Egad: Efficiently Evaluating and Extracting Errors from Deep Grammars

Michael Wayne Goodman

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

Master of Arts

University of Washington

2009

Program Authorized to Offer Degree: Linguistics

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Michael Wayne Goodman

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Emily M. Bender

Francis Bond

Date:

University of Washington

Abstract

Egad: Efficiently Evaluating
and Extracting Errors
from Deep Grammars

Michael Wayne Goodman

Chair of the Supervisory Committee:
Assistant Professor Emily M. Bender
Department of Linguistics

A useful property of deep grammars, such as those based on HPSG, is the ability to generate as well as parse sentences. Much effort is put into increasing the parsing coverage of a grammar, but less attention is given to its ability to generate. In this thesis, I introduce a system (**Egad**) that considers both the parsing and generation output of a grammar, then uses that information to find areas where performance differs between the two. **Egad** can be used to analyze the overall generation performance of a grammar—such as how well it can produce paraphrases—as well as, perhaps more importantly, finding probable errors in the grammar. Using **Egad**, we were able to increase the generation coverage of the Japanese grammar Jacy nearly 20%.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

CHARACTERISTIC: Some trait or quality extracted from a profile. Characteristics are used in to describe both entire items as well as their individual parts.

ITEM: An input string along with its sets of parses and realizations, if they are available.

LEXEME: In HPSG grammars, a lexeme is, at least, the pairing of a *stem* and a lexical type, referenced to by a lexical identifier. In this work, a "lexeme" usually means the lexical identifier, since the rest can be looked up in the lexicon.

PROFILE: A set of files produced by the grammar and system profiling tool [incr tsdb()] after it runs over a corpus.

REALIZATION: A generated sentence, including its surface form, derivation tree, and semantics.

STEM: Sometimes called the "orthography", a stem is a string representation of a lexeme. For example, the stem form of the *utsukushii-adj* lexeme is 美しい.

ACKNOWLEDGMENTS

There are many who deserve thanks for making this work possible.

Emily Bender, my adviser at the University of Washington, was exceptionally helpful by providing advice and information during our frequent meetings, even when we were in vastly different timezones. She also read and commented on multiple drafts of papers related to the project, including this one. Her patience and active involvement with all of her students' projects is commendable, and her keen insight into the cryptic errors of the LKB is uncanny.

It was Francis Bond's impressive efforts to make Jacy the best and most comprehensive deep grammar of Japanese that provided the impetus for this project. He invited me to NICT to do this work with him, provided valuable advice and guidance, helped with coding, testing and debugging the system, and coauthored conference and workshop papers related to this document. For these generous acts, a wonderful friendship, and more, I am truly grateful.

Takayuki Kuribayashi graciously and tirelessly provided native-speaker intuitions for a variety of questions (including an untold number about dogs barking), and helped with Jacy's lexical types. The people in the Language Infrastructure Group at NICT gave advice and audience for presentations about the system. It was a pleasure working with all of them.

The DELPH-IN community offered support to our numerous software issues. In particular, Stephan Oepen was a great help with [incr tsdb()], Dan Flickinger with the ERG, and Rebecca Dridan with RMRS comparison.

Alexander Koller and Stefan Thater were very helpful in getting Utool to work with our system, and were exceptionally quick to patch a bug we reported during this time.

The Treehouse members at the University of Washington also gave audience to a presentation and provided useful feedback.

This work was made possible by NICT, who funded travel and accomodations, and provided the resources on which the work was produced. Miwa Shiga and the rest of the staff were a great help in taking care of the paperwork and logistics of my visit.

Chapter 1
INTRODUCTION

Linguistically motivated analysis of text, such as deep parsing, provides much useful information for subsequent processing.[1] Grammars that perform deep parsing are being used in a variety of applications such as machine translation (Bond et al., 2005; Oepen et al., 2007) and relation extraction from structured texts (Nichols et al., 2005). The information obtained by these grammars, however, is generally at the cost of reduced coverage, due both to the difficulty of providing analyses for all phenomena, and the complexity of implementing these analyses. In this thesis I present a method of identifying problems in a deep grammar by exploiting the fact that it can be used for both parsing (going from text to its analysis as a semantic representation) and generation (going from a semantic representation to its realization as text). Since both parsing and generation use the same grammar, their performance is closely related: in general improving the performance or cover of one direction will also improve the other (Flickinger, 2008).

The central idea is that we test the grammar on a full round trip: parsing text to its semantic representation and then generating from it. In general, any sentence where we cannot reproduce the original text identifies a flaw in the grammar. This can be done over raw text: there is no need to treebank. This has two advantages: (a) it is possible to identify problematic phenomena in domains without existing treebanks, and (b) it is possible to identify which of these phenomena are most widespread, so that grammar development effort can be focused on the most rewarding problems. In order to train ranking models and regression test the grammar, it is still important to treebank some text during grammar development (Oepen et al., 2004). This round-trip testing is an addition to the existing set of grammar engineer's tools, not a substitution.

We call our system **Egad**, which stands for Erroneous Generation Analysis and Detec-

[1]See Uszkoreit (2002) for more motivation for research into deep linguistic processing.

tion.

1.1 Motivation

The specific motivation for this work was to increase the quality and coverage of paraphrases produced by the Japanese grammar Jacy (Siegel, 2000a). Bond et al. (2008) showed that by producing paraphrases of the English side of an aligned corpus (using the English Resource Grammar (Flickinger, 2000, 2008): the ERG) they could effectively increase the training corpus size and thus improve the performance of a statistical machine translation system. We want to increase the size of the Japanese side of the corpus for the same reasons. However, Jacy could not generate new sentences nearly as well as the ERG, and was thus unusable for the task. We are thus specifically focusing on improving generation coverage in this work. Improving generation would also greatly benefit X-to-Japanese machine translation tasks using Jacy.

1.2 Initial State of the Grammar

Figure 1.1 shows the capabilities[2] of both the ERG and Jacy (for more information on these two grammars, see Section 3.1). The ERG has 87% parsing coverage, 83% generation coverage, 70% can generate the original string (i.e. parsing and generation are symmetric), and 66% can generate different strings (i.e. paraphrases). In contrast, Jacy has 82% parsing coverage, 45% generation coverage, only 11% can generate the original string, and 44% can generate different strings. Clearly Jacy has room for improvement.

1.3 Goals

We hope that the end result of this system will allow us easily modify our grammar so it can reach the following goals over a larger subset of input sentences:

1. Parse the source sentence

2. Generate the parsed sentence

[2]These statistics are taken from a profile created using the Tanaka corpus as of 2008.08.08.

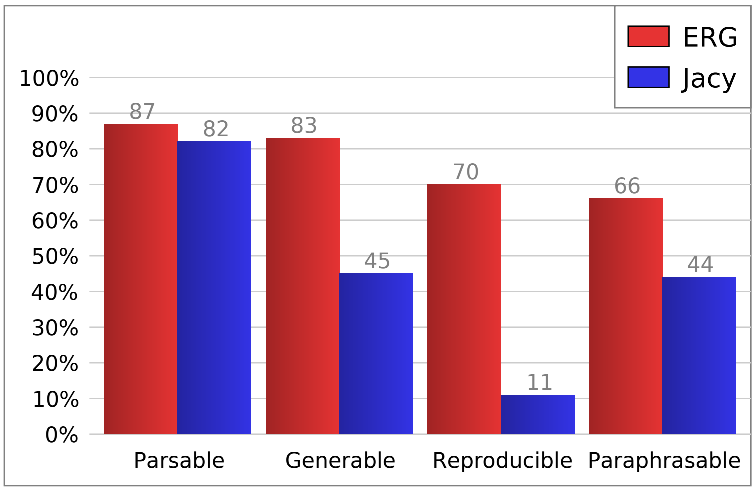

Figure 1.1: Initial grammar statistics

3. Generate different sentences when such paraphrases are possible

Also, for the generated sentences, we would like to fulfill the following goals:

1. Generated sentences are grammatical

2. Semantics of generated sentences are subsumed by the input semantics

This means that the grammar will be more robust: it parses and generates more sentences, and will generate more natural sentences.

1.4 Thesis Overview

In Chapter 2, I review the existing literature related to error mining, providing background for design decisions and situating this work in the field. In Chapter 3, I discuss the resources we used and overview the processes by which **Egad** characterizes and error mines a grammar. Chapter 4 goes into further detail about characterizing a grammar, with explanations of each feature we look at. I explain in Chapter 5 how we analyze parse tree (more specifically *derivation tree*) structure and use pieces of the structure, along with the information from the grammar characterization stage, to build a model for error mining. Chapter 6 further discusses the error mining process, including discussion of the kinds of errors **Egad** can find, and how to use **Egad**'s output to locate example items exhibiting the error. In Chapter 7, I list and discuss several of the errors in Jacy that we were able to find and fix. I provide an analysis of **Egad** in Chapter 8 with statistics from Jacy's original and improved generation coverage. Finally I conclude in Chapter 9 with future goals for the system and an overview of the work.

Chapter 2
LITERATURE REVIEW

Grammar developers generally try to improve parsing, rather than generation, coverage, and as a result most of the existing error mining for deep grammars focuses on parsing errors. In this section I will review works in error mining as well as other supporting works.

2.1 Error Mining for Parsing

The seminal work in error mining for parsing with deep grammars is van Noord (2004), who parsed a large corpus and used differences between sentences that parsed and those that didn't to identify problematic N-grams. A parsability score is calculated for all N-grams, depending on how often they occur in sentences that parse compared to the overall distribution. N-grams with low parsability ("suspicious forms") identify input that is problematic for the grammar. This method worked well, but would also give a low parsability score to N-grams that happened to occur in unparsable sentences, regardless if they were the true source of the error or not (suspicion-by-association).

Sagot and de La Clergerie (2006) alleviated the problem of suspicion-by-association by stating that all unparsable sentences have at least one source of error, then working to find the most likely problematic N-gram. They introduced an iterative, fixed-point algorithm that considered a suspicious form's appearance in parsable sentences, the existence of more than one suspicious form in the same sentence, and the length of each sentence in the calculation of parsability for each form. Sagot and de La Clergerie also created a novel graphical user interface for presenting the results of their error mining process.

More recently, de Kok et al. (2009) built on the work of Sagot and de La Clergerie (2006) to find problematic N-grams of arbitrary length. To alleviate the data sparsity issues that occur with longer N-grams, they considered an N-gram suspicious if the whole thing had a lower parsability score than each smaller N-gram it contains. Doing so particularly helped with multiword expressions.

The vast majority of parsing errors are caused by an inadequate lexicon, so most results in the previous approaches point out missing lexemes. In contrast, our method only considers items that have parsed, assumes the parse is correct, then looks for errors that arise in generation. The hope is that our approach will find more problems in grammar rules rather than just lexical inadequacies.

2.2 Error Mining for Generation

Gardent and Kow (2007) presented a system to detect overgeneration in a deep grammar. Their approach was to have a human annotate generated sentences as PASS or OVERGENERATION, after which their code would find tree structures that are likely associated with OVERGENERATION—that is, tree structures that only appear in sentences labeled OVERGENERATION. At this point a human would select a problem and fix it in the grammar. They then reparse and generate from their corpus, and their code would check to make sure they didn't lose any valid (PASS) coverage and that they did indeed reduce the OVERGENERATION sentences. This iterative process would continue for each problem found. The authors were able to reduce generation outputs by 70%, most of which was overgeneration. The types of problems they found included missing constraints, incomplete constraints and incorrect feature percolation, illicit elementary trees, incorrect semantics, and lexical exceptions.

The method described by Gardent and Kow is similar to ours in that we both look for grammar internal structures as errors, rather than strings of input text. Unlike their method, ours cuts the human out of the error detection stage. Rather than manually labeling items as acceptable or unacceptable, we rely on characteristics obtained from the items themselves. Further, their method only considers tree structures that appear solely in the unacceptable items, which could miss errors that can appear in both acceptable and unacceptable items. Both of our methods rely on humans to make changes to the grammar. Their method contains logic for tracking the effects of changes on the grammar, whereas our method relies on the existing [incr tsdb()] system for this task.

2.3 Work on Semantics

The well-formedness of the semantic representation of a sentence can also be used as an indicator for problems in a grammar. Minimal Recursion Semantics are primarily flat structures, with labels and handles linking predicate arguments (e.g. semantic roles). In the MRS for a sentence, the predicates and the links between them form a network, or graph. Flickinger et al. (2005) proposed that nearly all well formed sentences have semantic structures that form a full net. They found eleven rules in the ERG that consistently produced non-net sentences, and that every one of those rules contained an error in its implementation.

Another approach is that of Dickinson and Lee (2008) who, rather than looking for common structures in unparsable sentences, looked for anamolous annotation patterns in the semantic annotation of a corpus. The method is called the "variation N-gram method". It creates a probabilistic model of the likelihood that a particular argument structure (the semantic annotation) is accurate given the verb and some of its context. They ran their system over PropBank (Palmer et al., 2005), a version of the Penn Treebank augmented with semantic role labels, and found that 69% of a sample of 100 items flagged as deviant in their semantic annotation also contained syntactic annotation errors. While annotation errors have different sources and are quite different from errors in the output of a deep grammar, we can still learn a couple of things from this method. First, the anomaly detection logic could be useful for our approach as well, since considering the context around a possible error can help avoid false positives. Second, it is not coincidence that semantic errors accompany syntactic errors, and we find this observation to be true in our system, as well (see Section 4.1.9 for an example).

2.4 Representation of Grammar Rules

Toutanova et al. (2004) introduced a method of representing tree structures as non-branching paths of applied grammar rules. Their motivation was for parse ranking, for which they noted considerable gains using their method, but we think the approach is applicable for us as well. Toutanova et al. created *leaf projection paths*, which, for each leaf node in a derivation tree, is a direct path from the leaf to the root node. The authors did this because many machine learning algorithms require a vector of data of fixed dimensionality, and derivation trees do not easily fit this mold. Therefore they create vectors from sequences of rule applications in a derivation tree.

For our system, we use n-gram subsequences from these leaf projection paths. We do this, rather than using the full paths, because we believe that the source of a grammar error is likely to be contained in a small sequence of rules (most likely just one or two), and not the whole leaf-to-root sequence. If we use the full paths, any nodes that do not represent errors would become noise and pollute the model. Because we are just interested in short substructures from the derivation trees, we could also look at sibling relationships in the trees instead of just parent-child relationships, but we do not explore that possibility in this work.

2.5 Summary

I have reviewed the mainstream approaches to error mining for parsing as well as work on the detection of overgeneration. I also reviewed work others have done on using well-formedness of semantics and anomalies in semantic annotation for error detection. A method of representing derivation trees as multiple vectors inspired our approach to representing grammar errors with these substructures of trees.

Compared to some of the methods of error detection reviewed in this chapter, our method sacrifices some framework independence in exchange for more informative results. Rather than reporting N-grams of input text likely causing problems with a grammar, we present the actual grammar rules thought to be erroneous. Also, unlike previous work in the detection of overgeneration, we fully automate the error detection process so the user does not have to spend time annotating output. This allows the user to spend their time implementing fixes for the grammar, and since nothing needs annotation, this also allows the error detection process to analyze a large number of items.

Chapter 3
METHODOLOGY

The error mining method described in this thesis is not standalone–it requires, at least, an implemented grammar, a parser and generator, and a corpus of input items in order to create the raw data (e.g. derivation trees, MRS semantics) that will be used by **Egad**. In this chapter I will cover the various tools and resources we use with **Egad** in Section 3.1, as well as provide a high-level overview of the item characterization and error mining processes in Section 3.2. I also briefly discuss performance issues relating to the parsing and generation process in Section 3.3

3.1 Resources

To ensure that the error mining process is language independent we tested it on two grammars: Jacy (Siegel, 2000a), a Japanese grammar and the English Resource Grammar (ERG) (Flickinger, 2000, 2008). Both grammars are from the DELPH-IN[1] group, are written in the Head-driven Phrase Structure Grammar (HPSG) (Pollard and Sag, 1994) framework, and use Minimal Recursion Semantics (MRS) (Copestake et al., 2005) for their semantic representations. The ERG is more mature, having had more developer time than Jacy. They are both comparable in complexity, and share many core analyses. They provided some of the motivation for the LinGO Grammar Matrix (Bender et al., 2002): a skeleton of grammatical and lexical types that constitutes a possible formal backbone for a large scale grammar of — in principle — any language.

We use the efficient PET parser (Callmeier, 2000) for parsing and the LKB (Copestake, 2002) for grammar development and generation. Parsing uses a highly optimized chart-based algorithm. Generation is also chart-based with various optimizations (Carroll and

[1]Deep Linguistic Processing with HPSG Initiative – see `http://www.delph-in.net` for background information, including the list of current participants and pointers to available resources and documentation.

Oepen, 2005).

Generation results can legitimately differ from parsing in at least three principled ways: (i) It is normal to select different subsets of the grammar for parsing and generation. For parsing, an additional set of robust rules are allowed, such as rules for the English grammar than allow subject-verb agreement mismatches. (ii) In phenomena where the grammar makes no semantic distinction between two constructions, they will both be allowed in generation. For example in Jacy, SOV (subject-object-verb) and OSV sentences will typically have the same semantics and both will be generated from the same input. (iii) The grammar writer can also block lexical variants in generation. For example, Jacy can parse two kanji variants for *kirei* "pretty" (奇麗, 綺麗) in addition to the hiragana version (きれい), but only generates the hiragana.

Since generation with HPSG grammars constructs sentences from MRS semantic representations, one optimization for the generation process only inserts lexical entries with semantic contribution (i.e. an MRS predicate) into the generator chart. Note that all words in a realization must first exist as lexical entries on the generator chart. Lexical entries lacking a semantic predicate will only be inserted into the chart if a "trigger rule" defined by the grammar developer matches a certain context in the MRS. For example, in Jacy all morphemes are treated as lexemes. The lexeme for the perfective verb-final morpheme た *ta* contains no semantic predicate, so it must be inserted with a trigger rule. A rule inserts this lexeme if there exists a verb marked as being past-tense[2] in the MRS.

We use [incr tsdb()] (Oepen and Carroll, 2000) for performance profiling: parsing and generating from test-suites and storing their parse trees, MRS, and surface forms in **profiles**. In this thesis, I will refer to a corpus **item** as the collection of an input sentence, its parse information (parse trees, MRS), and its generation information (generated trees, MRS, surface forms).

We use the Tanaka Corpus (Tanaka, 2001) for both our English and Japanese input sentences. The Tanaka Corpus was created by Professor Yasuhito Tanaka, who asked his students to collect 300 English-Japanese translation pairs each. The original corpus had a

[2] "Past-tense" is a misnomer, since it marks perfective aspect and not tense, and is likely a result of using labels borrowed from another system or grammar.

significant amount of errors (misalignments, spelling and grammar errors, etc.) and duplicates, but much work has been done to clean it up. The work to clean up the corpus is ongoing, fueled largely by volunteers. Many of the sentences were taken from books, songs, and other sources, so there are many domains represented. The average sentence length (i.e. word-count) for the English side of the corpus was 7.72 words, and the longest was 45 words. The version of the corpus we used contained nearly 150,000 translation pairs.

3.2 Process

The error mining process consists of three stages of processing, of which **Egad** does the latter two. The first stage is creating parsing and generation profiles from a corpus. This is achieved with the PET parser for parsing, the LKB for generation, and [incr tsdb()] for the management of profiles. The second stage is to determine **characteristics** (see Chapter 4) of the parsed and generated items. The third and final stage finds the grammar rules most predictive of certain characteristics.

Parsing and generation with HPSG grammars uses the same grammar rules for both directions, although there may be additional rules affecting only one direction. First, input text is parsed with the grammar, producing derivation trees and MRSs for each parse. Generation then uses the MRSs to construct derivation trees and output text sentences. While it is possible to produce realizations from each MRS of multiple parses for the same input sentence, **Egad** only looks at realizations produced from the first parse. This will be the highest ranked parse if there is a parse ranking model.

To characterize the parsed and generated items, **Egad** uses both the parsing and generation profiles. This step is fairly straightforward, as it only requires the information from each individual item to determine that item's characteristics. These characteristics are useful for analyzing the performance of a grammar, but are also an integral part of the rule-association stage and thus for error-mining. Most characteristics are trivial and easily observable by looking at the parsing or generation profile, but others are difficult for humans to quickly discern. For example, it is time-consuming for a human (and occasionally for a computer, when the complexity is high enough) to determine if two MRSs are equivalent. The resulting characteristic summary produced by **Egad** is easily processable by humans and computers both.

In order to extract rules that appear to cause grammar errors, **Egad** constructs a statistical model where n-grams of grammar rules (**rule-paths**; see Section 5.1) from an item predict that item's set of characteristics. After collecting these rules-to-characteristics relations over the whole corpus, the model learns which rule-paths are associated with particular patterns of characteristics. The grammar developer then looks for rules that are, according to the model, strongly associated with an undesirable characteristic. See Section 5.2 for more information on the building of this model.

3.3 Notes on Performance

The user can place limitations on the parser and generator to compensate for machine performance or to manage run time. For our tests, we limited the parser to use no more than 50,000 edges (rule-applications in the search space of all parses). We limited the generator to use no more than 10,000 or 20,000 edges, we only collect the first 5 realizations per input item. If the grammar has a generation ranking model, the first 5 realizations will also be the top 5, but currently Jacy does not have a generation ranking model, so it is merely the first ones the generator produces. For each item, the generator only uses the top parse (highest ranked of all valid parses) as input. For the initial version of Jacy (prior to the changes listed in Chapter 7), Table 3.1 shows the distribution of realizations per input item. The data for this table came from 7,500 input items, of which 6,115 parsed. Also note that it is possible for the generator to run out of memory or timeout before reaching a realization, resulting in no realizations being returned for a given item, even if the grammar theoretically allows them. Unfortunately I don't have numbers for these kinds of errors.

3.4 Summary

In this chapter I have described the tools and resources **Egad** uses, and briefly covered the process by which items are parsed, generated, characterized, and how rules are associated with characteristics. The parser (PET), generator (LKB), and grammar (Jacy) are all in the DELPH-IN family of tools and grammars, and, along with sentences from the Tanaka Corpus, supply **Egad** with the data it works with. There are a variety of reasons why

Table 3.1: Distribution of the number of realizations per input item.

Generations	Count	Percent.
0	2,709	44.30%.
1	147	2.40%.
2	220	3.60%.
3	73	1.19%.
4	246	4.02%.
5	2,720	44.48%.
Total	6,115	≈100.00.

generation is not always symmetric with parsing, and **Egad** does two main steps to find such asymmetries in the grammar: first it analyzes and reports characteristics detailing the parses and realizations, and second it creates a statistical model associating grammar rules to these characteristics. These associations form the foundation of error mining with **Egad**.

Chapter 4
ITEM CHARACTERIZATION

The first thing that **Egad** does with parse and generation profiles is determine characteristics from each item. Most of the information used in this determination already exists in the profile, but it is spread across different files or not immediately usable. For instance, the set of lexemes in a parse or realization is contained in the derivation tree of an item. This stage of **Egad**'s processing extracts these data and uses them to determine characteristics both for items as a whole (parses and realizations), as well as in comparing each realization to the parse whence its semantics originated. In this chapter I will describe the various characteristics **Egad** reports and, using these characteristics, provide an analysis of the items and grammar performance over our test corpus.

4.1 Characteristics

From both the parsing and generation profiles, we are able to extract a handful of characteristics that can tell us something about the grammar used to produce those profiles. I will now briefly list the characteristics **Egad** determines, and they will be described in greater detail below. For input items, we look at those that are:

- Parsable
- Generable
- Reproducible
- Paraphrasable

Looking at individual generated sentences, we also analyze ways in which realizations can differ from the original sentence, including:

- The set of lexemes
- The derivation tree
- The set of rules used
- The surface string
- The MRS

Finally, there is one characteristic that applies to both parsed and generated sentences. It is determined by the external program Utool.

- The MRS does not form a net

After we determine all of these characteristics, we output a formatted and easily searchable string representation of the whole set, called a **characteristic pattern**, or CP. The possible values[1] for each characteristic are as follows:

0 There was no problem or difference observed

1 There was a problem or difference observed

– The characteristic is not applicable for this type of item.

? The value of the characteristic could not be accurately determined.

An example of a CP for a parsed sentence is given in Figure 4.1.

The first four numbers can be read, in order, as "parsable, generable, not reproducible, paraphrasable". The following five characteristics do not apply to a parsed item, so they are simply dashes. The final one in the CP means that the parse's MRS is ill-formed. An example CP for a generated sentence is given in Figure 4.2.

[1]In early versions of **Egad**, we considered each characteristic as representing an error. Thus, the reason why "0" means parsable, generable, etc. is because they used to mean unparsable, ungenerable, etc. In this previous system "0" meant "false" or "not observed," and "1" meant "true" or "observed."

```
0010 ----- 1
```

Figure 4.1: Example characteristic pattern for a parsed sentence.

```
---- 01110 0
```

Figure 4.2: Example characteristic pattern for a generated sentence.

In this case, the first four values are dashes as they do not apply to generated sentences. The next five can be read as “Has the same lexemes, a different derivation tree, a different set of rules, a different surface form, and the same MRS as the original parsed item”. The final zero means that the MRS is well formed.

Below we further define these characteristics, explain how they are determined, provide example sentences, and summarize their role in error detection.

4.1.1 Parsable

We find this characteristic by looking at a parsing profile and seeing which input strings do or do not have at least one corresponding complete parse. Once we find an item that is unparsable, there is little other information that can be gathered about it for further processing with **Egad**. Recall that Chapter 2 described what others have done to locate parsing errors, but, unlike those approaches, our system requires derivation trees to find errors. We have considered using partial parses to find common tree structures in unparsable sentences, but this is delegated to future work. (1) shows a sentence that was not parsed by the grammar.

(1)	その	事故 で、	あやうく	彼 は	命 を	落とす	ところ	だった。
	sono	jiken-de,	ayauku	kare-wa	inochi-wo	otosu	tokoro	datta.
	that	accident-INST,	nearly	him-TOP	life-ACC	lose	about to	was.

“The accident almost cost him his life.”

Although our main motivation is to improve generation, looking at unparsable items can still be useful. If a source sentence cannot be parsed, then the grammar would also be unable to generate that sentence given its semantics. Granted, the grammar may be able to generate different sentences that have the same semantics, but the source sentence is often produced by a human and is a natural way of expressing the meaning represented by the semantics, so it is beneficial to ensure that it can be parsed.

It is worth noting that 6% of a sample of 100 unparsable sentences in the Japanese corpus were perhaps unparsable for reasons other than errors in the implemented grammar. Of those 6 sentences, 2 were borderline grammatical, 1 was ungrammatical, and 3 were not single sentences (e.g. more than one sentence, bylines, etc.). This classification of parsing errors mirrors that found by Baldwin et al. (2005), i.e. parsing errors due to missing lexical entries, missing grammar constructions, ungrammatical input, and extragrammatical input. Similarly, for the English corpus, we found that 16% of a sample of 100 unparsable sentences had such problems (10 were ungrammatical, 1 was borderline grammatical, and 4 were not single sentences). For most of these items, we could say that the grammar did well in not parsing them. Also of note is that some of the Japanese unparsable items were grammatical, but not segmented properly by the morphological analyzer Chasen (Matsumoto et al., 2000), which might have caused the failure.

4.1.2 Generable

For sentences the grammar was able to parse into MRS, we would hope that it could at least generate the same sentence, but sometimes it does not generate anything at all. It is easy to detect these items–simply look for those that appear in a parsing profile, but not in the generation profile. However, we do not have any information about the failed generation process.

There are at least two pieces of information from the parsing profile that may prove useful: the derivation tree and the MRS. We can look at the derivation trees and MRSs from items that can generate and those that cannot, and attempt to pinpoint which individual rules or predicates are strongly associated with generability. In this thesis, I do this method for derivation trees, but not MRSs.

Consider (2). This item did not generate because the quantification rule was unable to be applied to the phrase 3匹 *san-biki* “three (animals)”. To be more specific, the quantification rule was a lexical rule, and thus should only be applied on lexemes, not phrases. The parser, however, allowed this constraint to be ignored, while the generator did not. If the sentence had an explicit subject (say, 3匹の犬が吠える *3 biki no inu ga hoeru* “three dogs bark” or 犬が3匹吠える *inu ga 3 biki hoeru* “three dogs bark”) then the single lexeme 犬 *inu* “dog” could be quantified without any problems and the sentence would generate properly. But nevertheless, this quantification issue is a grammar bug, and the generator was behaving properly. Quantification should be a phrasal rule, rather than a lexical rule.

(2)	３匹 が	吠える。
	san-biki-ga	hoeru.
	three-CL-NOM	bark.

“Three (animals) bark.”

Another possible reason for ungenerability is that the generator ran out of memory. Sometimes the solution is just to process the items on a machine with more memory (particularly for very long or complicated sentences), but the fact that it fails in such a way suggests that the rules may not be properly applied (or if they are, they could be implemented in an inefficient or problematic way).

4.1.3 Reproducible

Reproducibility means that, using the semantics of a parsed sentence, the generator can realize the exact same sentence as was parsed. In other words, the parse and realization are symmetric. If we can parse a sentence and generate from the resulting semantics, but cannot generate the sentence we initially parsed (see Section 4.1.8 for how we judge if the sentences are equal), then there are several possible culprits for this asymmetry. Note that when determining if a grammar produced the same sentence, we only compare the sentence’s surface form. It might be more correct and informative to also compare the derivation trees and semantics, but we did not explore that possibility in this version of **Egad**.

The first possible reason for irreproducibility is that the original lexemes are not being selected (see Section 4.1.5 for discussion on this phenomenon). (3) is an example of such a sentence:

(3)

タバコ が	とても	古かっ た。
tabako-ga	totemo	furukat-ta.
cigarettes-NOM	very	old-PERF.

“The cigarettes were very old.”

Given the semantics for this sentence, Jacy could generate three variants (hiragana, katakana, and kanji, respectively) for *tabako* “cigarette”: たばこ, タバコ, and 煙草. Japanese in particular often has several ways to write the same word, and in the Jacy grammar each form has its own lexical entry. When there are several competing lexemes, as in this case, the preferred form (likely the same as what was parsed) might not be generated first. Since we are only considering the first 5 realizations, if the preferred form does not show up in those 5, then **Egad** will consider them to not have been generated at all. This is the reason that (3) was labeled as having different lexemes in the parsed and generated sentences.

A second possibility for irreproducibility is also lexical, in that the semantics for a parsed word may be shared among several different words (this is distinct from the previous possibility, which was different forms of the same word). Pronouns are an excellent case in point. In Jacy, pronouns always have the semantic predicate *_pron_n_rel*, and use additional properties (person, number, or gender) to discriminate between them. Some pronouns have some of these properties unspecified, and their semantics can therefore select any similarly unconstrained pronouns. For instance, the Japanese pronoun 自分 *jibun* “self” is underspecified for person, number, and gender, so when Jacy generates a sentence from an item containing this pronoun, it inserts any pronoun as a possible realization. This overgeneration, as with the lexical variant problem, can easily cause the first 5 realizations to not be reproductions.

There are other possible reasons for irreproducibility. Insufficient trigger rules may cause the generation process to fail to insert a lexeme (although this might more often cause a generation failure). Also, a longer sentence with a larger number of constituents will likely allow more than 5 word reorderings. There are perhaps other causes we have not considered, but all of those discussed essentially cause irreproducibility in one of two ways: being unable to produce the original string, or not producing the original string in the set of realizations looked at by **Egad**.

4.1.4 Paraphrasable

While being able to generate the original parsed string satisfies the goal of being generable, if no other strings can be generated then we have failed to satisfy the goal of generating paraphrases. In **Egad**, a generated sentence is a paraphrase if the surface form is different from the parsed sentence, without regarding the derivation trees or semantics. To be more correct, the surface forms don't have to be exactly the same, as the user can provide a list of forms that are to be ignored, such as punctuation.

(4) shows two sentences. (4a) is a realization that is identical to the input (parsed) sentence, while (4b) is a different string. (4b) has a different word order from the original sentence, resulting in a different surface form. (4a) is not considered a paraphrase (it is a reproduction as defined in the previous section), but (4b) is a paraphrase.

(4) a.

彼 は	毎年	海外 へ	行く	。
kare-wa	maitoshi	kaigai-e	iku	.
he-TOP	every.year	overseas-LOC	go	.

"He goes abroad every year."

b.

彼 は	海外 へ	毎年	行く	。
kare-wa	kaigai-e	maitoshi	iku	.
he-TOP	overseas-LOC	every.year	go	.

"He goes abroad every year."

These grammars only produce strict paraphrases. There is no specific functionality to do sentence compression, lexical replacement, or complex restructuring. Note that these kinds of changes may take place as long as the semantics of the resulting sentence is the same as the original. For instance, the pronoun 彼 *kare* "he" may be replaced with another pronoun that matches the same constraints (third person, singular, masculine), such as そいつ *soitsu* "that guy". Also, as we can see in (4b), simple word reordering can take place.

4.1.5 Lexemes Differ From Original

Turning now to characteristics of generated strings, we start with whether the set of lexemes used in a realization is the same or different from those used in the parse. This characteristic is easy to determine. The penultimate nodes on every branch in a derivation tree are always lexeme identifiers, which are unique labels for an entry in the lexicon. Because of this fact, the set of lexemes for any parse or realization is just this set of nodes (since we are only looking for lexical equality, we do not need to know more than a lexeme's identifier). After we have the sets for both the parse and a realization, we just check to make sure the (unordered) sets are equivalent.

Lexemes are the basic units from which sentences are formed in HPSG grammars. While they can be thought of as "words", they are actually much more general. For instance, in the Jacy grammar there are lexemes for some kinds of punctuation (e.g. a comma: 、), morphemes (e.g. perfective inflections -た *-ta* or -だ *-da*[2]), and some word collocations and idioms (e.g. 毎週 *maishuu* "every week", お先に *osaki ni* "before; ahead of; pardon me (for leaving before you)"). Also, lexemes are pre-inflection, so they are usually the base form of a word. Despite these differences, a set of lexemes is often a good approximation of the words used in a sentence.

(5) shows a parsed sentence (5a) and a generated sentence (5b) where the lexemes differ:

(5) a. 話し相手 が 欲し い 。
hanashi-aite-ga hoshii .
speaking-partner-NOM want .
"(I) want someone to talk to."

b. 話し相手 が ほし い。
hanashi-aite-ga hoshii .
speaking-partner-NOM want .
"(I) want someone to talk to."

These lexemes differ in orthography, but have identical semantics. There is nothing wrong with either sentence, although it may be a stretch to say that the second is a paraphrase of the first since they both have the same words and structure. (6), however, is an example of parsed and generated sentences with different lexemes and semantics.

(6) a. そう おっしゃっ て 下さっ て ありがとう
sou osshat-te kudasat-te arigatou
so say-TE give-TE thank-you
"It's very kind of you to say so."

[2]The difference between -た *-ta* and -だ *-da* is merely phonological and there are no syntactic implications.

b. *

そう	おっしゃって	下さい	ありがとう	か
sou	osshat-te	kudasai	arigatou	ka
so	say-TE	give	thank-you	Q

* "Please say so thank you?"

The second has both a different inflection for the verb 下さる *kudasaru* "give (honorific)" (which, for the Jacy grammar, means a loss of the separate て *te* lexeme) and the insertion of the interrogative sentence ending particle か *ka*. Therefore, there are two lexical differences in this pair of sentences. Note that the second sentence is also ungrammatical.

4.1.6 Derivation Tree Differs From Original

When we compare derivation trees of parsed and generated sentences, we only look at phrasal nodes. That is, we exclude two levels of leaf nodes, since those levels are always stems and lexemes. We also remove some other information from the trees, such as edge numbers. After cleaning up the trees in this way, we can check if the derivation trees are identical without being concerned with differing lexemes (after all, we have a separate characteristic to look for those). Figure 4.3 shows a derivation tree for (7a) with two levels of leaf nodes in a different color, illustrating those nodes that would not be compared. The underlined nodes in the derivation tree show which subtrees would be swapped to produce the sentence is (7b). Note that it is just the underlined nodes and their left subtrees, not the right subtrees (i.e. head-specifier-rule becomes the right daughter of head_subj_rule). Figure 4.4 is the textual format of the derivation tree, which is what is actually compared in **Egad**. Note that the lexical nodes have been stripped out.

(7) a.

鳥は	鋭い	目を	もっている
tori-wa	surudoi	me-wo	mot-teiru
bird-TOP	sharp	eye-ACC	have-STAT

"Birds have sharp eyes."

b.

鋭い	目を	鳥は	もっている
surudoi	me-wo	tori-wa	mot-teiru
sharp	eye-ACC	bird-TOP	have-STAT

"Birds have sharp eyes."

(7) is a fine example of where we would like to see differences in derivation trees. In this example, the reordered second sentence is a valid, if slightly unnatural, paraphrase

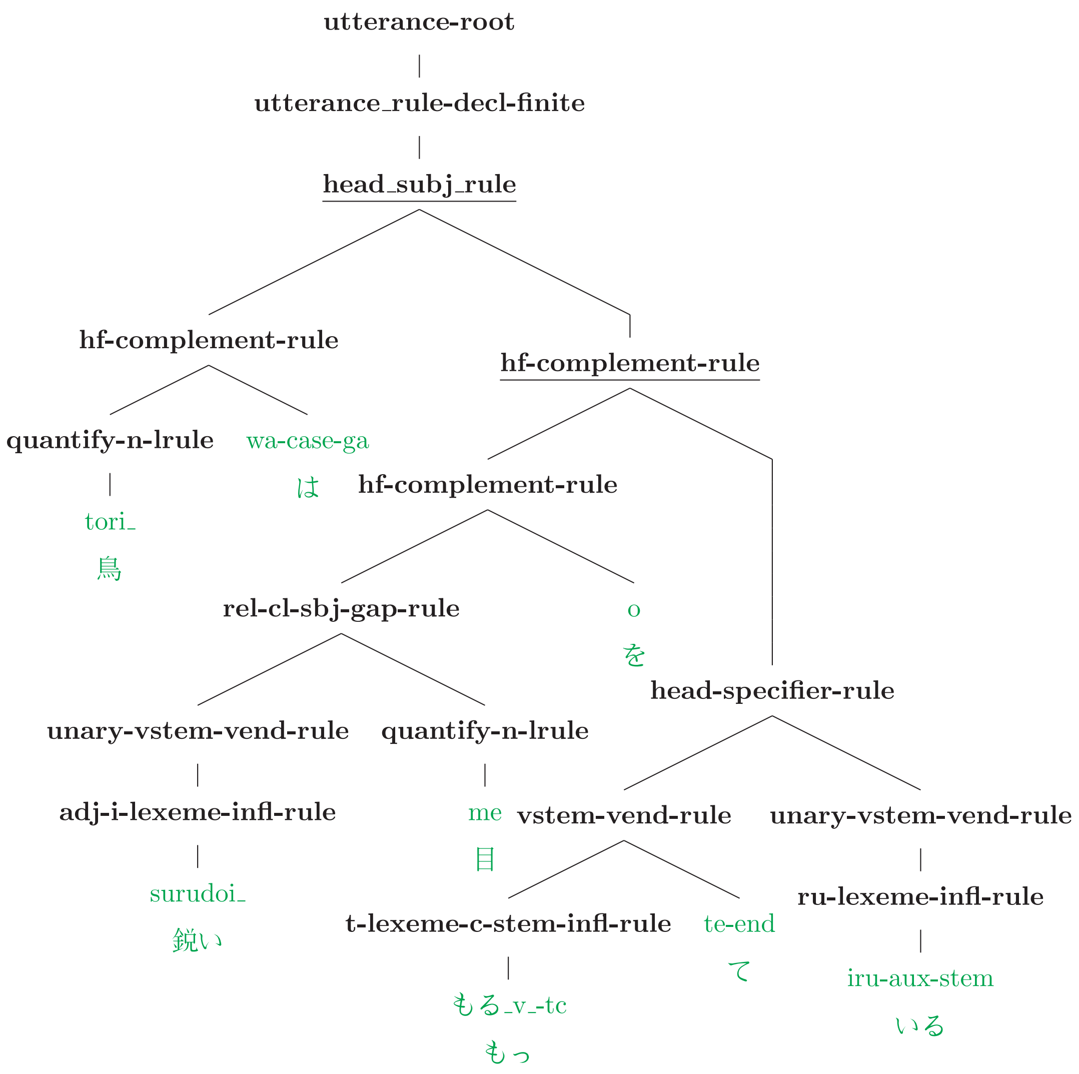

Figure 4.3: Parse Tree for "鳥 は 鋭い 目 を もっ て いる"

```
(utterance-root(utterance_rule-decl-finite(head_subj_rule(hf-complement-rule
(quantify-n-lrule))(hf-complement-rule(hf-complement-rule(rel-cl-sbj-gap-rule
(unary-vstem-vend-rule(adj-i-lexeme-infl-rule))(quantify-n-lrule)))
(head-specifier-rule(vstem-vend-rule(t-lexeme-c-stem-infl-rule))
(unary-vstem-vend-rule(ru-lexeme-infl-rule)))))))
```

Figure 4.4: Textual format of the derivation tree in Figure 4.3

of the original. There are some pairs of minimally restructured sentences that do not get labeled with this characteristic. For further discussion and an example, see Appendix A. Since Japanese can license many different word orderings for sentences, most, if not all, realizations with different derivation trees are grammatical (although often marked). We have only seen this characteristic exhibited on erroneous items that have some other characteristic as well, such as different lexemes or different rules, as in (8).

4.1.7 Rules Differ From Original

In addition to the derivation trees, we also look at the unordered set of phrasal rules for a parse or realization. This helps us distinguish which sentences (from those that have differing derivation trees) are merely reorderings and which are not. Note that all realizations having a different set of rules as the parsed sentence will also have different derivation trees.

(7) above did not have any different rules, only different rule structure. (8) does have a different set of rules. The realization is an ungrammatical sentence, but it has the same lexemes and, according to Jacy, the same MRS as the parsed sentence. This was easily found by searching for sentences with differing derivation trees and differing sets of rules.

(8) a.

昨日 は	とても	寒かっ た
kinou-wa	totemo	samukat-ta
yesterday-TOP	very	cold-PRF

“Yesterday was very cold.”

b.

*	昨日 は	とても	寒く た
	kinou-wa	totemo	samuku-ta
	yesterday	very	coldly-PRF

* “Yesterday was very coldly.”

4.1.8 Surface String Differs From Original

If there are any differences in the final string of the parsed and generated sentences, aside from some kinds of punctuation,[3] then the realization is labeled with this characteristic.

[3]The range of punctuation that is ignored is defined by the user. For Japanese, we used the list that the Jacy grammar itself ignores, which includes quotes, periods, asterisks, etc. The reason for ignoring them is because they are not always generated in the same form as they were parsed, if at all, and we wanted to avoid these trivial differences.

This characteristic is very common, as almost every realization with lexical, rule, or tree structure differences will also have a surface string difference. In the initial version of Jacy that we examined, more than half (51.5%) of the realizations exhibited this characteristic, but this may not be true for different languages or future versions of Jacy.

(3) was found by searching for items that had differing surface forms but the same lexemes and derivation trees. Notice that the generated sentence has the spurious (and incorrect) inflection with り *ri* at the end of the verb 吠え *hoe* "bark". Affixing rules do not appear on the derivation tree, which is why we do not catch this error by looking at differences in the derivation trees. There are other cases where the derivation tree check fails to detect differences in realizations, and this is discussed in Appendix A.

(9) a.

猫 が	吠え 方 が	分かっ た
neko-ga	hoe-kata-ga	wakat-ta
cat-NOM	bark-method-DAT	understand-PRF

"The cat knew how to bark."

b. *

猫 が	吠えり 方 が	分かっ た
neko-ga	hoeri-kata-ga	wakat-ta
cat-NOM	barking-method-DAT	understand-PRF

* "The cat knew how to barking."

Since this sentence was generated, it would be parsable as well. It is difficult for a grammar developer who only tests parsing coverage to anticipate this sort of error. The grammar developer can test with negative examples (to ensure they don't parse), but it would be very difficult and inefficient to try and cover all cases by doing so. Analyzing generation results allowed us to easily spot this error.

4.1.9 MRS Differs From Original

In comparing MRSs between parsed and generated sentences, we check not if they are equal, but if they are equivalent. The parsing and generation processes may assign different label and handle names or order the predicates differently, but the two may still be equivalent. Finding equivalency then becomes a graph comparison task. We wrote our own code for this comparison despite the existence of other packages with similar functionality (Dridan and Bond, 2006) because our task is slightly different from what the developers of the other packages chose to look at. We want to judge if two MRSs are equivalent rather than calculate a score of dissimilarity.[4]

Our process of determining equivalency checks that there are the same number and kinds of elementary predicates, then ensures that the argument labels of one form the exact same graph structure as the other. Analyzing the graph structure is largely a recursive process—it looks at each pair of predicates (one from each side being compared), assumes a mapping of their arguments, then proceeds with each successive predicate unless a conflict appears. There are a few optimizations we do to speed up this inherently slow process. First of all, we only compare predicates of the same PRED value, and in order of their number (e.g. we first look at predicates of which there are only one in each MRS, then two, etc.). The most time-intensive comparisons involve predicates that are high in number, which tend to be pronouns (because, in Jacy at least, they are all the same type of predicate: _pron_n_rel) and quantifiers. We therefore handle quantifiers separately and do some things to lessen the ambiguity, such as looking at which NPs they affect. We don't currently do anything special with pronouns. If we find a valid mapping of arguments covering all predicates, we halt the comparison routine and say the two MRSs are equivalent.

(10) is an example of a generated sentence whose semantics differs from the original parsed sentence. In this case, the change can be attributed to a different ordering of phrasal constituents (both the set of rules and lexemes are the same). The generated structure created at least two incompatible attachments: the 同 *dou* "same" attached to 画家 *gaka* "artist" instead of 時代 *jidai* "era", and the 中 *naka* "among" attached earlier, resulting in 画家中の *gaka-chuu-no* "those among the painter(s)" instead of 画家の中 *gaka-no-naka* "among the painter(s)". In other items, reasons for nonequivalent MRSs include differing lexemes (see Section 7.1 for discussion of this problem) or different rules (such as a verb taking a different inflection, e.g. the imperative form instead of the te-form).

[4]While we should be able to judge equivalency with a score of dissimilarity, the package we tried out was giving nonzero scores for two MRSs known to be identical. Future iterations of that package may be useful for our purposes.

(10) a.

ターナー は	同 時代 の	画家 の	中	でも	傑出	し ている
taanaa-wa	dou-jidai-no	gaka-no	naka	demo	keshhutsu	shi-teiru
Turner-TOP	same-era-GEN	painter-GEN	among	even	excel	do-STAT

“Turner stands out among the painters of his time.”

b. ?

時代 の	同 画家 中 の	でも	ターナー は	傑出	し ている
jidai-no	dou-gaka-chuu-no	demo	taanaa-wa	kesshutsu	shi-teiru
era-GEN	same-painter-among-GEN	even	Turner-TOP	excel	do-STAT

? “Turner stands out at those among the same painters of a time.”

Note that in some particularly difficult cases (e.g. those with a large number of pronouns or nouns, as these introduce many predicates of the same type) we occasionally timeout before we can finish the equivalency calculation. When this happens, the code reports “?” instead of “0” or “1”, signifying that we did not get an accurate answer.

4.1.10 MRS Not Forming A Net

The final characteristic we looked at and that I discuss in this thesis is that of semantic well-formedness. We use the Utool program (Koller and Thater, 2005) to analyze the MRS from a parsed or generated sentence. Before we do that, we must convert the MRS into Prolog, as Utool is, as of the time of this writing, incapable of reading the default MRS format output by [incr tsdb()]. Utool’s return value will let us know if the MRS forms a net or not, if it is logically ill-formed in some other way, or if there was an error reading the MRS.

In (11), the generated sentence (11b) was determined by Utool to have a non-net MRS:

(11) a.

太郎 が	テーブル を	綺麗 に	拭い た	。
tarou-ga	teeburu-wo	kirei-ni	fui-ta	.
Tarou-NOM	table-ACC	clean-NI	wipe-PERF	.

b. *

太郎 が	テーブル を	綺麗 であった に	拭い た	。
tarou-ga	teeburu-wo	kirei-deatta-ni	fui-ta	.
Tarou-NOM	table-ACC	clean-was-NI	wipe-PERF	.

“Tarou wiped the table clean.”

As Flickinger et al. (2005) predicted, the generated sentence (11b) is not only semantically ill-formed, but syntactically erroneous, and should be fixed in the grammar. This problem occurred because the に *ni* marker in Jacy was not constrained for tense, and during generation it accepted the perfective であった *deatta* "was" (which was likely inserted in the generator chart by a trigger rule). This problem is further discussed in Section 7.2. Whether an MRS is a net or not is, as we can see by the discovery of this problem, a useful characteristic for debugging.

4.2 Results of Item Characterization

We will demonstrate grammar analysis both with individual items and the grammar as a whole. The former reveals details about individual sentences or types of sentences, while the latter is useful for gathering statistics about the grammar.

4.2.1 Item Analysis

First we will look at how Egad characterized Jacy's output for (12).

(12)	あの	木 は	これ	ほど	高く ない	。
	ano	ki-wa	kore	hodo	takaku-nai	.
	that	tree-TOP	this	as	tall-NEG	.

"That tree is not so tall as this."

The characteristic pattern generated by **Egad** is `0010 ----- 0`. The first four values in this CP indicate that it can be parsed, generated (evidenced by the two realizations provided below in (13) and (14)), that it cannot generate the original string, and that it generated strings other than the original. The next five values are irrelevant for parsing. The final value indicates that the MRS is well formed. Given this sentence and its realizations, this CP is accurate.

We will now look at two sentences Jacy generated from the parsed semantics of (12). First is (13), which **Egad** gave the CP `---- 01010 0`:

(13)	これ	ほど	あの	木 は	高く ない	。
	kore	hodo	ano	ki-wa	takaku-nai	.
	this	as	that	tree-TOP	tall-NEG	.

"That tree is not so tall as this."

The first four values are now irrelevant, but the next five values are particular to generation. The first one (0) means that the lexemes are the same (which we can verify by looking at the string or derivation tree). The next two (1, 0) mean that the derivation tree is different, but the set of rules are the same, which indicates a reordering of constituents. The fourth value (1) means that the surface form has changed, which is obvious because the constituents have been reordered. The fifth value (0) means that the MRS is the same as the MRS from the parsed sentence. The final value (0) means that this generated sentence has well-formed semantics.

Now consider (14). **Egad** gave this sentence a CP of `---- 11010 0`.

(14)	こいつ	ほど	あの	木 は	高く ない	。
	koitsu	hodo	ano	ki-wa	takaku-nai	.
	this	as	that	tree-TOP	tall-NEG	.

“That tree is not so tall as this.”

The only difference between this realization and the previous is that the first value is now a 1, meaning that there is a lexical difference. This can be observed by the use of the casual こいつ *koitsu* “this” instead of the more general これ *kore* “this”. Both this CP and the previous one are accurate for their respective generated sentences.

4.2.2 Corpus Analysis

With the kind of information available for individual items, as described in Section 4.2.1, one can calculate statistics of an entire corpus. For instance, one could output the percentage of items generated by a grammar that have lexical variation. See Table 4.1 for these kinds of results from the ERG and Jacy. One could also obtain an estimate of how many items can produce paraphrases, or how many can generate the original string (a rudimentary statistic for generation accuracy). These kinds of results for the initial grammar were shown in Figure 1.1 in Section 1.2, and are presented in tabular form in Table 4.2.

4.3 Summary

This chapter outlined the various characteristics we determine from parsed and generated items, as well as covering how they can be used to analyze the performance of a grammar. Full items (parse and realizations) can be characterized as being parsable, generable, reproducible, and paraphrasable, and individual realizations can be compared to the associated

Table 4.1: Initial comparative statistics

	ERG	Jacy
Lexemes differ	50%	90%
Tree differs	69%	72%
Rules differ	64%	51%
String differs	63%	94%
MRSs differs	5%	10%

Table 4.2: Initial general statistics

	ERG	Jacy
Parsable	87%	82%
Generable	83%	45%
Reproducible	70%	11%
Paraphrasable	49%	44%

parse for differences in lexemes, derivation trees, rules, surface forms, and MRSs. These characteristics can be used to analyze single items in order to better understand what is wrong or different. Many of them collected from a large corpus can be used to get general performance statistics about a grammar over that corpus. These characteristics will be an integral part of the error mining process.

Chapter 5
RULE ASSOCIATION

Our approach to detecting problematic rules is to train a classifier with the rules as features and the characteristic patterns (CPs), as labels. Once trained, we use it as though we were doing feature selection to find the individual features (i.e. rules) that are the most closely associated with a some CP. Doing this, we can find rules that are most predictive of a unique CP, or of a range of CPs.

The rules we use as features are not necessarily single rules, but could be paths of rules from the derivation tree. By using paths of rules, we get information about the parse structure and the interaction of rules with each other. We use a maximum entropy-based classifier for this task, and it provides useful results, but we are not claiming it is the only or best tool for the job.

In this section, I will explain what "rule paths" are and how we obtain them. I will then describe how they are used to train a classification model. Finally, I explain how we can use the model to rank grammar rules according to their association with different CPs.

5.1 Rule Paths

The **rule paths**, or RPs, we extract from the derivation trees are single-node paths. That is, every time we encounter a branching node in the tree we will extract a separate path for each branch. Unlike the method we use for tree comparison (described in Section 4.1.6), we do include lexemes and stems. This is because it may be individual lexemes that cause problems. We take n-grams of those paths, where the range of n can be configured by the user. Take, for example, the sentence in (15). Its derivation tree is shown in Figure 5.1, and the trigram RPs we extract are shown in Figure 5.1. Note that if a branch, from the top node to the leaf, is not at least depth n, it will not be included in the set of n-grams.

During the development of **Egad**, there was a bug in the generator where derivation trees of realizations inconsistently used root conditions, whereas the parser always applied them. This bug has since been fixed, but we included functionality to ignore RPs that included root nodes, such as the first two RPs in Figure 5.1. We did this because root nodes were becoming strongly—and incorrectly—associated with parsed items, even though such a distinction was not encoded into the input grammars.

5.2 Model Building

We build a classification model by using parsed or generated sentences' RPs as features and each sentence's CP as the class label. The grammar developer can provide two parameters to control the building of the model: the maximum depth, n, of the RPs, and a set of labels to use. The set of RPs for a sentence includes n-grams over all specified values of n. The labels are, to be more accurate, regular expressions that specify equivalence classes of CPs. Using regular expressions allows one to fully specify unique CPs or to generalize over a less granular range of CPs by leaving certain values underspecified. For example, the grammar developer can choose to only build a model comparing items that are reproducible to those that are not. This is not just for more convenient (i.e. less verbose) output, but it can affect the results as well. Uninteresting characteristics can be a kind of noise, causing potentially important results to be distributed across multiple CPs, diminishing their score in the model for the particular characteristic the user is interested in. In addition, the training of the model is significantly faster when using fewer labels. If the grammar developer does not provide a set of labels to **Egad**, then the model will include the fully specified string for all CPs observed in the profile.

For example, consider again (15). We would provide the list of all valid RPs (those in Figure 5.1, minus the first two, adding all valid RPs for other values of n) as a set of features with `01-- ---- 0` (can parse, cannot generate, MRS is valid and forms a net) as the label. Since it did not generate, we do not add observations for the generated sentences.

We are training over all n-grams in the same model, so we allow the user to weight

(15)

彼女 は	写真 写り が	いい	。
kanojo-wa	shashin-utsuri-ga	ii	.
she-TOP	picture-taking-NOM	good	.

"She is good at taking pictures."

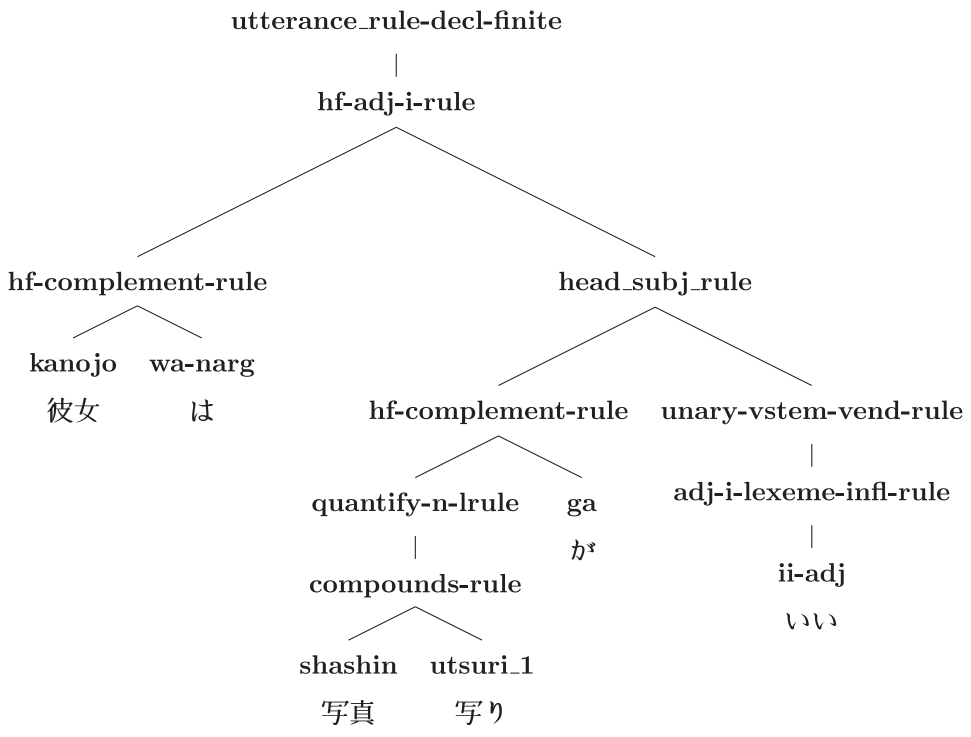

Figure 5.1: Derivation tree for "彼女は写真写りがいい。"

Table 5.1: Trigram rule paths extracted from the tree in Figure 5.1

utterance_rule-decl-finite → hf-adj-i-rule → hf-complement-rule

utterance_rule-decl-finite → hf-adj-i-rule → head_subj_rule

hf-adj-i-rule → head_subj_rule → hf-complement-rule

hf-adj-i-rule → head_subj_rule → unary-vstem-vend-rule

head_subj_rule → hf-complement-rule → quantify-n-lrule

head_subj_rule → unary-vstem-vend-rule → adj-i-lexeme-infl-rule

hf-complement-rule → quantify-n-lrule → compounds-rule features based on their value of n. For example, the user could weight unigrams more than any other n-gram if they want to easily find individual lexemes causing problems.

Regarding sets of CPs to analyze, consider the two regular expressions in Figure 5.2. Using only these two labels, the classifier would look for the most distinguishing features separating items that can generate from those that cannot. Note that neither of these regular expressions matches generated items (although the first item in Figure 5.2 matches generable items), so the classifier would only look at parsed items. Also note that that first item has a period for the third, fourth, and last characteristics. This implies that we do not care whether the parsed item could generate the original string, different strings, or if the MRS forms a net.

Table 5.2: Example set of labels for the classifier.

1. `^00.. ----- .$`
2. `^01-- ----- .$`

Note that we are training our model over grammar rules and not n-grams of words in the input sentence. There are significantly fewer possible n-grams of rules than words, so our model requires fewer training instances to be accurate and interesting. During our tests, we noticed mostly stable results whether we used 1,500 or 150,000 input sentences. I would estimate that the minimum for a decent model is around 1,000 items, since other tests with 200 and 400 items were unstable (i.e. the top results changed significantly between the sets of 200 and 400 items).

5.3 *Finding Rule Associations*

After training the model, we have a classifier that predicts CPs given a set of RPs. **Egad**'s task is not characteristic prediction, but error mining (recall our motivation in Section 1.1), and for this we would like to present the user with grammar rules causing a certain condition in the grammar. Therefore, what we want from the model is to discover the RP most strongly associated with a given CP. The maximum entropy-based classifier we use

(Perl's AI::MaxEntropy package[1]) provides an easy method to view the score a given feature has for some label. We iterate over all RPs, get their score, then sort them based on the score. To help eliminate redundant results, we exclude any RP that either subsumes or is subsumed by a previous RP. For example, if we return an RP $A \rightarrow B$, then following RPs such as B or $A \rightarrow B \rightarrow C$ would be removed, but RPs such as $B \rightarrow C$ would not (since it is not a subset or superset of the first RP, and thus it presents additional information).

Given a CP, the RP with the highest score should indeed be the one most closely associated to that CP, but it might not lead to the greatest number of items affected. This is because the model only represents RP to CP association, not frequency. If an RP only occurs once, and that item has a CP included in the model, then the RP will be strongly associated with that CP, but it is probably not the most important error for the grammar developer to fix. For example, say you want to find and fix rules causing items to be unable to generate the original string. You might find that fixing the problem with the RP having the highest score corrects fewer items than had you fixed those of the RP with the next highest score. Both RPs could be valid errors, but one gives greater benefit for the time invested in fixing the bug. To help the grammar developer decide the priority of problems to fix, we also output the count of items observed with the given CP and RP.

5.4 Rule Association Results

The primary purpose of **Egad** is to aid a grammar developer in finding and fixing problems in a grammar. The process of analyzing the profile and finding these problems is called error mining. Here we explain how **Egad** accomplishes this task.

Table 5.3 lists the ten highest ranked RPs associated with items that could parse but could not generate in Jacy. We could list the highest ranked rules for any set of item characteristics, but, for fixing problems with generation, looking at ungenerable items yielded the most low-hanging fruit. Some rules in Table 5.3 appear several times in different contexts. We used some methods to decrease the redundancy, such as removing items subsuming or subsumed by a previous RP, but clearly this could be improved.

[1]Available at `http://search.cpan.org/~laye/AI-MaxEntropy-0.20/`

Table 5.3: Top 10 RPs for ungenerable items

Score	Count	Rule Paths
1.4234	109	hf-complement-rule → quantify-n-lrule → compounds-rule
0.9601	54	hf-complement-rule → quantify-n-lrule → nominal-numcl-rule → head-specifier-rule
0.7562	63	head-specifier-rule → hf-complement-rule → no-nspec → "の"
0.7397	62	hf-complement-rule → head-specifier-rule → hf-complement-rule → no-nspec
0.7391	22	hf-complement-rule → hf-adj-i-rule → quantify-n-lrule → compounds-rule
0.6942	36	hf-complement-rule → hf-complement-rule → to-comp-quotarg → "と"
0.6762	82	vstem-vend-rule → te-adjunct → "て"
0.6176	26	hf-complement-rule → hf-complement-rule → to-comp-varg → "と"
0.5923	36	hf-adj-i-rule → hf-complement-rule → quantify-n-lrule → nominal-numcl-rule
0.5648	62	quantify-n-lrule → compounds-rule → vn2n-det-lrule

In this list of ten problematic RPs, there are four unique problematic pieces of the grammar represented: `quantify-n-lrule` (noun quantification), `no-nspec` (の *no* used in some constructions of noun specification), `to-comp-quotarg` (と *to* quotative particle), and `te-adjunct` (verb conjugation used to combine verbs, such as the で *de* in (16)).

(16)	ボール を	進ん で	打つ
	booru-wo	susun-de	utsu
	ball-TOP	advance-TE	throw

“Hit the ball stepping forward.”

The extra rules listed in each RP show the context in which each problem occurs, and this can be informative as well. For instance, because we see `quantify-n-lrule` used both with the `compounds-rule` and `nominal-numcl-rule`, we can presume that the problem resides mostly, if not entirely, with the `quantify-n-lrule`, rather than the other two.

What is interesting to note is that each RP listed does indeed contain the rule that is problematic. In other words, while the top ten results do not cover a large spread of problems, each RP covered does represent a problem — it does not mislead.

Further, the problems identified are not always lexically marked. *quantify-n-lrule* occurs for all bare noun phrases (i.e. without determiners). This kind of error cannot be accurately identified by using just word or POS n-grams, we need to use the actual parse tree. Another example is with lexical distinctions that are made grammar-internally, but are not obvious by just looking at the string or even POS tag. The の *no* and と *m*arkers both represent multiple lexical types in the grammar. Some examples of the former include genitive and possessive markers, and a clausal complementizer, and examples of the latter include a quotative marker (one for speech and another for thoughts), and a coordinating conjunction. It would be difficult to single out which is causing the error when only looking at the surface form or POS tags. Grammar-specific supertags could help narrow down the candidates, but **Egad**, being aware of types internal to the grammar, can find and report the exact one causing problems.

5.5 Summary

In this chapter I explained what rule paths are, how we build a model to associate rules to characteristics, and how we use the model to find interesting associations. Rule paths are non-branching n-grams of grammar rules taken from derivation trees. **Egad**'s model uses these rule paths as features that predict classes of characteristics, and we use the model's scoring of each rule path to find those most highly associated with a certain characteristic. Looking for rule paths with high scores for an undesirable characteristic allows the user to find likely erroneous rules, and the count of occurences of those items helps the user find the most important problems to fix.

Chapter 6
ERROR MINING

After **Egad** has characterized the items in a profile and associated paths of grammar rules to specified patterns of characteristics, a grammar developer can use this output for error mining. The first task is to specify which characteristics to find associated RPs for. The second task is finding example sentences exhibiting those characteristics. The grammar developer can use any information from the CP or derivation tree (including stems and lexical identifiers) to find examples.

6.1 Mining Problematic Rules

When **Egad** builds a model of RPs associated to CPs, it scores the RPs according to how predictive they are with regard to the user-provided CPs. That is, if the user only provides two CPs, it will assign scores according to how well each RP predicts one or the other CP. If there are three CPs, it will discriminate among the three. If no CPs are provided, it will discriminate among all observed CPs in the corpus.

In general, the finer-grained the distinctions between CPs are made (e.g. by using all CPs in the model), the less useful the results will be. While the CPs the RPs are associated to will yield more specific information, the RPs will likely overfit to characteristics that are not informative, reducing the score for each association. Also, when RPs representing the same problem are separated across different characteristics, the user has to look in more places to find the problem. For example, consider an RP that occurs often in irreproducible items. In a model that only compares reproducible to irreproducible items (generalizing over the other characteristics), this RP will be highly associated to irreproducible items. If, however, the user built a model with all CPs, this RP may appear in items with different CPs (e.g. irreproducible and paraphrasable, irreproducible and not paraphrasable), and the RP might not be strongly associated to any of them. Even if it was strongly associated to one or both CPs, the user now has to look in more than one place to see what is essentially the same error.

For further examples, see Figures 6.1–6.5. Figure 6.1 simply compares generable to ungenerable items, while Figure 6.2 looks at items that are reproducible or irreproducible. Note that specifying that the items are parsable and generable is redundant and unnecessary, but it might help the pattern be clearer to the user.

```
/01.. ----- ./
/00.. ----- ./
```

Figure 6.1: Search patterns for items that can or cannot generate.

```
/001. ----- ./
/000. ----- ./
```

Figure 6.2: Search patterns for items that are reproducible or irreproducible.

Figure 6.3 compares items that differ, or don't, in semantics. Keep in mind that for the model to work, the user must provide at least two patterns. Usually these patterns will simply compare items where an individual characteristic is observed or not (as in this and previous patterns), but it could compare totally different features or more than two patterns. For example Figure 6.4 compares two patterns where a characteristic is observed in both cases, but it is comparing parses to realizations. This works here because, unlike the other characteristics, the well-formedness of semantics applies to both parses and realizations. Also notice how the use of dashes is used to discriminate between parses and realizations.

```
/---- ....1 ./
/---- ....0 ./
```

Figure 6.3: Search patterns for items that differ or don't differ in semantics.

```
/0... ----- 1/
/---- ..... 1/
```

Figure 6.4: Search patterns for parsed and generated items that have non-net semantics.

Figure 6.5 has four patterns enumerating all possible values of reproducibility and paraphrasability. Note that the ungenerable[1] characteristic is used instead of irreproducible and not paraphrasable, as any realization is either a reproduction or a paraphrase (i.e. there is no CP represented by `/0011 -----./`). While the CPs in Figure 6.5 are all unique and mutually exclusive among the items, there might not be RPs that are strongly associated to each one of them. This is an extended case of Figure 6.1, where instead of simply looking an generable and nongenerable items, we separate those that are generable into different subclasses. Such a set of patterns could be useful when trying to make items generate both reproductions and paraphrases.

```
/01-- ----- ./
/0010 ----- ./
/0001 ----- ./
/0000 ----- ./
```

Figure 6.5: Search patterns for items of all values of reproducibility and paraphrasability.

Besides the patterns the model is built on, the user should provide the value of n—the maximum length of n-grams for rule paths. In our tests, most errors could be easily spotted and fixed with only bigrams, but we were using 4-grams to build the model. We chose to use 4-grams because our tests of the model had greater accuracy with the value of n being 4 rather than 2, and the time complexity of training the model was still acceptable. The user should run some tests to find the optimal value for their grammar. Future versions

[1]The third and fourth value in this CP use the "inapplicable" character (–). Because the item did not generate, **Egad** does not check whether it was reproducible or paraphrasable. We could also generalize (e.g. with the dot character) for these, and it would match the same set of items, but it would be less clear that the second two values are inapplicable.

of **Egad** will likely use the iterative training approach of Sagot and de La Clergerie (2006) and the support for arbitrarily long n-grams described in de Kok et al. 2009. After these changes have been made, the selection of n will no longer be relevant. See Chapter 2 for more information on these approaches.

6.2 Item Searching

Once a grammar developer has found a problematic RP (from such as those reported in Table 5.3 in Section 5.4), they may want to find examples of it in their corpus. The output from the item characterization step of **Egad** places the item's ID number, its CP, its surface string, and its derivation tree on one line in a file, making it ideal for searching with popular command-line tools like *grep*.

Because **Egad** outputs the full CP regardless of whether it is for a parse or a realization, one can use the inapplicable character ("-") as an anchor for a regular expression search, and also as a way to limit the search to only generation or parse items. Figures 6.6–6.10 are a sample of grep searches that a user could make.

```
grep " ---- 1.... ."
```

Figure 6.6: grep search for realizations with different lexemes than the original.

```
grep " ---- .10.. ."
```

Figure 6.7: grep search for realizations with differing derivation trees, but the same set of rules as the original.

```
grep " 0010 ----- ."
```

Figure 6.8: grep search for items that are irreproducible and paraphrasable.

```
grep " 01-- ----- . .* 彼 "
```

Figure 6.9: grep search for ungenerable items containing the word 彼 *kare* “he”.

```
grep " ---- ....1 . .*te-adjunct"
```

Figure 6.10: grep search for realizations with differing MRSs and using *te-adjunct* lexemes.

6.3 Summary

I covered the details of the error mining process in this chapter. First, the user must select patterns of item characteristics and the maximum size of the n-grams from which the model will be built. After building the model, and selecting a problem to look at from the results, the user can use search tools such as grep to find examples of that problem in their corpus. Finding examples is important for helping the grammar developer understand and reproduce the problem. By using this method of error mining, the grammar developer can significantly reduce the time needed to find and categorize errors.

Chapter 7
GRAMMAR CHANGES

In this chapter, I go into some detail about the grammar fixes applied to Jacy as a result of analyses obtained by **Egad**, showing the variety of problems that we were able to identify. **Egad** also identified some issues with the ERG: both over-generation (an under-constrained inflectional rule) and under-generation (sentences with the construction *take* {*care*|*charge*|...} *of* were not generating.

7.1 Overgenerating Topic Markers

Jacy's original analysis of punctuation such as commas, colons, and equals was such that they were subtypes in the semantic hierarchy of the topic marker は *wa*. This caused the punctuation to generate any time the original sentence used a wa-topic marker (as in (17)). We found this problem by looking at realizations with different semantics than the parsed sentence.

(17) a.

その	計画 は	具体 化	し て	き た
sono	keikaku-wa	gutai-ka	shi-te	ki-ta
that	plan-TOP	tangible-ization	do-TE	come-PERF

"The project is taking shape"

b. その 計画 、 具体 化 し て き た

c. その 計画 ： 具体 化 し て き た

d. その 計画 ＝ 具体 化 し て き た

Figure 7.1 shows the original semantic hierarchy for topic relations, where _wa_d_rel is at the top. Figure 7.2 shows the updated version, where we introduced a new top-level node, _wa_d_super (which is not the semantic predicate of any lexeme, unlike _wa_d_rel). We restructured this part of Jacy so the punctuation characters were sister, rather than daughter, types of the topic marker, and this fixed the problem. All are still treated as topic relations, so there is no drop in parsing coverage and we still get accurate semantic representations of input sentences, but we don't have the overgeneration problem as before.

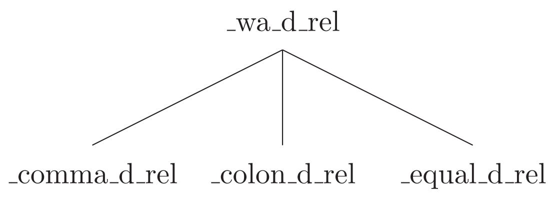

Figure 7.1: Original semantic hierarchy for topic relations in Jacy

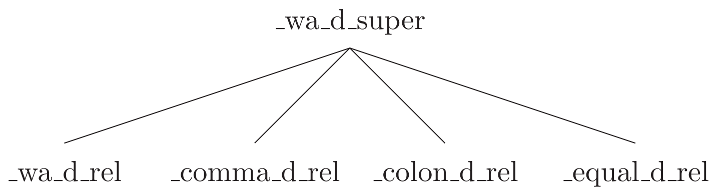

Figure 7.2: Updated semantic hierarchy for topic relations in Jacy

7.2 Incorrect Tense Constraints

(18) is an ungrammatical sentence found by looking at the sentences judged to have ill-formed MRS semantics. The correct sentence would be as in (19).

(18)	* 静か	で あった	に	犬	が	吠えた
	shizuka	de atta	ni	inu	ga	hoeta
	quiet	de was	in	dog	NOM	barked

“The dog barked quietly”

(19)	静か に	犬 が	吠え た
	shizuka-ni	inu-ga	hoe-ta
	quiet-ADV	dog-NOM	bark-PRF

“The dog barked quietly”

The problem was with the に *ni* particle not properly constraining its complement in regards to tense. We constrained it to present-tense, which fixed these instances, but lost the ability to accept items such as (20). Adding a new に *ni* particle for tenseless verbs like 遊び *asobi* “playing” regained the ability to accept these items.

(20)	子供 が	遊び に	いく
	kodomo-ga	asobi-ni	iku
	child-NOM	play-LOC	go
	“The child goes to play”		

7.3 Unquantifiable Nouns

Some nouns, such as the compound noun in (15) or the nominalized numerical classifier in (2), were able to be parsed, but could not be generated. The problem was that the rule for quantifying the nouns, `quantify-n-lrule`, was a lexical rule, which is not supposed to apply on phrasal nodes. The parser, however, will relax this constraint and allow it to be accepted. The generator is not as permissive, and does not allow such a construction. Figure 7.3 shows how the `quantify-n-lrule` was applied at a phrasal node during parsing.

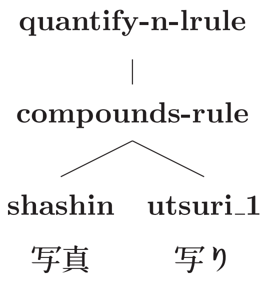

Figure 7.3: Noun compound with quantification by lexical-rule

Having the quantification be a lexical rule was not only problematic for generation, but was also an incorrect analysis, so we converted it to a phrasal rule: `quantify-n-rule`. Doing so cleared up the issue and allowed these kinds of nouns to be quantified, and hence generated.

7.4 Noun Specification with の **no**

Japanese often uses の *no* for noun specification. There were two types of this kind of specification that were problematic for generation with Jacy. The first is when a noun is specified by another noun, such as 出発 の 合図 *shuppatsu no aizu* "departure signal". The second is when a noun is given a numeric count in such a construction, such as 40 人 の 生徒 *40 nin no seito* "40 students".[1] The の *no* in both of these does not show up in the MRS semantics, thus they are not automatically entered into the generator. We found this problem by looking for ungenerable items, and fixed it by adding trigger rules that insert these into the MRS if certain conditions are met (such as a noun with a numeric count).

7.5 Overgenerating Pronouns

As discussed in Sections 4.1.3 and 4.1.4, Jacy had problems with the generation of pronouns. Japanese has many variations of pronouns, and if the semantics of a parsed pronoun match any others, then they, too, will be generated. For example, Jacy has the following third-person singular masculine pronouns: あいつ *aitsu*, そいつ *soitsu*, 誰々 *daredare*, 誰彼 *darekare*, 誰それ *daresore*, 彼 *kare*, 奴 *yatsu*. This is even an incomplete list, as there are kanji and hiragana versions of most of them. The list of first person singular pronouns is even longer. The worst offender of this is 自分 *jibun* "self", which in Jacy was unconstrained for person, number, or gender, and would thus generate any pronoun.

The overgeneration of pronouns was causing the items to be irreproducible, because the generator was generating the other variants before the original one (recall that we only look at the first 5 realizations, not the full set). We therefore found this problem by looking at irreproducible items.

There is also the issue of pragmatics with regard to pronouns. Japanese distinguishes many words according to politeness, providing a means for the speaker to express honor towards the adressee or to referrents in the situation described. Jacy encodes these distinctions in a separate feature to semantics: CONTEXT, because the pragmatic information is not strictly truth conditional (Siegel, 2000b). Much of the over-generation comes from the fact that this CONTEXT is not currently used by the generator. This is not an issue for parsing, but strongly degrades generation. We have not yet implemented a fix for the issue of pragmatic differences between pronouns.

[1]But note that this phrase is ambiguous. It could either mean "40 students" or "the student belonging to 40 people."

We did, however, make an initial fix to the problem of reflexive pronouns, allowing Jacy to distinguish between standard pronouns 私 *watashi* “I/me”) and reflexive pronouns (自分 *jibun* “self”), by moving this information from pragmatics into the semantics. This change fixed relatively few items (compared to problems with other pronouns), but it is interesting to note that there were no items with 自分 *jibun* “self” that were reproducible, so it is an important fix.

7.6 Summary

In this chapter I discussed some of the problems that we were able to find with **Egad**, and how we fixed them. This list is not complete, but it shows the variety of problems **Egad** could help us discover. We found items by looking at differences in semantics, ill-formed semantics, ungenerability, and irreproducibility. Some changes affected a large number of items, and other changes fixed serious, if relatively uncommon, problems. All of our changes only required four weeks of grammar development time, and had significant effects to the generation coverage, as described in Chapter 8.

Chapter 8
ANALYSIS

After fixing the most significant problems in Jacy (outlined in Chapter 7) as reported by **Egad**, we obtained new statistics about the grammar's coverage and characteristics. For ease of comparison, we show the original Jacy statistics next to the updated statistics in Tables 8.1 and 8.2, and also as bar charts in Figures 8.1–8.3. Both tables include the counts of items where a characteristic was observed as well as percentages. We use the version numbers from the Subversion repository in this chapter. Jacy 419 is the original version we started with, and Jacy 441 is the updated version after our grammar changes.

8.1 Coverage Analysis

Table 8.1 describes general characteristics, so it considers every input item. The absolute percentage ("abs") is the percentage of items exhibiting a characteristic out of all input items (in this case, 7,500), while the relative percentage ("rel") uses the count of relevant items as the denominator. Only parsable items are relevant when looking at generablity, and likewise only generable items are relevant when looking at reproducibility or paraphrasability.

Table 8.1: Jacy's improved general statistics

	Jacy 419			Jacy 441		
	count	abs	rel	count	abs	rel
Parsable	6,115	82%	-	6,244	83%	-
Generable	3,406	45%	56%	4,738	63%	76%
Reproducible	809	11%	24%	1,639	22%	35%
Paraphrasable	3,336	44%	98%	4,595	61%	97%

Consider Table 8.1. There were large gains in generation coverage — we nearly halved the remaining under-generation. A large portion of this is due to our fix for quantifying noun phrases (see Section 7.3). Our changes to improve generation also increased parsing coverage by 1%, which is a 5.5% reduction of error. We doubled the absolute percentage of reproducible items, which is relative increase of about 46% (24% to 35% of generable items). For these reproductions, some are due to generating sentences that couldn't generate previously, but another part is because we filtered out unwanted realizations, allowing the reproduction to appear in the first 5 realizations. Regarding paraphrasable items, the absolute percentage increased from 44% to 61%, but we see that the relative percentage actually dropped 1%. This is because paraphrasable items did not increase fully in proportion to generable items. The relative drop in paraphrasable items coupled with the increase in reproducible items is likely the effect of our efforts to reduce spurious overgeneration.

8.2 Realization Analysis

Table 8.2 shows statistics from comparisons between parses and all realizations, so the count of all realizations is used as the denominator (note that this is different from the count of generable items). For Jacy 419, the count of all realizations is 15,390. For Jacy 441, it is 19,793.

Table 8.2: Jacy's improved comparative statistics

	Jacy 419		Jacy 441	
	count	%	count	%
Lexemes differ	13,852	90%	15,951	81%
Tree differs	11,122	72%	15,683	79%
Rules differ	7,928	52%	11,449	58%
String differs	14,518	94%	17,988	91%
MRS differs	1,539	10%	1,270	6%

Increasing the percentage of reproducible items is, intuitively, correlated with a drop in the percentages of items with differing strings. We also noticed significant drops in the percentage of items with different lexemes and MRS. Many of these surely are because of the fixes of topic markers (see Section 7.1) and pronouns (see Section 7.5). There were increases in items with differing trees and rules, despite the 1% relative drop in paraphrasable items, and this, too, can be attributed to the large gain in generable items.

8.3 Concerning Paraphrases

One statistic we chose not to explore directly with **Egad** is the number of paraphrases per paraphrasable item. We don't provide this statistic to the user because the number of realizations (including paraphrases) is not reliable.[1] Nevertheless, it is not difficult for the user to figure out an approximation of this number with the output from the characterization stage of **Egad**. Jacy 419, with 3,336 paraphrasable items and 14,518 paraphrases, has about 4.4 paraphrases per paraphrasable item. Jacy 441, with 4,595 paraphrasable items and 17,988 paraphrases, has around 3.9.

8.4 Summary

This chapter presented results from the analysis of our efforts to fix errors in the Jacy grammar discovered by **Egad**. I presented an analysis of the change in coverage statistics, showing how much generation coverage increased with a small amount of grammar-fixes. I explained the changes to realization statistics, such as the proportion of realizations with different semantics or derivation trees. Some statistics are not reported by **Egad** (as they could be inaccurate), but can be approximated by the user with a little effort. I presented the number of paraphrases per paraphrasable item as an example. Looking at these analyses, we see that the grammar is now much better at generating (up to 63% from 45%) and is better at reproducing the original string, which (given that we only look at 5 realizations per item) is an indicator of a reduction of spurious overgenerations.

[1]Recall that if the generator runs out of memory or reaches the edge limit, no generations are produced. Also, we are only looking at the first 5 generations, which means that the true number is not being reported.

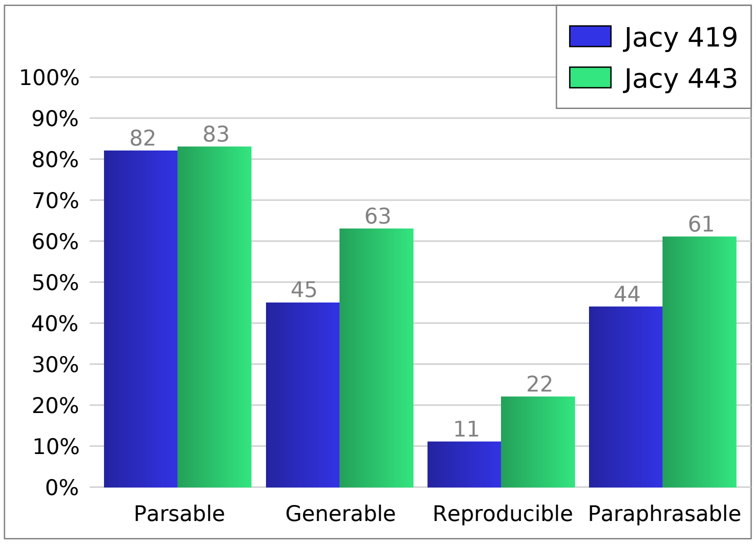

Figure 8.1: Initial and updated absolute general statistics for Jacy

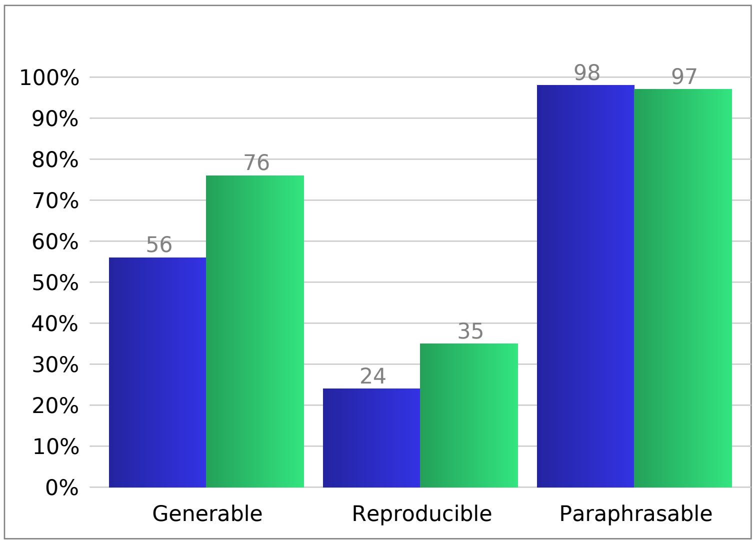

Figure 8.2: Initial and updated relative general statistics for Jacy

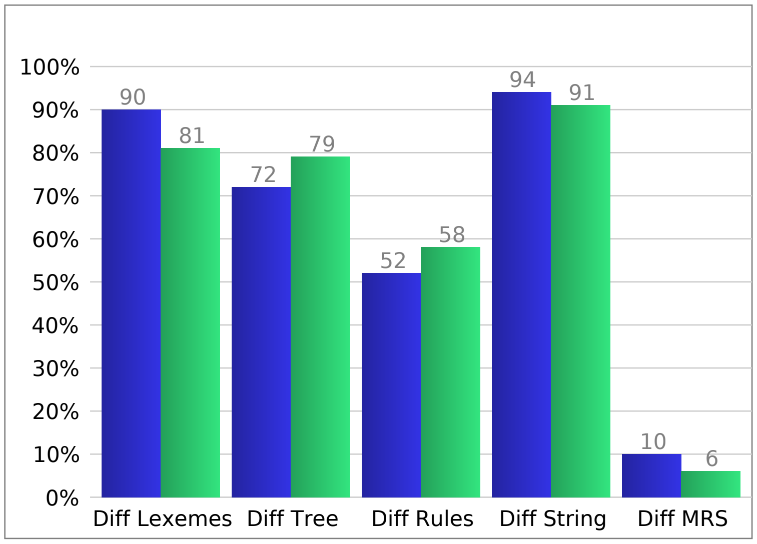

Figure 8.3: Initial and updated comparative statistics for Jacy

Chapter 9
CONCLUSION

I have described the background, methodology, implementation, and results of **Egad**, and explained how we used its results to improve the Jacy grammar. **Egad** has now been shown to be a useful tool for grammar developers, but that is not to say it is finished. In this chapter I cover some ideas for future work to the system, then provide some concluding remarks.

9.1 Future Work

We think it could be beneficial to analyze more characteristics of the grammars. With more points of reference, a grammar developer can make even more informed decisions about what problems to fix. In particular, we have considered adding characteristics for performance-related factors, such as the amount of time or space needed to produce a parse or realization. This could allow us to search for rules causing inefficiencies in the grammar. In this case, perhaps a ratio of time/space to sentence length would be more appropriate. We have also considered getting information about failed parses from the parser, such as error messages or derivation tree fragments from partial parses. Lastly, we would like to try replacing lexical ids (specific to a lexeme) with lexical types in the RPs, since all lexemes of the same type should behave identically. Generalizing over all lexemes of the same type could allow some erroneous lexical types to be noticed more easily by **Egad**.

We would like to improve the filtering of redundant RPs in the error-mining phase, as there is still too much redundancy in the output. The work of Sagot and de La Clergerie (2006) and de Kok et al. (2009) may be able to help with the filtering (or rather, more intelligent ranking). A more long-term goal would allow **Egad** to analyze the internals of the grammar and point out specific features within the grammar rules that are causing problems.

Some of the errors detected by **Egad** have simple fixes, and we believe there is room to explore methods of automatic error correction. In particular, ungenerable items that just require new trigger rules could be automatically fixed if we could predict the context into which the semantically empty lexical items need to be inserted.

We have not yet fixed all the errors identified by **Egad**. One of the high priority fixes includes making all of the pragmatic information available to the generator — this would cut out a lot of the over-generation. It would also make more interesting paraphrases possible: for example, changing the text genre from formal to informal or vice-versa.

Lastly, in order to increase adoption of **Egad**, we plan to create a proper software package and distribute it to other researchers who could benefit from it. This would require making the code more robust for running on different systems, documentation (including installation and usage instructions), and some more usability enhancements (such as reducing or eliminating any grammar-specific files that **Egad** requires, and providing a graphical user interface).

9.2 Availability

The **Egad** sources and documentation for the current version are available on the DELPH-IN wiki at `http://wiki.delph-in.net/moin/EgadTop`. A bug tracker and code repository for the next version of **Egad** are available at `https://launchpad.net/egad`.

9.3 Conclusion

We have introduced a system that characterizes the capabilities of a grammar by looking at its parsing and generation output, identifies errors in implemented HPSG grammars, finds and ranks the possible sources of those problems, and enables a grammar developer to easily find occurences of these errors. This tool can greatly reduce the amount of time a grammar developer would spend finding bugs, and helps them make informed decisions about which bugs are best to fix. Using our system, we were able to improve Jacy's absolute generation coverage by 18% (45% to 63%), and double the absolute percentage of reproducible items (from 11% to 22%), with only four weeks of grammar development.

BIBLIOGRAPHY

Timothy Baldwin, John Beavers, Emily M. Bender, Dan Flickinger, Ara Kim, and Stephan Oepen. 2005. Beauty and the beast: What running a broad-coverage precision grammar over the bnc taught us about the grammarand the corpus. *Linguistic Evidence: Empirical, Theoretical, and Computational Perspectives*, pages 49–70.

Emily M. Bender, Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pages 8–14. Taipei, Taiwan.

Francis Bond, Eric Nichols, Darren Scott Appling, and Michael Paul. 2008. Improving statistical machine translation by paraphrasing the training data. In *International Workshop on Spoken Language Translation*, pages 150–157. Honolulu.

Francis Bond, Stephan Oepen, Melanie Siegel, Ann Copestake, and Dan Flickinger. 2005. Open source machine translation with DELPH-IN. In *Open-Source Machine Translation: Workshop at MT Summit X*, pages 15–22. Phuket.

Ulrich Callmeier. 2000. PET - a platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99–108.

John Carroll and Stephan Oepen. 2005. High efficiency realization for a wide-coverage unification grammar. *LECTURE NOTES IN COMPUTER SCIENCE*, 3651:165.

Ann Copestake. 2002. *Implementing Typed Feature Structure Grammars*. CSLI Publications.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. 2005. Minimal Recursion Semantics. An introduction. *Research on Language and Computation*, 3(4):281–332.

Daniël de Kok, Jianqiang Ma, and Gertjan van Noord. 2009. A generalized method for iterative error mining in parsing results. In *Grammar Engineering Across Frameworks (GEAF 2009)*.

Markus Dickinson and Chong Min Lee. 2008. Detecting errors in semantic annotation. In *Proceedings of the Sixth International Language Resources and Evaluation (LREC'08)*. Marrakech, Morocco.

Rebecca Dridan and Francis Bond. 2006. Sentence comparison using robust minimal recursion semantics and an ontology. In *Proceedings of the Workshop on Linguistic Distances*, pages 35–42. Sydney. URL `http://www.aclweb.org/anthology/W/W06/W06-1106`.

Dan Flickinger. 2000. On building a more efficient grammar by exploiting types. *Natural Language Engineering*, 6(1):15–28. (Special Issue on Efficient Processing with HPSG).

Dan Flickinger. 2008. The English resource grammar. Technical Report 2007-7, LOGON, `http://www.emmtee.net/reports/7.pdf`. (Draft of 2008-11-30).

Dan Flickinger, Alexander Koller, and Stafan Thater. 2005. A new well-formedness criterion for semantics debugging. In *Proceedings of the 12th International Conference on HPSG*, page 129142. URL `http://cslipublications.stanford.edu/HPSG/6/abstr-hb.shtml`.

Claire Gardent and Eric Kow. 2007. Spotting overgeneration suspects. In *11th European Workshop on Natural Language Generation*, page 41.

Alexander Koller and Stefan Thater. 2005. Efficient solving and exploration of scope ambiguities. *Interactive Poster and Demonstration Sessions*, page 9.

Yuji Matsumoto, Kitauchi, Yamashita, Hirano, Matsuda, and Asahara. 2000. *Nihongo Keitaiso Kaiseki System: Chasen*. `http://chasen.naist.jp/hiki/ChaSen/`.

Eric Nichols, Francis Bond, and Daniel Flickinger. 2005. Robust ontology acquisition from machine-readable dictionaries. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-2005*, pages 1111–1116. Edinburgh.

Stephan Oepen and John Carroll. 2000. Performance profiling for grammar engineering. *Natural Language Engineering*, 6(1):81–97.

Stephan Oepen, Dan Flickinger, and Francis Bond. 2004. Towards holistic grammar engineering and testing — grafting treebank maintenance into the grammar revision cycle. In *Beyond Shallow Analyses — Formalisms and Statistical Modelling for Deep Analysis (Workshop at IJCNLP-2004)*. Hainan Island. URL `http://www-tsujii.is.s.u-tokyo.ac.jp/bsa/`.

Stephan Oepen, Erik Velldal, Jan Tore Løning, Paul Meurer, and Victoria Rosen. 2007. Towards hybrid quality-oriented machine translation. on linguistics and probabilities in MT. In *11th International Conference on Theoretical and Methodological Issues in Machine Translation: TMI-2007*, pages 144–153.

Martha Palmer, Dan Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1).

Carl Pollard and Ivan A. Sag. 1994. *Head Driven Phrase Structure Grammar*. University of Chicago Press, Chicago.

Benoît. Sagot and Éric. de La Clergerie. 2006. Error mining in parsing results. In *Annual Meeting-Association For Computational Linguistics*, volume 44, page 329.

Melanie Siegel. 2000a. HPSG analysis of Japanese. In Wolfgang Wahlster, editor, *Verbmobil: Foundations of Speech-to-Speech Translation*, pages 265 – 280. Springer, Berlin, Germany.

Melanie Siegel. 2000b. Japanese honorification in an HPSG framework. In *14th PACLIC*, pages 289–300. Tokyo.

Yasuhito Tanaka. 2001. Compilation of a multilingual parallel corpus. In *Proceedings of PACLING 2001*, pages 265–268. Kyushu. (`http://www.colips.org/afnlp/archives/pacling2001/pdf/tanaka.pdf`).

Kristina. Toutanova, Penka Markova, and Christopher Manning. 2004. The leaf projection path view of parse trees: Exploring string kernels for hpsg parse selection. In *Proceedings of EMNLP*, pages 166–173.

Hans Uszkoreit. 2002. New chances for deep linguistic processing. In *19th International Conference on Computational Linguistics: COLING-2002*, pages XIV–XXVII. Taipei.

Gertjan van Noord. 2004. Error mining for wide-coverage grammar engineering. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics Morristown, NJ, USA.

Appendix A
PROBLEMATIC CHARACTERISTICS

Here I describe a problem **Egad** had with detecting the similarity of derivation trees. In order to fix this problem, we have considered reintroducing (or not removing, at least) stems and lexical ids into the derivation trees during comparison.

Figures A.1–A.2 show two different trees that **Egad** judged to be the same. Both sentences mean “The rain changed into snow.” The order of the arguments is reversed in the second sentence, but it is a valid paraphrase. **Egad** failed to notice the difference because it only looks at phrasal nodes, and in the Jacy grammar, both から *kara* “from” and に *ni* “to” use the same phrasal rules to combine with their complements. While this problem is not terribly common, it happens more often than we would like. In order to fix this, we would have to allow the derivation tree comparisons to analyze lexical information.

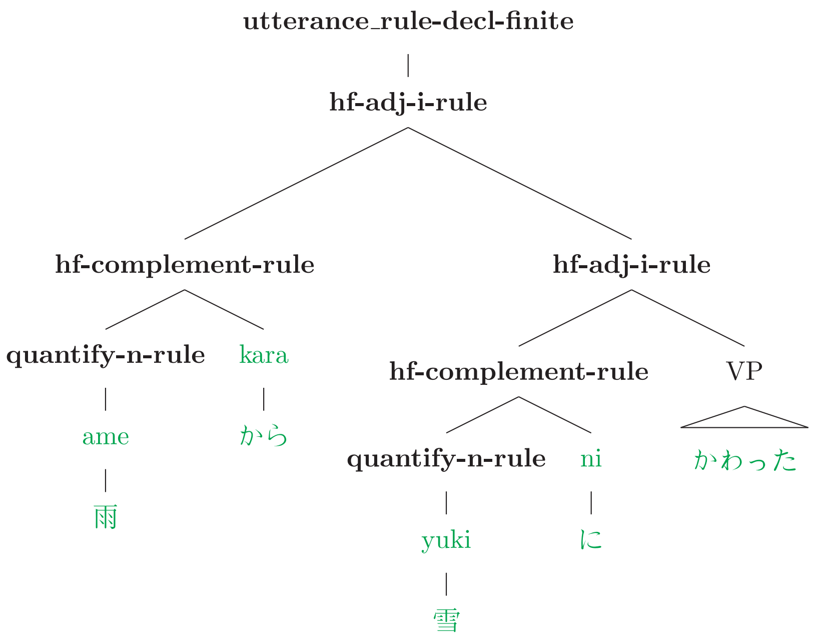

Figure A.1: Derivation Tree for “雨 から 雪 に かわっ た”

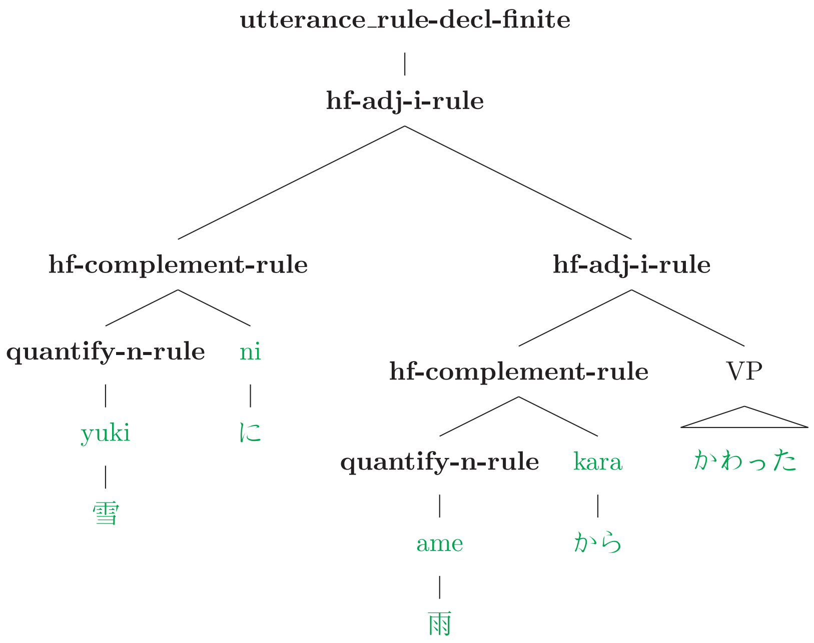

Figure A.2: Derivation Tree for “雪 に 雨 から かわっ た”