the DDEC task described in the next chapter and which was slightly ahead of that combination for the combined rule comparisons in Table 5.3 and Table 5.5. To show the iterative effect of rules on edit distance, the edit distance is sampled at several points. The graphs show a general progression towards the altered DeepBank trees becoming more

similar to the PTB trees as the rules are applied, with longer sentences being generally less accurate.

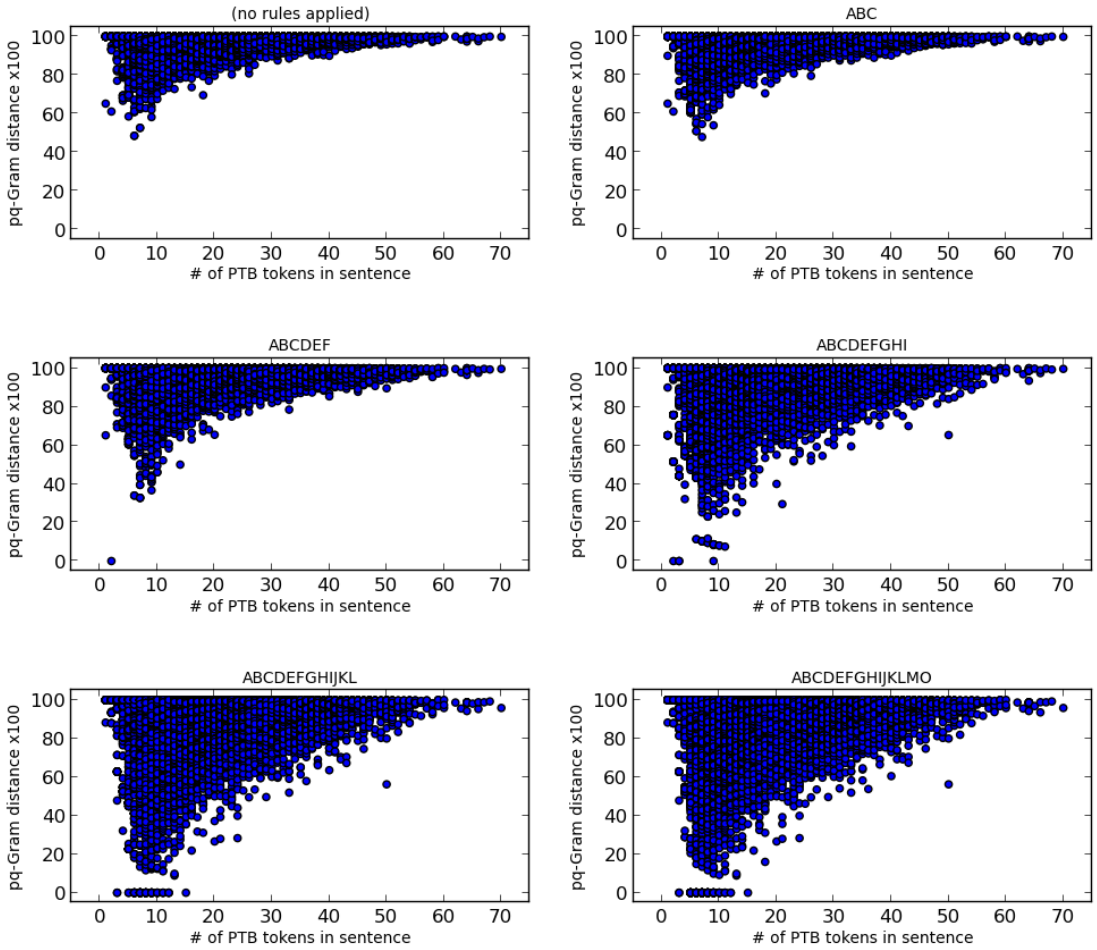

Figure 5.2: Scatter Plot of pq-Gram Distance vs Sentence Length (ABCDEFGHIJKLMO)

5.5 Comparison of Remaining Tree Differences

In order to provide insight into the remaining differences between the altered DeepBank parse trees and the Penn Treebank, I randomly sampled 100 sentences and performed a detailed analysis of where they differed. The rule combination chosen was ABCDEFGHI-JKLMO for the same reasons described above. The differences are discussed below and

grouped by type.

5.5.1 Selection/Representation Differences

Several of the remaining tree differences are due to the PTB and DeepBank selecting different parses for a given sentence, rather than being due to the underlying tree structure before or after transformation. In six cases either the PTB or DeepBank selected a parse where one or more tokens in the sentence was given a label which appears to be simply incorrect. This usually involved modifiers, as in the example shown below where the word *stock* is labeled NN in the altered DeepBank parse but VB in the PTB. Given the context in the rest of the sentence, the PTB parse here appears to be not correct. This particular sentence also had clause structure differences, which are discussed further below.

(1) One group says the futures contribute to stock/NN market volatility; the other contends that futures are a sideshow of speculation that detracts from the stock market's basic function of raising capital. (DeepBank)

(2) One group says the futures contribute to stock/VB market volatility; the other contends that futures are a sideshow of speculation that detracts from the stock market's basic function of raising capital. (PTB)

There are also 43 cases of the DeepBank and PTB having different but valid parses for a given sentence due to label differences. This usually involves determiners or modifiers, such as identifying the word *a* as meaning *one* or *per*. In the example below, both parses appear to be valid interpretations of the phrase *front page*, so neither seems actually wrong but at the same time any rules applied to the DeepBank parse will never result in precisely matching the PTB parse for that phrase because the underlying feature structure for *front* is different. These differences are not correctable in a rule-based manner.

(3) Your Oct. 4 front/NN page/NN noted that British lawyers have to wear wigs in court and that these wigs are made from horses' tails. (DeepBank)

(4) Your Oct. 4 front/JJ page/NN noted that British lawyers have to wear wigs in court and that these wigs are made from horses' tails. (PTB)

Also falling under parse selection is the treatment of titles of books, movies, songs, and other created works. In the PTB these titles are given the function tag -TTL but otherwise parsed normally, whereas in the ERG the parse should have the title words selected as proper nouns. This difference is illustrated in Figure 5.3 and Figure 5.4, and is also not correctable via rules because it would require re-selecting the parse of the sentence in the ERG. Out of the 100 sentences this occurred once.

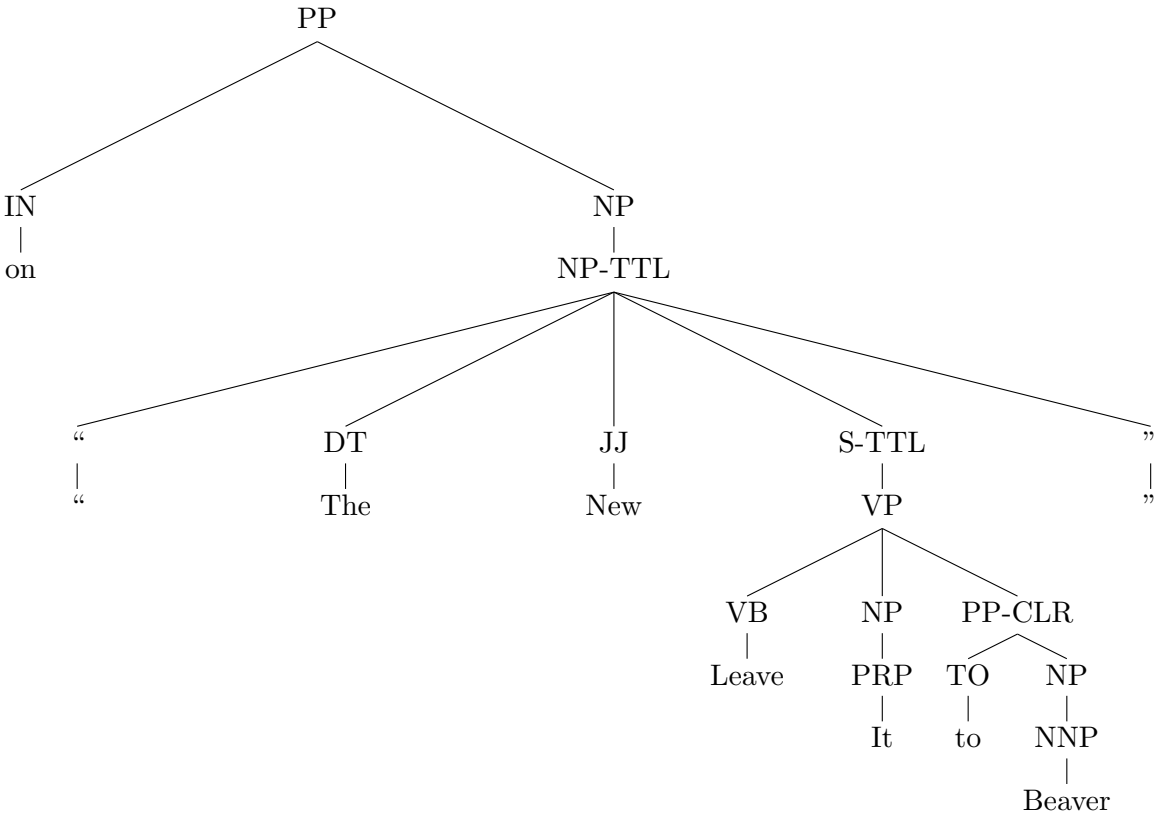

Figure 5.3: PTB Title Example (Part of DeepBank ID 21419013)

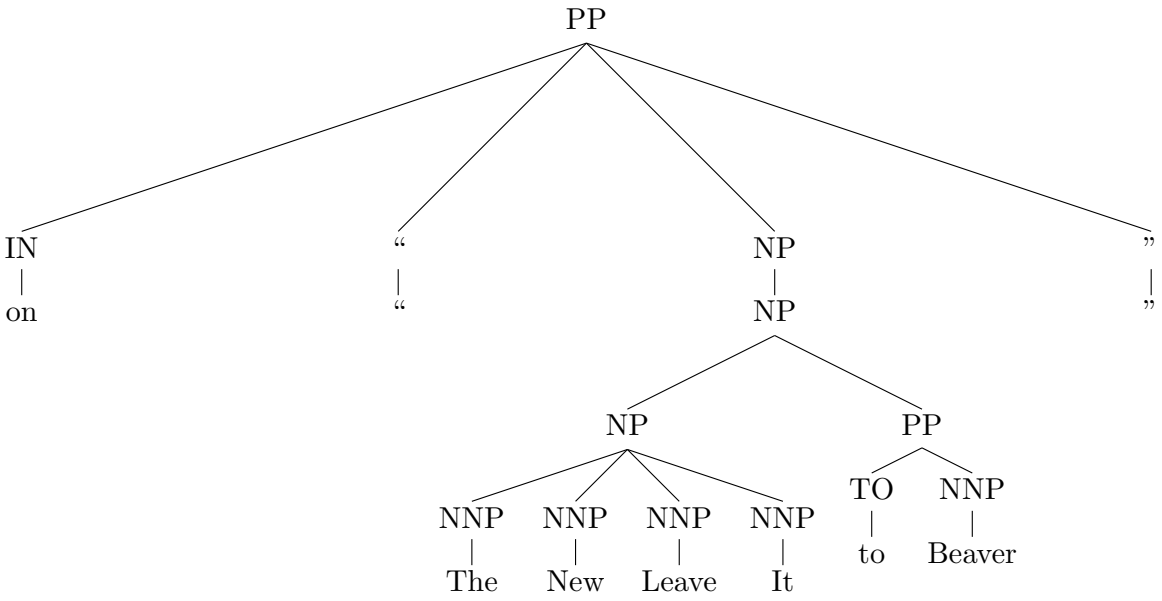

Figure 5.4: DeepBank Title Example (Part of DeepBank ID 21419013)[2]

In addition, the PTB and ERG both have multiple-word phrases which are treated differently from their components. In the ERG this is controlled by the lexicon; phrases like *as much as* are part of the lexicon and have a single type applied to the entire phrase. In the PTB *as much as* and how it should be annotated is strictly laid out in the treebank guidelines. In the sample this occurred 11 times. In order to correctly account for this behavior in the altered DeepBank parses, lexically-specific rules would need to be created to specifically apply the PTB-mandated behavior for each phrase to the tree.

(5) `as/RB much/RB as/RB` (Projected DeepBank)

(6) `as/RB much/RB as/IN` (PTB)

The PTB also draws a distinction between hyphenated multi-word modifiers and non-hyphenated multi-word modifiers. For example, in "*Market-based pollution control may consume some capital that would otherwise purchase state industries.*" the PTB treats *market-based* differently than it would treat *market based* in the same context. This is because it

[2]The parse selection here includes a PP, which seems unlikely to be the desired parse; expected parse selection would have *to Beaver* bracketed as an NP and part of *The New Leave It*.

requires all hyphenated multi-word modifiers to be labeled JJ, and allows non-hyphenated multi-word modifiers to be labeled normally. The ERG follows the same behavior for both the hyphenated and non-hyphenated modifiers as the PTB does for its non-hyphenated modifiers. This results in the parse selection having `(JJ market-based)` in the PTB and `(NN market-) (VBN based)` in the ERG, which becomes `(VBN market-based)` in the altered DeepBank parse after the nodes are combined as part of retokenization. This occurred 8 times in the sample.

5.5.2 Additional Tree Differences

A variety of other differences are due to gaps in the transformation process where tree differences are not accounted for by existing rules, or existing rules are being too broadly or narrowly applied.

In addition to the parse selection differences due to labels, there are also parse selection differences due to phrase structure attachment, as shown below in Figure 5.5 and Figure 5.6; this occurred 28 times. Some instances of this may be correctable by rules. For example, in the case of possessive endings the PTB treats possessives as simple noun modifiers and attaches them at the lowest level possible while the ERG attaches them at the highest level possible. This form of attachment disagreement is easily correctable.

Another gap involves the differences for coordination in the altered DeepBank parses and the PTB, which had 31 examples in the sample. In the PTB, multi-word conjunctions are specifically labeled with CONJP, whereas the DeepBank parses do not contain a phrasal structure specifically associated with the conjunction itself. Additionally, all coordinating conjunctions should be children of the top phrase node containing the coordinating conjunction in the PTB. In the altered DeepBank parses some of these structures are correctly flattened through other rules, but phrases and clauses are usually not altered to adjoin at PTB-appropriate levels of the tree. As a result the structure is usually that of the original ERG parse, with the conjunction as the leftmost daughter of the second conjunct. Examples of phrase coordination with CONJP are in Figure 5.7 and Figure 5.8. Clause coordination behaves similarly, and coordination with single conjunctions replaces the entire CONJP

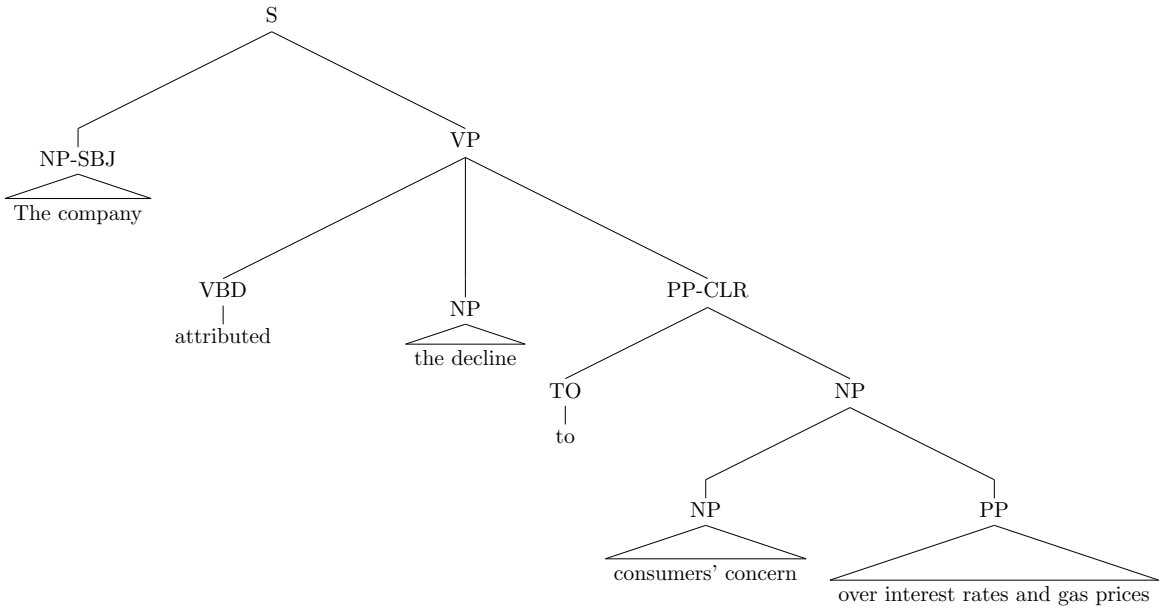

Figure 5.5: PTB Phrase Structure Attachment (Part of DeepBank ID 21463010)

structure with the single conjunction. Single-word conjuncts with single conjunctions are correctly flattened through the flatten noun phrases rule (I).

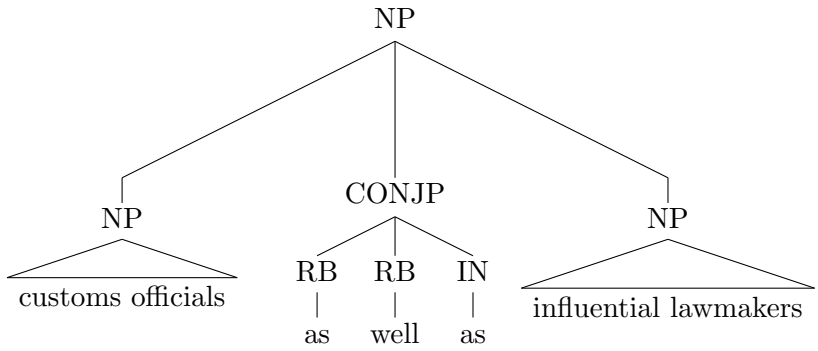

Figure 5.7: PTB Phrase Coordination Example (Part of DeepBank ID 21800011)

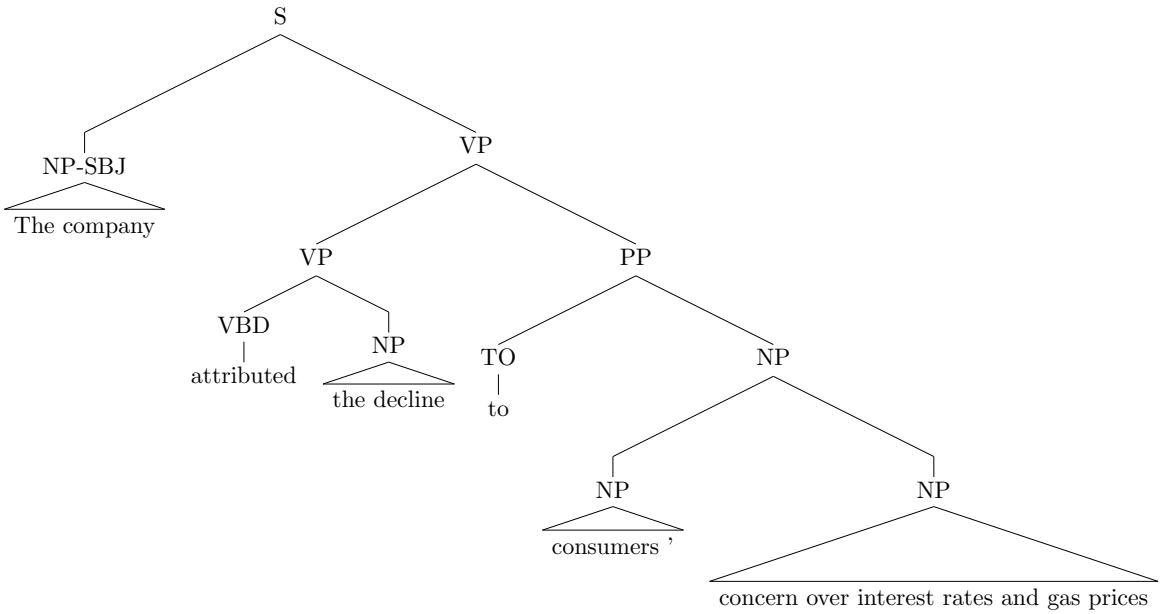

Figure 5.6: Altered DeepBank Phrase Structure Attachment (Part of DeepBank ID 21463010)

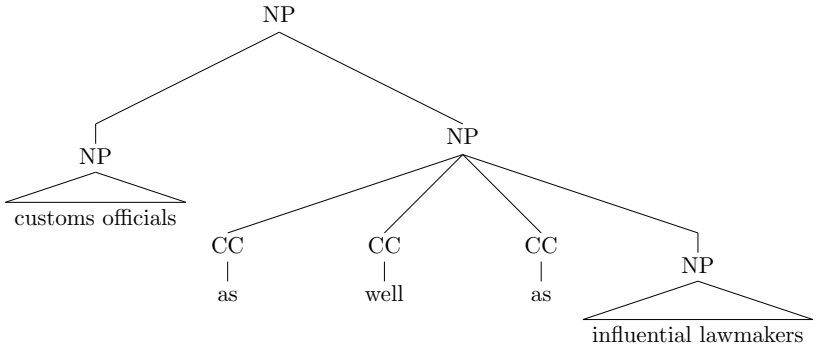

Figure 5.8: DeepBank Phrase Coordination Example (Part of DeepBank ID 21800011)

Another gap occurs because the PTB has several basic differences from the altered DeepBank parses for clauses. This was the most common difference in the sample, occurring 58 times. Many of these differences stem from fundamental differences in how certain clauses are represented in the PTB and ERG. For example, indirect quotes in the PTB are always labeled SBAR and direct quotes are always labeled S. In the ERG, direct and indirect quotes

are treated identically and are both labeled S in the modified DeepBank parses. Examples of an indirect quote parse in the PTB and DeepBank are in Figure 5.9 and Figure 5.10. Infinitives in the PTB are labeled with an S parent wherever they occur in the parse tree; infinitives in the ERG are labeled as verb phrases but not as clauses.

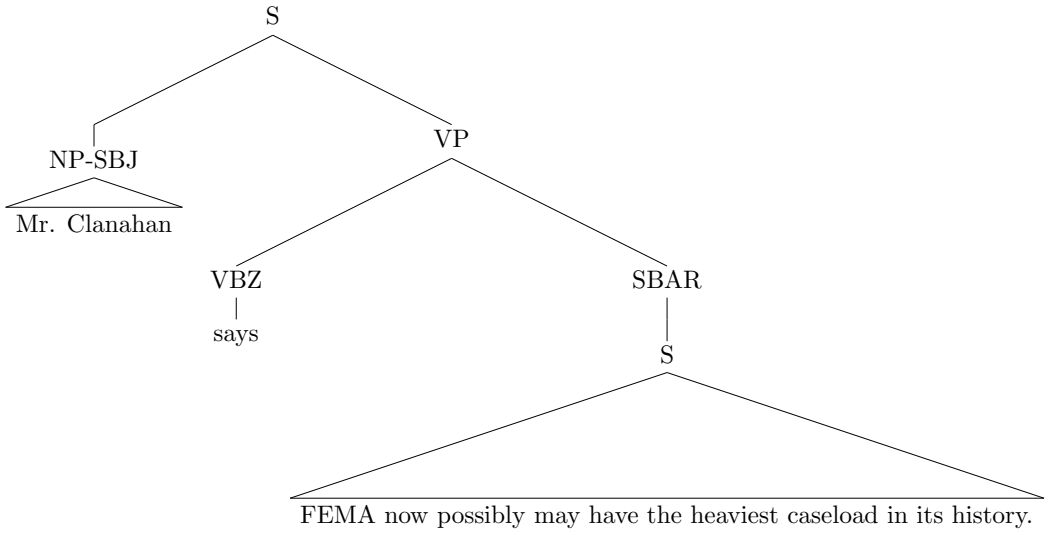

Figure 5.9: PTB Indirect Quote Example (Part of DeepBank ID 21824017)

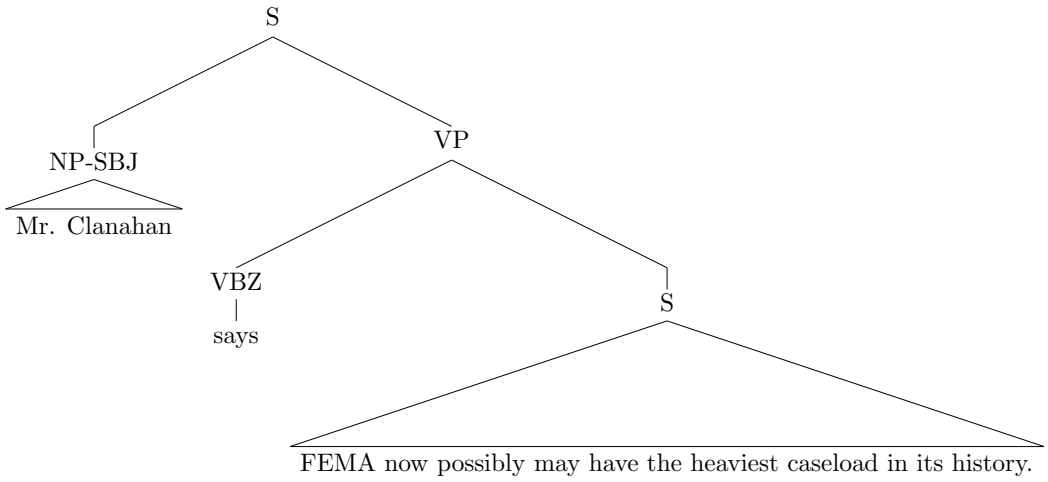

Figure 5.10: DeepBank Indirect Quote Example (Part of DeepBank ID 21824017)

Several current rules can be fine-tuned. For example, in the PTB paired punctuation, such as parentheses or quotes, should adjoin at the same level of the tree and so should the contained phrase. The ERG leaves punctuation attached to words; as a result all punctuation movement in the tree is a result of the transformation rules. The move punctuation rule does not enforce the behavior of paired punctuation because many of the punctuation pairings present in the PTB are not represented in the DeepBank parse tree by any meaningful rule or feature. Punctuation differences appeared in 25 sentences, usually in sentences which also involved coordination.

There is also room for improvement in the altered labels file and the rule which corrects part-of-speech labels. The labels file does not account for many of the PTB function tags and it can also be more fine-tuned in its use of rule name matching for certain phrase labels, like QP. The current version is intended to cover the broad strokes, but there are many instances where distinctions between labels are not being properly made. For example, in the PTB tagging guidelines, *up* should be tagged as RB instead of RP when concerning currency or commodity prices. This distinction is not made in the labels file because it is very sensitive to context. Another example is the label PRN, which in the PTB is left up to the annotator's intuition for application to a given tree; this label is currently excluded from the altered labels file.

5.6 Effects of Remaining Differences on Parser Performance

In terms of parsing, some of the previously described differences have larger effects than others. The selection and representation differences result in fundamentally different—and frequently semantically different—parse trees through the addition of clauses in one format that do not exist in the other or through selecting lexically different tokens. For example, selecting JJ vs NN or the attachment of prepositional phrases to NPs vs VPs in the same sentence. This type of semantically meaningful difference is likely to have the largest impact on parser performance and can also affect what dependencies are produced.

The treatment of phrasal proper nouns like book or movie titles is also likely to have a large impact on the performance of a parser trained on the modified DeepBank trees. The PTB and DeepBank parses label these completely differently – the PTB labels the lowest

parent of these tokens with a function tag denoting that the phrase is a title but otherwise labels and attaches each token normally. The DeepBank parses on the other hand select the proper noun version of each word in the title, making the structure semantically opaque.

The coordination differences are not large as far as what the PTB and DeepBank parses are trying to represent – the tree structure is even not significantly different and can easily be altered via rules. The DeepBank parses and PTB parses both label the conjuncts under a single node and group the words or phrases similarly.

Another difference likely to have a relatively small impact on parsing performance is the treatment of words containing spaces. The PTB separates these words but also often explicitly describes in the guidelines exactly how these words should be tagged, while the ERG keeps these words as a single token which perforce has a single label. The modified DeepBank parses separate the tokens but keep the same label across each token. The underlying goal here is the same between the ERG and PTB; these tokens are treated as a single unit for labeling purposes, even if they do not all share the same label.

5.7 Summary

This chapter laid out the basic metrics used for comparing the effectiveness of rules individually and in combination, and described the remaining differences between the modified DeepBank and the PTB. The metrics can be divided into those specifically examining tokenization and the leaf node labels, and those examining the full tree structure and its labels. Additional tests were run with the full tree structure compared against the number and length of the sentences. The resulting numbers show room for improvement in the set of rules for most of the metrics. The discussion of the remaining differences between the modified trees and the PTB provides insight into where the largest gaps are for future rules and how existing rules can be improved.

Chapter 6

COMPARISONS WITH THE PTB, DEEPBANK, AND TRANSFORMATIONS

This chapter discusses the comparisons done with the Stanford parser and tests on its dependencies output. It then goes on to examine how the remaining tree differences described in the previous chapter affect the parser results and where improvements can be made.

6.1 DDEC Evaluation Overview

As previously discussed, the DDEC consists of 100 example sentences each of ten linguistic phenomena. Each instance of a phenomenon is represented by a dependency triple identifying the heads and dependents along with their respective word positions in the sentence. Some sentences contained more than one instance of a phenomenon, resulting in a total of 2127 dependency triples in the gold standard. Of these, 580 have more than one correct interpretation and 253 are negative dependencies. The DDEC evaluation script examines the output from the parser for each of the dependency triples with a series of regular expressions which describe what the dependency output looks like when the parser correctly identifies a phenomenon. It then outputs the number of positive, negative, and total target dependencies found by the parser.

6.2 Penn Treebank Comparisons

Several comparisons were performed involving the Penn Treebank before examining the variously PTB-like DeepBank outputs. The Stanford parser was trained on sections 02-21 of the Penn Treebank and then tested on the DDEC sentences. This output was then evaluated against the DDEC gold standard to produce dependency accuracy results. This produced slightly different numbers from those in Bender et al. (3), possibly due to different versions of the Stanford parser. The same procedure was also performed with only those sentences

of the Penn Treebank which also occurred in the DeepBank. The latter results provide a comparison point for results from a PTB-like DeepBank output if it were completely consistent with the Penn Treebank, including the occasional variabilities in tree structure and label assignment.

Description	Correct Total Targets	Correct Total Targets %	Correct Positive Targets	Correct Positive Targets %	Correct Negative Targets	Correct Negative Targets %
Maximum	2127	100%	1874	100%	253	100%
PTB 00-22, 24	1114	52.374	986	52.614	128	50.592
PTB 02-21	1105	51.951	976	52.081	129	50.988
PTB (DeepBank-selected)	1095	51.480	968	51.654	127	50.197

Table 6.1: Baseline PTB-related Results

6.3 Exported DeepBank Parse Comparisons vs PTB

One set of results for this thesis involves comparing combinations of transformation rules versus the DDEC sentence set. The DeepBank selected parse exported outputs are run through each set of ordered rules and the Stanford parser is trained on the final outputs. The trained parser is then tested against the DDEC sentences and compared against the DDEC gold standard. As previously noted in Chapter 4, certain rule combinations are not allowed unless the results are also passed through the rule enforcing that word nodes' labels are converted to phrasal node labels in phrasal positions.

These results show that the modified ERG-produced parses fall short of the overall and positive-target DDEC numbers produced by the PTB. They improve on the negative target numbers from the PTB results, however this largely appears to be due to the parser trained on the modified DeepBank output identifying fewer dependencies overall. The Stanford parser proved very strict in its determination of valid training inputs; two types of errors appear during training: a head rule error and an array index out of bounds error. With many instances of these errors, parser models were still generated for testing purposes but

upon testing no results were produced. Marneffe et al. (27) describe how the dependencies are produced; tregex patterns are applied to the phrase-structure trees. These patterns are clearly very tightly dependent on the PTB structure and are not tolerant of much deviation. Additionally there appears to be little correlation between improvement on any of the metrics used for rule evaluation laid out in Chapter 5 and the DDEC results laid out above.

It was hoped that the DeepBank-exported trees would allow for higher DDEC numbers than those of the PTB, because the ERG allows for consistently and reliably producing complex structures. The complexity allows for fine-grained label assignment while the consistency and reliability carry over into the application of those labels across different parse trees. The transformed DeepBank-exported trees do allow for retraining the constituency parser portion of the Stanford parser; the dependency extraction portion however is rule-based. These rules are static and do not adapt automatically, and are particularly adapted to the Penn Treebank and its occasional quirks. As a result only a limited set of the rule combinations could be trained and then tested against the DDEC; the remaining combinations failed during dependency generation.

Rule Combination	Correct Total Targets	Correct Total Targets %	Correct Positive Targets	Correct Positive Targets %	Correct Negative Targets	Correct Negative Targets %
Maximum	2127	100%	1874	100%	253	100%
PTB (DeepBank-selected)	1095	51.480	968	51.654	127	50.197
ABCDEFGHIJKLMEO	613	28.820	432	23.052	181	71.542
ABCDEFGHIJKLMNO	677	31.829	505	26.948	172	67.984
ABCDEFGHIJKLMO	679	31.923	496	26.467	183	72.332
ABCDEFGHIJNKLMEO	615	28.914	437	23.319	178	70.356
ABCDEFGJIHKLMEO	614	28.867	433	23.106	181	71.542
ABCDEFGJIHKLMO	679	31.923	496	26.467	183	72.332
ABCDEFGJKLMHIEO*	‡	‡	‡	‡	‡	‡
ABCDEFGKHIJLMEO	605	28.444	423	22.572	182	71.937
ABCDEFGNJKLMHIEO*	†	†	†	†	†	†
ABCDEKFGHIJLMEO	573	26.939	412	21.985	161	63.636
ABCDEKLFGHIJMEO	581	27.315	419	22.359	162	64.032
ABCDEKLFGHIJMNEO	587	27.598	419	22.359	168	66.403
ABCDFGHIJKLM*	†	†	†	†	†	†
ABCDFGHIJKLMEO	590	27.739	418	22.305	172	67.984
ABCDFGHIJKLMO	661	31.077	481	25.667	180	71.146
BCDEFGHIJKLMEO*	†	†	†	†	†	†
BCDEFGHIJNKLMEO*	†	†	†	†	†	†
BCDEGHIJKME*	†	†	†	†	†	†
BHGIEEKDJACFOML	‡	‡	‡	‡	‡	‡
BIGHKCJMDEEAFLO*	†	†	†	†	†	†
BIKHGCJMDEEAFLO*	†	†	†	†	†	†
CDEFGHIJKLMEO*	†	†	†	†	†	†
DEFGHIJKLMEO*	†	†	†	†	†	†

†= Failed Build - head rule error ‡= Failed Build - array index out of bounds
* = additionally failed to produce serialized output

Table 6.2: PTB-like DeepBank DDEC Results

Chapter 7

CONCLUSION

In summary, I found that the modified ERG-produced parses from the DeepBank did not perform as well as the PTB with the DDEC. In large part this is due to the combination of the ERG-produced parses not entirely matching the PTB structure and the Stanford dependencies generation being very tightly coupled to the PTB structure. I also described in detail the remaining differences between the modified DeepBank parses and the Penn Treebank, providing insight for future work in this area.

7.1 Usability and Limitations

The comparisons between the DeepBank and Penn Treebank are useful for anyone seeking to look into their differences or transform into or out of either of their formats.

At present the dependencies comparison work is limited to the Penn Treebank and the Stanford parser. The Stanford parser proved brittle towards anything not very close to the PTB format for training purposes. Enforcing PTB restrictions on combinations of word-level and phrase-level node labels by overriding tree node labels somewhat mitigated the parser's brittleness.

An additional limitation is that the ERG is not able to parse all of the PTB sentences in the DeepBank. With increasing versions of the ERG this limitation should decrease; the change from 1111 to 1212 with the DeepBank already showed noticeable increases in coverage.

7.2 Future Paths

Reducing the restrictiveness of the Stanford dependencies code would allow for more successful builds when given inputs which do not closely match the PTB format. This may or may not allow for better results with the DDEC. Alternately, the test versus the DDEC could

be run with a parser which generates dependencies through a more data-driven method. Additional non-dependency-related tests could also be run.

Continuing against the Stanford parser and DDEC, there is still room for improvement or tweaking to move closer to PTB structure and labeling from the transformations described in this thesis. While the transformations are intended to provide a large degree of insight into their effect on dependency parsing, they are not intended to be exhaustive. Further approaching 100% compliance with the PTB format while still taking advantage of the consistency provided by the ERG analyses may result in higher accuracy than presented here.

Future versions of the ERG are expected to introduce new features and increase parse coverage. While most of these modifications are likely to allow the code produced as part of this thesis to run as-is, it is possible that some changes will require modifications to the transformation rules and parse-nodes.tdl changes.

The general structure of the parsing and tree modification rules is intended to be generic as pertains to corpora and parsers. The portions that require modification for use with other grammars or treebanks are the initial parsing code to get outputs into tree formats and the final output code to export the transformed trees into the format required by the parser. Further development on the code described herein is planned for the near future and any new results will be posted on that same site[1].

[1] `https://bitbucket.org/caelum/thesis-transform`

BIBLIOGRAPHY

[red]

[1] N. Augsten, M. Böhlen, and J. Gamper. Approximate Matching of Hierarchical Data Using pq-Grams. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 301–312. VLDB Endowment, 2005.

[2] T. Baldwin, E. M. Bender, D. Flickinger, A. Kim, and S. Oepen. Road-testing the English Resource Grammar Over the British National Corpus. In *LREC*, 2004.

[3] E. M. Bender, D. Flickinger, S. Oepen, and Y. Zhang. Parser Evaluation Over Local and Non-local Deep Dependencies in a Large Corpus. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 397–408, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. ISBN 978-1-937284-11-4. URL `http://dl.acm.org/citation.cfm?id=2145432.2145479`.

[4] A. Bies, M. Ferguson, K. Katz, R. MacIntyre, V. Tredinnick, G. Kim, M. Marcinkiewicz, and B. Schasberger. Bracketing Guidelines for Treebank II Style Penn Treebank Project. *University of Pennsylvania*, 1995.

[5] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.

[6] E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, Liberman M., M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In *Speech and natural language: proceedings of a workshop, held at Pacific Grove, California, February 19-22, 1991*, page 306. Morgan Kaufmann Pub, 1991.

[7] U. Callmeier. *Efficient Parsing with Large-scale Unification Grammars*. PhD thesis, Masters thesis, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[8] David Carter. The TreeBanker. A Tool for Supervised Training of Parsed Corpora. In *Proceedings of the workshop on computational environments for grammar development and linguistic engineering*, 1997.

[9] A. Copestake. Definitions of Typed Feature Structures. In Stephan Oepen, Dan Flickinger, Jun-ichi Tsujii, and Hans Uszkoreit, editors, *Collaborative Language Engineering*, pages 227–230. CSLI Publications, Stanford, CA, 2002.

[10] A. Copestake. *Implementing Typed Feature Structure Grammars*, volume 110. CSLI publications Stanford, CA, 2002.

[11] A. Copestake and D. Flickinger. An Open-Source Grammar Development Environment and Broad-Coverage English Grammar Using HPSG. In *Proceedings of LREC*, volume 2, 2000.

[12] A. Copestake, D. Flickinger, C. Pollard, and I.A. Sag. Minimal Recursion Semantics: An Introduction. *Research on Language & Computation*, 3(2):281–332, 2005.

[13] DELPH-IN. DELPH-IN, May 2013. URL `http://www.delph-in.net`.

[14] M. Dickinson and W.D. Meurers. Prune Diseased Branches to Get Healthy Trees! How to Find Erroneous Local Trees in a Treebank and Why It Matters. In *In Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT 2005*. Citeseer, 2005.

[15] R. Dridan and S. Oepen. Tokenization: Returning to a Long Solved Problem a Survey, Contrastive Experiment, Recommendations, and Toolkit. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, ACL '12, pages 378–382, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=2390665.2390750`.

[16] D. Flickinger. On Building a More Efficient Grammar by Exploiting Types. *Natural Language Engineering*, 6 (1):15 – 28, 2000.

[17] D. Flickinger. Accuracy vs. Robustness in Grammar Engineering. *Language from a Cognitive Perspective: Grammar Usage, and Processing. Stanford*, pages 31–50, 2011.

[18] D. Flickinger, V. Kordoni, and Y. Zhang. DeepBank: A Dynamically Annotated Treebank of the Wall Street Journal. In *In Proceedings of the 11th International Workshop on Treebanks and Linguistic Theories*, 2012.

[19] R. Gabbard, S. Kulick, and M. Marcus. Fully Parsing the Penn Treebank. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 184–191, 2006.

[20] D. Gildea. Corpus Variation and Parser Performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*, pages 167–202, 2001.

[21] P. Kingsbury and M. Palmer. From Treebank to Propbank. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC-2002)*, pages 1989–1993. Citeseer, 2002.

[22] D. Klein and C. Manning. Accurate Unlexicalized Parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[23] A. Koller and S. Thater. Efficient Solving and Exploration of Scope Ambiguities. In *Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*, pages 9–12. Association for Computational Linguistics, 2005.

[24] V. Kordoni and Y. Zhang. Disambiguating Compound Nouns for a Dynamic HPSG Treebank of Wall Street Journal Texts. In *Proceedings of the 7th international conference on language resources and evaluation, Valetta, Malta*, 2010.

[25] M. Marcus, G. Kim, M.A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. The Penn Treebank: Annotating Predicate Argument Structure.

In *Proceedings of the workshop on Human Language Technology*, pages 114–119. Association for Computational Linguistics, 1994.

[26] M.P. Marcus, M.A. Marcinkiewicz, and B. Santorini. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.

[27] M. Marneffe, B. Maccartney, and C. Manning. Generating Typed Dependency Parses From Phrase Structure Parses. In *In LREC 2006*, 2006.

[28] G.A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[29] Y. Miyao, T. Ninomiya, and J. Tsujii. Corpus-Oriented Grammar Development for Acquiring a Head-driven Phrase Structure Grammar from the Penn Treebank. *Natural Language Processing–IJCNLP 2004*, pages 684–693, 2005.

[30] S. Oepen and D. Flickinger. Towards Systematic Grammar Profiling. Test Suite Technology Ten Years After. *Journal of Computer Speech and Language*, 12(4):411–436, 1998.

[31] S. Oepen, D. Flickinger, K. Toutanova, and C.D. Manning. LinGO Redwoods. *Research on Language & Computation*, 2(4):575–596, 2004.

[32] S. Oepen, E. Velldal, J. Lnning, P. Meurer, V. Rosn, and D. Flickinger. Towards Hybrid Quality-Oriented Machine Translation. On Linguistics and Probabilities in MT. In *TMI:07*, Skvde, Sweden, 2007.

[33] C. Pollard and I.A. Sag. *Head-driven Phrase Structure Grammar*. University of Chicago Press, 1994.

[34] I.A. Sag, T. Wasow, and E.M. Bender. *Syntactic Theory: A Formal Introduction*, volume 152. Center for the Study of Language and Information, 2nd edition, 2003.

[35] B. Santorini. Part-of-speech Tagging Guidelines for the Penn Treebank Project (3rd revision). 1990.

[36] T. Tanaka, F. Bond, S. Oepen, and S. Fujita. High Precision Treebanking-Blazing Useful Trees Using POS Information. In *Annual Meeting - Association For Computational Linguistics*, volume 43, page 330, 2005.

[37] A. Taylor, M. Marcus, and B. Santorini. The Penn Treebank: An Overview. In *Treebanks*, pages 5–22. Springer, 2003.

[38] Gisle Ytrestøl, Stephan Oepen, and Daniel Flickinger. Extracting and Annotating Wikipedia Sub-domains. In *Proceedings of the 7th International Workshop on Treebanks and Linguistic Theories*, pages 185–197, 2009.

[39] Y. Zhang and V. Kordoni. Discriminant Ranking for Efficient Treebanking. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 1453–1461. Association for Computational Linguistics, 2010.

[40] Y. Zhang and H.U. Krieger. Large-scale Corpus-driven PCFG Approximation of an HPSG. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 198–208. Association for Computational Linguistics, 2011.

Appendix A

CODE FOR TREE TRANSFORMATION RULES

All of the code produced here is Python; version 2.7 is what was run to produce the results.

A.1 Rule A: split_multiword_leaves

```
def split_multiword_leaves(original_tree):
  leaf_positions = original_tree.treepositions(order='leaves')
  preterminals = set([x[:-1] for x in leaf_positions])
  nodes_with_leaves = list()
  for tree_pos in reversed(sorted(preterminals)):
    node_to_copy = original_tree[tree_pos]
    children = node_to_copy[0:]
    num_children = len(children)
    if num_children > 1:
      logging.debug('children are %s', children)
      remaining_leaves = list(children)
      first_leaf = remaining_leaves.pop(0)
      node_to_copy[0:] = [first_leaf]
      while remaining_leaves:
        new_leaf = remaining_leaves.pop()
        new_node = node_to_copy.copy(deep=True)
        new_node.node = deepcopy(node_to_copy.node)
        new_node[0:] = [new_leaf]
        node_to_copy.add_right_sibling(new_node)
```

A.2 Rule B: retokenize_tree

```
def retokenize_tree(original_tree, repp_tokens):
  lower_repp_tokens = [x.lower() for x in repp_tokens]
  logging.debug('repp tokens are %s', lower_repp_tokens)
  erg_tokens = original_tree.leaves()
  lower_erg_tokens = [x.lower() for x in erg_tokens]
  max_tries = 50
```

```
  num_tries = 0
  while ((lower_repp_tokens != lower_erg_tokens) and
         (num_tries <= max_tries)):
    num_tries += 1
    erg_indices = range(len(lower_erg_tokens))
    repp_indices = range(len(lower_repp_tokens))
    for idx, (erg_idx, repp_idx) in enumerate(map(None, erg_indices, repp_indices)):
      if erg_idx is None:
        last_erg_pos = original_tree.leaf_treeposition(idx-1)
        node_to_copy = original_tree[last_erg_pos[:-1]]
        new_node = node_to_copy.copy(deep=True)
        new_node.node = deepcopy(node_to_copy.node)
        new_node[0:] = [repp_tokens[repp_idx]]
        node_to_copy.add_right_sibling(new_node)
        erg_tokens = original_tree.leaves()
        lower_erg_tokens = [x.lower() for x in erg_tokens]
        break
      repp_token = lower_repp_tokens[repp_idx]
      original_erg_token = lower_erg_tokens[erg_idx]
      erg_token = reconcile_characters(repp_token, original_erg_token)
      if original_erg_token != erg_token:
        current_pos = original_tree.leaf_treeposition(idx)
        current_node_and_sibs = original_tree[current_pos[:-1]][0:]
        # replace current node with character-reconciled one
        new_node_and_sibs = list()
        for current_sib in current_node_and_sibs:
          if current_sib == original_erg_token:
            new_node_and_sibs.append(erg_token)
          else:
            new_node_and_sibs.append(current_sib)
        original_tree[current_pos[:-1]][0:] = new_node_and_sibs
        erg_tokens = original_tree.leaves()
        lower_erg_tokens = [x.lower() for x in erg_tokens]
        erg_token = lower_erg_tokens[idx]

      if erg_token != repp_token:
        len_erg = len(erg_token)
        len_repp = len(repp_token)

        if len_erg > len_repp:
```

```
  if erg_token[:len_repp] == repp_token:
    left_word = erg_token[:len_repp]
    right_word = erg_token[len_repp:]
    erg_pos = original_tree.leaf_treeposition(idx)
    nth_child_to_edit = erg_pos[-1]
    node_to_copy = original_tree[erg_pos[:-1]]
    new_node = node_to_copy.copy(deep=True)
    new_node.node = deepcopy(node_to_copy.node)
    new_node[0:] = [right_word]
    node_to_copy.add_right_sibling(new_node)
    node_to_copy[nth_child_to_edit] = left_word

    erg_tokens = original_tree.leaves()
    lower_erg_tokens = [x.lower() for x in erg_tokens]
    break
else:
  if repp_token[:len_erg] == erg_token:
    initial_start_idx = idx
    start_idx = initial_start_idx
    initial_end_idx = idx + 1
    end_idx = initial_end_idx
    combined_words = ''.join([lower_erg_tokens[initial_start_idx],
                              lower_erg_tokens[initial_end_idx]])

    start_pos = original_tree.leaf_treeposition(initial_start_idx)
    end_pos = original_tree.leaf_treeposition(initial_end_idx)
    same_parent = bool(start_pos[:-1] == end_pos[:-1])

    start_node_and_sibs = original_tree[start_pos[:-1]][0:]
    if len(start_node_and_sibs) > 1:
      # really want leftmost sibling of this node
      if same_parent:
        nth_idx_of_child = start_pos[-1]
        if nth_idx_of_child != 0:
          # if start pos is not child idx 0, make it so
          extra_start = nth_idx_of_child
          start_idx = initial_start - extra_start
          additional_start_words = \
            lower_erg_tokens[start_idx:initial_start_idx]
          additional_start_words.append(combined_words)
```

```
                combined_words = ''.join(additional_start_words)
              else:
                extra_start = len(start_node_and_sibs) - 1
                start_idx = initial_start_idx - extra_start
                additional_start_words = \
                  lower_erg_tokens[start_idx:initial_start_idx]
                additional_start_words.append(combined_words)
                combined_words = ' '.join(additional_start_words)

            end_node_and_sibs = original_tree[end_pos[:-1]][0:]
            if len(end_node_and_sibs) > 1:
              if same_parent:
                nth_idx_of_child = end_pos[-1]
                max_end = len(end_node_and_sibs) - 1
                if nth_idx_of_child != max_end:
                  extra_end = max_end - nth_idx_of_child
                  end_idx = initial_end_idx + extra_end
                  additional_end_words = \
                    lower_erg_tokens[initial_end_idx+1:end_idx+1]
                  additional_end_words.insert(0, combined_words)
                  combined_words = ''.join(additional_end_words)
              else:
                extra_end = len(end_node_and_sibs) - 1
                end_idx = initial_end_idx + extra_end
                additional_end_words = \
                  lower_erg_tokens[initial_end_idx+1:end_idx+1]
                additional_end_words.insert(0, combined_words)
                combined_words = ' '.join(additional_end_words)

            original_tree = \
              _recombine_nodes(original_tree, start_idx, end_idx, combined_words)
            erg_tokens = original_tree.leaves()
            lower_erg_tokens = [x.lower() for x in erg_tokens]
            break
    if lower_repp_tokens != lower_erg_tokens:
      logging.error('Unable to reconcile tokenization!')
      logging.debug('ERG tokens %s', erg_tokens)
      logging.debug('REPP tokens %s', repp_tokens)
```

A.3 Rule C: rename_tokens

```
def rename_tokens(original_tree, safechar_repp_words):
  leaf_positions = original_tree.treepositions(order='leaves')
  for tree_pos, new_word in map(None, leaf_positions, safechar_repp_words):
    if tree_pos != None and new_word != None:
      tree_word = original_tree[tree_pos]
      if tree_word.lower() == new_word.lower():
        original_tree[tree_pos] = new_word
```

A.4 Rule D: replace_leaf_chars

```
def replace_leaf_chars(original_tree):
  left_replacement_chars = [
    ('"', "``"),
  ]
  right_replacement_chars = [
    ('"', "''"),
  ]
  all_replacement_chars = [
    ('\xe2\x80\x9c', "``"),
    ('\xe2\x80\x9d', "''"),
    ('\xe2\x80\x93', '--'),
    ('\xe2\x80\x93', '--'),
    ('\xe2\x80\x99', "'"),
    ('\xe2\x80\x98', '`'),
    ('\xe2\x80\xa6', '...'),
    ('/', '\\/'),
    ('(', '-LRB-'),
    (')', '-RRB-'),
    ('[', '-LSB-'),
    (']', '-RSB-'),
    ('{', '-LCB-'),
    ('}', '-RCB-'),
  ]
  positions = original_tree.treepositions(order='leaves')
  for leaf_pos in positions:
    word = original_tree[leaf_pos]
    for (char_to_replace, replacement_char) in all_replacement_chars:
      word = re.sub(re.escape(char_to_replace), replacement_char, word)
    for (char_to_replace, replacement_char) in left_replacement_chars:
```

```
      if word[0] == char_to_replace:
        word[0] = replacement_char
    for (char_to_replace, replacement_char) in right_replacement_chars:
      if word[-1] == char_to_replace:
        word[-1] = replacement_char
    original_tree[leaf_pos] = word
```

A.5 Rule E: correct_node_pos

```
def correct_node_pos(original_tree):
  positions = original_tree.treepositions(order='leaves')
  for leaf_pos in positions:
    word = original_tree[leaf_pos]
    original_pos = original_tree[leaf_pos[:-1]].node['pos']
    token_is_own_pos = (word in HAS_OWN_POS_TOKENS
                        or word in SPLIT_ON_LAST_WORD_TOKENS)
    if token_is_own_pos:
      #initial round; replacement below also possible
      original_tree[leaf_pos[:-1]].node['pos'] = word
    #overrides
    if (word.endswith('ing')
        and original_pos in ['VP', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']):
      original_tree[leaf_pos[:-1]].node['pos'] = 'VBG'
    elif word in ['--', ';', ':', '...']:
      original_tree[leaf_pos[:-1]].node['pos'] = ':'
    elif word == 'U.S.':
      original_tree[leaf_pos[:-1]].node['pos'] = 'NNP'
    elif word == '`' and original_pos != 'POS':
      original_tree[leaf_pos[:-1]].node['pos'] = '``'
    elif word == "'" and original_pos != 'POS':
      original_tree[leaf_pos[:-1]].node['pos'] = "''"
    elif word == '%':
      original_tree[leaf_pos[:-1]].node['pos'] = 'NN'
    elif word == "n't":
      original_tree[leaf_pos[:-1]].node['pos'] = 'RB'
    elif word == 'to':
      original_tree[leaf_pos[:-1]].node['pos'] = 'TO'
    elif word in ['more', 'less']:
      original_tree[leaf_pos[:-1]].node['pos'] = 'RBR'
    elif word == 'which':
```

```
      original_tree[leaf_pos[:-1]].node['pos'] = 'WDT'
    elif word == '&':
      original_tree[leaf_pos[:-1]].node['pos'] = 'CC'
    elif word == '-LRB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-LRB-'
    elif word == '-RRB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-RRB-'
    elif word == '-LSB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-LSB-'
    elif word == '-RSB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-RSB-'
    elif word == '-LCB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-LCB-'
    elif word == '-RCB-':
      original_tree[leaf_pos[:-1]].node['pos'] = '-RCB-'
```

A.6 Rule F: move_punctuation_on_erg_tree

```
def move_punctuation_on_erg_tree(erg_tree):
  # find the other punctuation nodes which should move; punctuation should move to
  # the left or right outside of the phrase structure containing it
  leaves = erg_tree.leaves()
  leaf_positions = erg_tree.treepositions(order='leaves')
  for leaf_idx in reversed(range(len(leaves))):
    leaf_position = leaf_positions[leaf_idx]
    leaf = erg_tree[leaf_position]
    leaf_parent = erg_tree[leaf_position[:-1]]
    leaf_pos = leaf_parent.node['pos']
    if leaf_pos in ['``', "''", '.', '?', '!', ',', ':']:
      current_position = leaf_position[:-1]
      # expects leaf position
      ancestor_as_leftmost = \
        find_highest_tree_with_node_as_leftmost(erg_tree, leaf_position)
      ancestor_as_rightmost = \
        find_highest_tree_with_node_as_rightmost(erg_tree, leaf_position)
      if ancestor_as_leftmost != current_position:
        move_to_left_of_given_ancestor(erg_tree, current_position,
                                       ancestor_as_leftmost)
      elif ancestor_as_rightmost != current_position:
        move_to_right_of_given_ancestor(erg_tree, current_position,
```

```
                                        ancestor_as_rightmost)
      leaf_positions = erg_tree.treepositions(order='leaves')
      current_position = leaf_positions[leaf_idx][:-1]
```

A.7 Rule G: move_determiners_on_erg_tree

```
def move_determiners_on_erg_tree(erg_tree):
  # find the other punctuation nodes which should move; punctuation should move to
  # the left or right outside of the phrase structure containing it
  leaves = erg_tree.leaves()
  leaf_positions = erg_tree.treepositions(order='leaves')
  for leaf_idx in reversed(range(len(leaves))):
    leaf_position = leaf_positions[leaf_idx]
    leaf = erg_tree[leaf_position]
    leaf_parent = erg_tree[leaf_position[:-1]]
    leaf_pos = leaf_parent.node['pos']
    if leaf_pos in ['DT']:
      current_position = leaf_position[:-1]
      if (erg_tree[current_position].right_sibling
          and isinstance(erg_tree[current_position].right_sibling, Tree)
          and erg_tree[current_position].right_sibling.node['pos']
            in ['NNP', 'NN', 'NNS', 'NNP', 'NNPS', 'NP', 'NP-SBJ']
          and isinstance(erg_tree[current_position].right_sibling[0], Tree)):
        move_to_be_leftmost_child_of_right_sibling(erg_tree, current_position)
        leaf_positions = erg_tree.treepositions(order='leaves')
        current_position = leaf_positions[leaf_idx][:-1]
```

A.8 Rule H: flatten_sentential_phrases

```
def flatten_sentential_phrases(erg_tree):
  def has_s_child(node):
    for child_idx, child in enumerate(node[0:]):
      if (isinstance(child, Tree)
          and child.node['pos'] in ['S', 'SBAR', 'SBARQ', 'SINV', 'SQ']):
        return child_idx
    return -1

  # find lowest non-branching node
  current_node = erg_tree
  while len(current_node[0:]) == 1:
    if not isinstance(current_node[0], Tree):
```

```
      break
    current_node = current_node[0]

  while has_s_child(current_node) != -1:
    child_idx = has_s_child(current_node)
    s_node = current_node[child_idx]
    remove_parent_and_move_children_up(erg_tree, s_node.treeposition)
```

A.9 Rule I: flatten_noun_phrases

```
def flatten_noun_phrases(erg_tree):
  nouny_tags = ['CC', 'CD', 'DT', 'JJ', 'JJR', 'JJS', 'NN', 'NNS', 'NNP', 'NNPS',
                ',', 'NP', 'NP-SBJ', 'RB', 'RBR', 'RBS']
  # go from the top
  all_positions = sorted(erg_tree.treepositions(order='preorder'), key=len)
  leaf_positions = erg_tree.treepositions(order='leaves')
  positions_to_examine = strip_preterminals_from_positions(all_positions, leaf_positions)
  while positions_to_examine:
    position_to_check = positions_to_examine.pop()
    position_pos = erg_tree[position_to_check].node['pos']
    if position_pos in ['NP', 'NP-SBJ']:
      all_nouny = subtree_only_contains_given_tags(erg_tree, position_to_check, nouny_tags)
      if all_nouny:
        flatten_subtree(erg_tree, position_to_check)
        positions_to_examine = \
          strip_subtrees_from_positions(positions_to_examine, position_to_check)
```

A.10 Rule J: move_verb_premodifier_phrases

```
def move_verb_premodifier_phrases(erg_tree):
  # go from the top
  orig_positions = sorted(erg_tree.treepositions(order='preorder'), key=len)
  cur_len = -1
  temp_pos = list()
  for pos_ex in orig_positions:
    if len(pos_ex) == cur_len:
      temp_pos[-1].append(pos_ex)
    else:
      cur_len = len(pos_ex)
      temp_pos.append(list())
      temp_pos[-1].append(pos_ex)
```

```
  positions_to_examine = list()
  for pos_list in temp_pos:
    newlist = pos_list[::-1]
    for pos in newlist:
      positions_to_examine.append(pos)
  while positions_to_examine:
    position_to_check = positions_to_examine.pop(0)
    parent_node = erg_tree[position_to_check]
    if (isinstance(parent_node, Tree)
        and len(parent_node[0:]) > 0
        and isinstance(parent_node[0], Tree)):
      position_pos = parent_node.node['pos']
      only_child = len(parent_node[0:]) == 1
      if position_pos in ['VP']:
        leftmost_child = parent_node[0]
        leftmost_child_pos = leftmost_child.node['pos']
        if leftmost_child_pos not in ['MD', 'VP', 'VB', 'VBD',
                                      'VBG', 'VBN', 'VBP', 'VBZ']:
          move_to_left_of_parent(erg_tree, leftmost_child.treeposition)
          if only_child:
            del erg_tree[parent_node.treeposition]
          positions_to_examine = \
            strip_subtrees_from_positions(positions_to_examine, position_to_check)
```

A.11 Rule K: modified_collapse_unary

```
def modified_collapse_unary(tree, collapsePOS=False, collapseRoot=False):
  if collapseRoot == False and isinstance(tree, Tree) and len(tree) == 1:
    node_list = [tree[0]]
  else:
    node_list = [tree]

  found_rule = False
  # depth-first traversal of tree
  while node_list != []:
    node = node_list.pop()
    if isinstance(node, Tree):
      if (len(node) == 1
          and isinstance(node[0], Tree)
          and (collapsePOS == True or isinstance(node[0,0], Tree))):
```

```
        prev_node = node[0].node
        child_nodes = list()
        for i in range(len(node[0])-1, -1, -1):
          child_nodes.insert(0, node[0].pop(i))
        node[0:] = child_nodes
        node_list.append(node)
        if (found_rule == False and
            not (prev_node['rule'].endswith('_C')
                 or prev_node['rule'].endswith('_c'))
            and (node.node['rule'].endswith('_C')
                 or node.node['rule'].endswith('_c'))):
          node.node = prev_node
          found_rule = True
      else:
        found_rule = False
        for child in node:
          node_list.append(child)
        parent = node
    else:
      found_rule = False
```

A.12 Rule L: add_rb_parent_phrases

```
def add_rb_parent_phrases(erg_tree):
  # go from the top
  # TODO exclude fragments
  phrasal_labels = ['S', 'SBAR', 'SBARQ', 'SINV', 'SQ', 'ADJP', 'ADVP', 'CONJP',
                    'FRAG', 'INTJ', 'LST', 'NAC', 'NP', 'NX', 'PP', 'PRN', 'PRT',
                    'QP', 'RRC', 'UCP', 'VP', 'WHADJP', 'WHAVP', 'WHNP', 'WHPP', 'X']
  positions_to_examine = sorted(erg_tree.treepositions(order='preorder'), key=len)
  while positions_to_examine:
    position_to_check = positions_to_examine.pop()
    node_to_examine = erg_tree[position_to_check]
    if isinstance(node_to_examine, Tree):
      position_pos = node_to_examine.node['pos']
      if node_to_examine._parent is None:
        continue
      siblings = node_to_examine._parent[0:]
      sibling_poses = [x.node['pos'] for x in siblings]
      phrasal_bin = [bool(x in phrasal_labels) for x in sibling_poses]
```

```
      phrasal_siblings = any(phrasal_bin)
      if position_pos in ['RB', 'RBR', 'RBS'] and phrasal_siblings:
        node_parent = node_to_examine._parent
        index_on_parent = node_to_examine.parent_index
        parent_pos = node_parent.node['pos']
        if parent_pos not in ['ADVP', 'ADJP']:
          if parent_pos in ['VP'] and index_on_parent != 0:
            continue
          new_node = node_to_examine.copy(deep=True)
          new_node.node = deepcopy(node_to_examine.node)
          node_to_examine[0:] = [new_node]
          node_to_examine.node['pos'] = 'ADVP'
          positions_to_examine = \
            strip_subtrees_from_positions(positions_to_examine, position_to_check)
```

A.13 Rule M: add_initial_npsubj_trees

```
def add_initial_npsubj_trees(erg_tree):
  # for S-like things which are not the first element of the second layer
  nouny_tags = ['CC', 'CD', 'DT', 'NN', 'NNS', 'NNP', 'NNPS',
                'NP', 'PRP', 'PRP$']

  positions_to_examine = sorted(erg_tree.treepositions(order='preorder'), key=len)
  while positions_to_examine:
    position_to_check = positions_to_examine.pop()
    if not isinstance(erg_tree[position_to_check], Tree):
      continue
    position_to_check_pos = erg_tree[position_to_check].node['pos']
    if position_to_check_pos not in ['S']:
      continue
    # this is S-like; find children
    children = erg_tree[position_to_check][0:]
    if len(children) < 2:
      continue
    child_poses = [x.node['pos'] for x in children]
    child_positions = [x.treeposition for x in children]
    if 'VP' in child_poses:
      first_vp_location_idx = child_poses.index('VP')
      first_non_punc_idx = 0
      for idx in range(first_vp_location_idx):
```

```
        if child_poses[idx] not in ['"', "''", "'", '``', '`']:
          first_non_punc_idx = idx
          break
      if first_vp_location_idx != first_non_punc_idx:
        node_to_examine = erg_tree[child_positions[first_non_punc_idx]]
        if node_to_examine.node['pos'] in nouny_tags:
          new_node = node_to_examine.copy(deep=True)
          new_node.node = deepcopy(node_to_examine.node)
          node_to_examine[0:] = [new_node]
          node_to_examine.node['pos'] = 'NP-SBJ'
          positions_to_examine = \
            strip_subtrees_from_positions(positions_to_examine, position_to_check)
```

A.14 Rule N: move_prepositional_phrases

```
def move_prepositional_phrases(erg_tree, w1, w2):
    # repeat this until there are none left to do
    prep_phrase_moved = True    # start with true to force at least one run-through
    while(prep_phrase_moved):
        prep_phrase_moved = False
        # go from the top, lop off terminals
        all_positions = set(erg_tree.treepositions(order='preorder'))
        leaf_positions = set(erg_tree.treepositions(order='leaves'))
        these_positions = all_positions - leaf_positions
        orig_positions = sorted(these_positions, key=len)

        cur_len = -1
        temp_pos = list()
        for pos_ex in orig_positions:
            if len(pos_ex) == cur_len:
                temp_pos[-1].append(pos_ex)
            else:
                cur_len = len(pos_ex)
                temp_pos.append(list())
                temp_pos[-1].append(pos_ex)
        positions_to_examine = list()
        for pos_list in temp_pos:
            newlist = pos_list[::-1]
            for pos in newlist:
                positions_to_examine.append(pos)
```

```
        while positions_to_examine and not prep_phrase_moved:
            position_to_check = positions_to_examine.pop(0)
            parent_node = erg_tree[position_to_check]
            if (isinstance(parent_node, Tree)
                    and len(parent_node[0:]) > 0
                    and isinstance(parent_node[0], Tree)):
                position_pos = parent_node.node['pos']
                only_child = len(parent_node[0:]) == 1
                if position_pos in ['NP', 'NP-SBJ', 'ADJP']:
                    rightmost_child = parent_node[-1]
                    rightmost_child_pos = rightmost_child.node['pos']
                    if rightmost_child_pos in ['PP']:
                        move_to_right_of_parent(erg_tree, rightmost_child.treeposition)
                        prep_phrase_moved = True
                        if only_child:
                            del erg_tree[parent_node.treeposition]
                        positions_to_examine = \
                          strip_subtrees_from_positions(positions_to_examine,
                                                        position_to_check)
```

A.15 Rule O: correct_tree_node_pos

```
def correct_tree_node_pos(original_tree):
  all_positions = sorted(original_tree.treepositions(order='preorder'), key=len)
  leaf_positions = original_tree.treepositions(order='leaves')
  positions_to_examine = \
    strip_preterminals_from_positions(all_positions, leaf_positions)
  for position in positions_to_examine:
    original_pos = original_tree[position].node['pos']
    if original_pos in ['NN', 'NNS', 'NNP', 'NNPS', 'PRP', 'PRP$', 'CD',
                        'POS', 'DT', '$', '.', '``', "''"]:
      original_tree[position].node['pos'] = 'NP'
    elif original_pos in ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ', 'MD']:
      original_tree[position].node['pos'] = 'VP'
    elif original_pos in ['IN']:
      original_tree[position].node['pos'] = 'PP'
    elif original_pos in ['RB', 'RBR', 'RBS', 'TO']:
      original_tree[position].node['pos'] = 'ADVP'
    elif original_pos in ['JJ', 'JJR', 'JJS']:
      original_tree[position].node['pos'] = 'ADJP'
```

```
elif original_pos in ['WDT', 'WP', 'WP$']:
  original_tree[position].node['pos'] = 'WHNP'
elif original_pos in ['WRB']:
  original_tree[position].node['pos'] = 'WHAVP'
elif original_pos in ['CC']:
  original_tree[position].node['pos'] = 'CONJP'
elif original_pos in ['UH']:
  original_tree[position].node['pos'] = 'INTJ'
elif original_pos in ['LS']:
  original_tree[position].node['pos'] = 'LST'
elif original_pos in ['WRB']:
  original_tree[position].node['pos'] = 'WHAVP'
elif original_pos in ['EX']:
  original_tree[position].node['pos'] = 'S'
```