Thesis for the Degree of Doctor of Philosophy

Formal Methods for Testing Grammars

Inari Listenmaa

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2019

Formal Methods for Testing Grammars
INARI LISTENMAA

ISBN 978-91-7833-322-6
Technical Report 168D

Research groups: Functional Programming and Language Technology
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Typeset in Palatino and Monaco by the author using XeTeX
Printed at Chalmers reproservice
Gothenburg, Sweden 2019

Abstract

Grammar engineering has a lot in common with software engineering. Analogous to a program specification, we use descriptive grammar books; in place of unit tests, we have gold standard corpora and test cases for manual inspection. And just like any software, our grammars still contain bugs: grammatical sentences that are rejected, ungrammatical sentences that are parsed, or grammatical sentences that get the wrong parse.

This thesis presents two contributions to the analysis and quality control of computational grammars of natural languages. Firstly, we present a method for finding *contradictory grammar rules* in Constraint Grammar, a robust and low-level formalism for part-of-speech tagging and shallow parsing. Secondly, we generate *minimal and representative test suites* of example sentences that cover all grammatical constructions in Grammatical Framework, a multilingual grammar formalism based on deep structural analysis.

Keywords: *Grammatical Framework, Constraint Grammar, Satisfiability, Test Case Generation, Grammar Analysis*

Acknowledgements

I am grateful to my supervisor Koen Claessen for his support and encouragement. I cherish our hands-on collaboration, which often involved sitting in the same room writing code together. You have been there for all aspects of a PhD, both inspiration and perspiration! My co-supervisor Aarne Ranta is an equally important figure in my PhD journey, and a major reason I decided to pursue a PhD in the first place. The reader of this thesis may thank Aarne for making me rewrite certain sections over and over again, until they started making sense! Furthermore, I want to thank my opponent Fred Karlsson, and Torbjörn Lager, Gordon Pace and Laurette Pretorius for agreeing to be in my grading committee, as well as Graham Kemp for being my examiner.

Looking back, I have an enormous gratitude the whole GF community. It all started back in 2010 when I was a master's student in Helsinki and joined a research project led by Lauri Carlson. Lauri Alanko was most helpful office mate when I was learning GF, Kaarel Kaljurand was my first co-author and the Estonian resource grammar we wrote was my first large-scale GF project. As important as it was to learn from others during my first steps, becoming a teacher myself has been even more enlightening. I am happy to have met and guided newer GF enthusiasts, especially Bruno Cuconato and Kristian Kankainen—I may have learned more from your questions than you from my answers!

During my PhD studies, I've had the chance to collaborate with several people outside Gothenburg and my research group. I want to thank Eckhard Bick, Tino Didriksen and Francis Tyers for introducing me to CG and keeping up with my questions and weird ideas. I am grateful to Jose Mari Arriola and Maxux Aranzabe for hosting my research visit in the Basque country and giving me a chance to work with the coolest language I've seen so far—eskerrik asko!

In addition to my roles as a student, teacher and colleague, I've enjoyed more unstructured exploration among peers, and pursuing interesting side tracks. Wen Kokke deserves special thanks for making such an unlikely collaboration happen; as well as for reading a number of nonsensical test sentences in Dutch, and for everything else!

On a more day-to-day basis, I want to thank all my colleagues at the department. My office mates Herb and Prasanth have shared with me joys and frustrations, and helped me to decipher supervisor comments. In addition to my supervisors and office mates, I thank the rest of the language technology group: Grégoire, John, Krasimir, Peter, Ramona and Thomas.

Outside my office and research group, I want to thank Anders, Dan, Daniel, Elena, Irene, Gabriel, Guilhem, Simon H., Simon R. and Víctor for all the fun things during the 5 years: interesting lunch discussions, fermentation parties, hairdyeing parties, climbing, board games, forest excursions, playing music together, sharing a houseboat—just to name a few things. (Also it's handy to have a stack of your old theses in my office to use as an inspiration for writing acknowledgements!)

Sometimes it's also fun to meet people outside research! I want to thank the wonderful people in Kulturkrock and Chalmers sångkör for making me feel at home in Sweden, not just in the small bubble of my research group. Ett extra tack till Anka och Jonatan för att ni rättat min svenska!

All the factors I've mentioned previously have been important in finishing my PhD. Listing all the factors that enabled me to start a PhD would take more space than the actual monograph, so let me be brief and thank my parents, firstly, for making me exist, and secondly, for raising me to believe in myself, be willing to take challenges, and never stop learning.

This work has been carried out within the REMU project — Reliable Multilingual Digital Communication: Methods and Applications. The project was funded by the Swedish Research Council (*Vetenskapsrådet*) under grant number 2012-5746.

Contents

Chapter 1

Introduction to this thesis

There are many approaches to natural language processing (NLP). Broadly divided, we can contrast data-driven and rule-based approaches, which in turn contain more subdivisions.

Data-driven NLP is based on learning from examples, rather than explicit rules. Consider how would a system learn that the English string *cat* corresponds to the German string *Katze*: we feed it millions of parallel sentences, and it learns that whenever *cat* appears in one, *Katze* appears in another with a high probability. This approach is appropriate, when the target language is well-resourced (and grammatically simple), the domain is unlimited and correctness is not crucial. For example, a monolingual English speaker who wants to get the gist of any non-English web page will be happy with a fast, wide-coverage translation system, even if it makes some mistakes.

Rule-based NLP has a different set of use cases. Writing grammar rules takes more human effort than training a machine learning model, but in many cases, it is the more feasible approach:

- **Quality over coverage.** Think of producers instead of consumers of information: the producer needs high guarantees of correctness, but often there is only a limited domain to cover.
- **Less need for language resources**. Most of the 6000+ languages of the world do not have the abundance of data needed for machine learning, making rule-based approach the only feasible one.
- **Grammatically complex languages** benefit proportionately more from a grammar-based approach. Think of a grammar as a compression method: a compact set of rules

generates infinitely many sentences. Grammars can also be used in conjunction with machine learning, e.g. creating new training data to prevent data sparsity.

- **Grammars are explainable**, and hence, testable. If there is a bug in a particular sentence, we can find the reason for it and fix it. In contrast, machine learning models are much more opaque, and the user can just tweak some parameters or add some data, without guarantees how it affects the model.

Testing grammars has one big difference from testing most software: natural language has no formal specification, so ultimately we must involve a human oracle. However, we can automate many useful subtasks: detect *ambiguous constructions* and *contradictory grammar rules*, as well as generate *minimal and representative* set of examples that cover all the constructions. Think of the whole grammar as a haystack, and we suspect there are a few needles—we cannot promise automatic needle-removal, but instead we help the human oracle to narrow down the search.

Our work is by no means the first approach to grammar testing: for instance, Butt et al. (1999) recommend frequent use of test suites, as well as extensive documentation of grammars. However, we automate the process further than our predecessors, using established techniques from the field of software testing, and applying them to a novel use case: natural language grammars.

In the following sections, we present our research on testing computational natural language grammars, showcasing two different types of grammar formalisms and testing methods. The first type of grammars are *reductionistic*: the starting point is all the words in a given sentence, with all possible analyses: for example, in the sentence "can you can the can", every *can* has 3 analyses, and *you* has 2. The grammar rules are constraints, whose task is to get rid of inappropriate analyses. The second type grammars are *generative*: the starting point is an empty string, and the grammar is a set of rules that generate all grammatical sentences in the language. For example, "I saw a dog" would be in the set, because there are rules that construct just that sequence, but "saw dog I a" would not, because there are no rules to construct it.

1.1 Symbolic evaluation of a reductionistic formalism

Constraint Grammar (CG) (Karlsson et al., 1995) is a robust and language-independent formalism for part-of-speech tagging and shallow parsing. A grammar consists of disambiguation rules for initially ambiguous, morphologically analysed text: the correct analysis for the sentence is attained by removing improper readings from ambiguous words. Consider the word *wish*: without context, it could be a noun or a verb. But any English speaker can easily tell that "a wish" is a noun. In CG, we can generalise the observation and write the following rule:

```
SELECT noun IF (0 noun/verb) (-1 article) ;
```

Wide-coverage CG grammars, containing some thousands of rules, generally achieve very high accuracy: thanks to the flexible formalism, new rules can be written to address even the rarest phenomena, without compromising the general tendencies. But there is a downside to this flexibility: large grammars are difficult to maintain and debug. There is no internal mechanism that prevents rules from contradicting each other—the following is a syntactically valid CG grammar, and normal CG compilers will not detect anything strange.

```
SELECT noun IF (-1 article) ;
SELECT verb IF (-1 article) ;
```

Apart from obvious cases as above, with two rules saying contradicting truths about the language, there can be indirect conflicts between the rules. Take the following example:

```
REMOVE article ;
SELECT noun IF (-1 article) ;
```

The first rule removes all articles unconditionally, thus rendering the second rule invalid: it can never apply, because its condition is never matched.

In real life, these conflicts can be much more subtle, and appear far apart from each other. The common way of testing grammars is to apply them into some test corpus, with a gold standard, and gather statistics how often the rules apply. While this method can reveal that `SELECT noun IF (-1 article)` never applied, it cannot tell whether it is just because no sentence in the test corpus happened to trigger the contextual test, or whether some other rule before it made it impossible to ever apply.

We use a method called *symbolic evaluation*: in high level terms, we pretend to apply the grammar to every possible input, and track the consequences of each decision. The rules become interconnected, and we can find the reason for a conflict. This allows us to answer questions such as "given this set of 50 rules, is there an input that goes through the first 49 rules and still triggers the 50th rule?"

1.2 Test case generation for a generative formalism

Grammatical Framework (GF) (Ranta, 2011) is a generative, multilingual grammar formalism with a high level of abstraction. As opposed to CG, where the rules *reduce* the number of possible readings, GF and other generative formalisms use rules to *generate* a language from scratch. In other words, the language corresponding to a grammar is defined as the set of strings that the grammar rules generate.

The most typical example of a generative grammar is a context-free grammar. GF is a more complex formalism, and we will present it in Section 2.2. To quickly illustrate the properties of a generative grammar, we use the following context-free grammar G:

```
           Sentence ::= Noun Verb | Noun "says that" Sentence
 G =       Noun     ::= "John" | "Mary"
           Verb     ::= "eats" | "sleeps"
```

We call `Sentence` as the start category: all valid constructions in the language G are sentences. There are two ways of forming sentences: one is to combine a `Noun` and a `Verb`, which results in the finite set {*John eats, John sleeps, Mary eats, Mary sleeps*}. However, the second way of constructing a sentence is recursive, resulting in an infinite amount of sentences: *John says that Mary says that …Mary eats*.

Again, the grammars we are really interested in are much larger, and the GF formalism has more complex machinery in place, such as taking care of agreement, i.e. allowing *John sleeps* and *I sleep*, but rejecting **I sleeps* and **John sleep*. But the basic mechanism is the same: a formal, rather compact description that generates a potentially infinite set of sentences.

It is fully possible to write a "bad" generative grammar, in the sense that it produces sentences that any English speaker would deem grammatically incorrect. But there is no notion of internal inconsistency, like there was in CG—the question is simply "does this grammar describe the English language adequately?". In order to test this, we must go beyond the grammar itself, and find a human oracle to answer the question. But here we face a problem: what kind of subset of the grammar should we test? How do we know if we have tested enough?

The problem is relevant for any kind of software testing, and we use existing techniques of test case generation, applied to the GF formalism. For each language, we generate a minimal and representative set of example sentences, which we give to native or fluent speakers to judge.

1.3 Structure of this thesis

The thesis is divided in two parts: one about Constraint Grammar (Chapters 3 and 4) and another about Grammatical Framework (Chapter 5). The core of Chapters 3 and 4 is based on two articles, which present the SAT-encoding and its application to grammar testing: "Constraint Grammar as a SAT problem" (Listenmaa and Claessen, 2015) and "Analysing Constraint Grammars with a SAT-solver" (Listenmaa and Claessen, 2016). A third article, "Cleaning up the Basque grammar: a work in progress" (Listenmaa et al., 2017), presents a practical application of the method on a Basque CG, and is included in the evaluation section. A fourth article, "Exploring the Expressivity of Constraint Grammar" (Kokke and Listenmaa, 2017), presents a novel use for the SAT-encoding.

Chapter 5 is an extension of one article, "Automatic test suite generation for PMCFG grammars" (Listenmaa and Claessen, 2018).

For both parts, some of the content has been updated since the initial publication; in addition, the implementation is described in much more detail. The thesis is structured as a stand-alone read; however, a reader who is familiar with the background, may well skip Chapter 2.

Chapter 2 presents a general introduction to both software testing and computational grammars, aimed for a reader who is unfamiliar with the topics. Chapter 3 introduces the SAT-encoding of Constraint Grammar, with two different schemes for conflict handling. Chapter 4 describes the method of analysing CG by using symbolic evaluation, along with evaluation on three different grammars. Chapter 5 presents the method of generating test cases for GF grammars, along with evaluation on a large-scale resource grammar and a couple of smaller grammars. The user manual of the program is included an appendix. Chapter 6 concludes the thesis.

1.4 Contributions of the author

The three articles which present the general methods (Listenmaa and Claessen, 2015, 2016, 2018) are co-written by the author and Koen Claessen. In general, the ideas behind the publications were joint effort. For the first article (Listenmaa and Claessen, 2015), all of the implementation of SAT-CG is by the author, and all of the library SAT+ by Koen Claessen. The version appearing in this monograph, Chapter 3, is thoroughly rewritten by the author since the initial publication.

For the second article (Listenmaa and Claessen, 2016), the author of this work was in charge of all the implementation, except for the ambiguity class constraints, which were written by Koen Claessen. The version appearing in Chapter 4 is to large extent the same as the original article, with added detail and examples throughout the chapter. The chapter incorporates material from two additional articles: "Exploring the Expressivity of Constraint Grammar" (Kokke and Listenmaa, 2017), which is joint work between the author and Wen Kokke, and "Cleaning up the Basque grammar: a work in progress" (Listenmaa et al., 2017), co-authored with Jose Maria Arriola, Itziar Aduriz and Eckhard Bick. In (Kokke and Listenmaa, 2017), the initial idea was developed together, implementation was split in approximately half, and writing was done together. In (Listenmaa et al., 2017), the author was in a main role in both the implementation and writing.

The work in the third article (Listenmaa and Claessen, 2018) was joint effort: both authors participated in all parts of planning and implementation. Both the original article and the version appearing in this monograph, Chapter 5, was mainly written by the author.

Chapter 2

Background

Much of the methodology of grammar testing is dictated by common sense.
Miriam Butt et al.
A Grammar Writer's Cookbook

In this chapter, we present the main areas of this thesis: computational natural language grammars and software testing. Section 2.1 introduces the CG formalism and some of the different variants, along with the historical context and related work. For a description of any specific CG implementation, we direct the reader to the original source: CG-1 (Karlsson, 1990; Karlsson et al., 1995), CG-2 (Tapanainen, 1996), VISL CG-3 (Bick and Didriksen, 2015; Didriksen, 2014). Section 2.2 introduces the GF formalism (Ranta, 2011) and the Resource Grammar Library (Ranta, 2009).

Section 2.3 is a brief and high-level introduction to software testing and verification, aimed for a reader with no prior knowledge on the topic. For a more thorough introduction on software testing in general, we recommend Ammann and Offutt (2016), and for SAT in particular, we recommend Biere (2009).

2.1 Constraint Grammar

Constraint Grammar (CG) is a formalism for disambiguating morphologically analysed text. It was first introduced by Karlsson (1990), and has been used for many tasks in computational linguistics, such as part-of-speech tagging, surface syntax and machine translation

(Bick, 2011). CG-based taggers are reported to achieve F-scores of over 99 % for morphological disambiguation, and around 95-97 % for syntactic analysis (Bick, 2000, 2003, 2006). CG disambiguates morphologically analysed input by using constraint rules which can select or remove a potential analysis (called *reading*) for a target word, depending on the context words around it. Together these rules disambiguate the whole text.

In the example below, we show an initially ambiguous sentence "the bear sleeps". It contains three word forms, such as `"<bear>"`, each followed by its *readings*. A reading contains one lemma, such as `"bear"`, and a list of morphological tags, such as `noun sg`. A word form together with its readings is called a *cohort*. A cohort is ambiguous, if it contains more than one reading.

```
"<the>"                          "<sleeps>"
       "the" det def                    "sleep" noun pl
"<bear>"                                "sleep" verb pres p3 sg
       "bear" noun sg            "<.>"
       "bear" verb pres                 "." sent
       "bear" verb inf
```

We can disambiguate this sentence with two rules:

1. `REMOVE verb IF (-1 det)` 'Remove verb after determiner'
2. `REMOVE noun IF (-1 noun)` 'Remove noun after noun'

Rule 1 matches the word *bear*: it is tagged as verb and is preceded by a determiner. The rule removes both verb readings from *bear*, leaving it with an unambiguous analysis `noun sg`. Rule 2 is applied to the word *sleeps*, and it removes the noun reading. The finished analysis is shown below:

```
"<the>"              "<bear>"              "<sleeps>"
     "the" det def        "bear" noun sg        "sleep" verb pres p3 sg
```

It is also possible to add syntactic tags and dependency structure within CG (Didriksen, 2014; Bick and Didriksen, 2015). However, for the remainder of this introduction, we will illustrate the examples with the most basic operations, that is, disambiguating morphological tags. The syntactic disambiguation and dependency features are not fundamentally different from morphological disambiguation: the rules describe an *operation* performed on a *target*, conditional on a *context*.

2.1.1 Related work

CG is one in the family of shallow and reductionist grammar formalisms. Disambiguation using constraint rules dates back to 1960s and 1970s—the closest system to modern CG was Taggit (Greene and Rubin, 1971), which was used to tag the Brown Corpus. Karlsson et al. (1995) list various approaches to the disambiguation problem, including manual intervention, statistical optimisation, unification and Lambek calculus. For disambiguation rules based on local constraints, Karlsson mentions (Herz and Rimon, 1991; Hindle, D., 1989).

CG itself was introduced in 1990. Around the same time, a related formalism was proposed: finite-state parsing and disambiguation system using constraint rules (Koskenniemi, 1990), which was later named Finite-State Intersection Grammar (FSIG) (Piitulainen, 1995). Like CG, a FSIG grammar contains a set of rules which remove impossible readings based on contextual tests, in a parallel manner: a sentence must satisfy all individual rules in a given FSIG. Due to these similarities, the name Parallel Constraint Grammar was also suggested (Koskenniemi, 1997). Other finite-state based systems include Gross (1997) and Graña et al. (2002). In addition, there are a number of reimplementations of CG using finite-state methods (Yli-Jyrä, 2011; Hulden, 2011; Peltonen, 2011).

Brill tagging (Brill, 1995) is based on transformation rules: the starting point of an analysis is just one tag, the most common one, and subsequent rule applications transform one tag into another, based on contextual tests. Like CG, Brill tagging is known to be efficient and accurate. The contextual tests are very similar: Lager (2001) demonstrates a systen that automatically learns both Brill rules and CG rules. Similar ideas to CG have been also explored in other frameworks, such as logic programming (Oflazer and Tür, 1997; Lager, 1998), constraint satisfaction (Padró, 1996), and dependency syntax (Tapanainen and Järvinen, 1997).

2.1.2 Properties of Constraint Grammar

Karlsson et al. (1995) list 24 design principles and describes related work at the time of writing. Here we summarise a set of main features, and relate CG to the developments in grammar formalism since the initial description.

CG is a *reductionistic* system: the analysis starts from a list of alternatives, and removes those which are impossible or improbable. CG is designed primarily for analysis, not generation; its task is to give correct analyses to the words in given sentences, not to describe a language as a collection of "all and only the grammatical sentences".

The syntax is decidedly *shallow*: the rules do not aim to describe all aspects of an abstract phenomenon such as noun phrase; rather, each rule describes bits and pieces with concrete conditions. The rules are self-contained and do not, in general, refer to other rules. This makes it easy to add exceptions, and exceptions to exceptions, without changing the more general rules.

There are different takes on how *deterministic* the rules are. The current state-of-the-art CG parser VISL CG-3 executes the rules strictly based on the order of appearance, but there are other implementations which apply their own heuristics, or remove the ordering completely, applying the rules in parallel. A particular rule set may be written with one application order in mind, but another party may run the grammar with another implementation—if there are any conflicting rule pairs, then the behaviour of the grammar is different.

2.1.3 Ordering and execution of the rules

The previous properties of Constraint Grammar formalism and rules were specified by Karlsson et al. (1995), and retained in further implementations. However, in the two decades following the initial specification, several independent implementations have experimented with different ordering schemes. In the present section, we describe the different parameters of ordering and execution: *strict vs. heuristic*, and *sequential vs. parallel*. Throughout the section, we will apply the rules to the following ambiguous passage, "*What question is that*":

```
"<what>"             "<question>"             "<is>"              "<that>"
       "what" det           "question" noun          "be" verb           "that" det
       "what" pron          "question" verb                              "that" rel
```

Strict vs. heuristic aka. "In which order are the rules applied to a single cohort?"
An implementation with a *strict order* applies each rule in the order in which they appear in the file. Suppose that a grammar contains the following rules the given order:

```
REMOVE verb IF (-1 det)
REMOVE noun IF (-1 pron)
```

In a strict ordering, the rule that removes the verb reading in *question* will be applied first. After it has finished, there are no verb readings available anymore for the second rule to fire.

How do we know which rule is the right one? There can be many rules that fit the context, but we choose the one that just happens to appear first in the rule file. A common design pattern is to place rules with a long list of conditions first; only if they do not apply, then try a rule with fewer conditions. For a similar effect, a *careful mode* may be used: "remove verb after *unambiguous* determiner" would not fire on the first round, but it would wait for other rules to clarify the status of *what*.

An alternative solution to a strict order is to use a *heuristic order*: when disambiguating a particular word, find the rule that has the longest and most detailed match. Now, assume that there is a rule with a longer context, such as `SELECT noun IF (-1 det) (1 verb)`, even if this rule appears last in the file, it would be preferred to the shorter rules, because it is a more exact match. There are also methods that use explicit weights to favour certain rules, such as Pirinen (2015) for CG, and Voutilainen (1994); Oflazer and Tür (1997); Silfverberg and Lindén (2009) for related formalisms.

Both methods have their strengths and weaknesses. A strict order is more predictable, but it also means that the grammar writers need to pay more thought to rule interaction. A heuristic order frees the grammar writer from finding an optimal order, but it can give unexpected results, which are harder to debug. As for major CG implementations, CG-1 (Karlsson, 1990) and VISL CG-3 (Didriksen, 2014) follow the strict scheme, whereas CG-2 (Tapanainen, 1996) is heuristic[1].

Sequential vs. parallel aka. "When does the rule application take effect?"
The input sentence can be processed in sequential or parallel manner. In *sequential execution*, the rules are applied to one word at a time, starting from the beginning of the sentence. The sentence is updated after each application. If the word *what* gets successfully disambiguated as a pronoun, then the word *question* will not match the rule `REMOVE verb IF (-1 det)`.

In contrast, a *parallel execution* strategy disambiguates all the words at the same time, using their initial, ambiguous context. To give a minimal example, assume we have a single rule, `REMOVE verb IF (-1 verb)`, and the three words *can can can*, shown below. In parallel execution, both *can*$_2$ and *can*$_3$ lose their verb tag; in sequential only *can*$_2$.

```
"<can₁>"              "<can₂>"              "<can₃>"
      "can" noun            "can" noun            "can" noun
      "can" verb            "can" verb            "can" verb
```

The question of parallel execution becomes more complicated if there are multiple rules that apply for the same context. Both `REMOVE v IF (-1 det)` and `REMOVE noun IF (-1 pron)` would match *question*, because the original input from the morphological analyser contains both determiner and pronoun as the preceding word. The result depends on various details:

[1]Note that CG-2 is heuristic *within sections*: the rules in a given section are executed heuristically, but all of them will be applied before any rule in a later section.

	Strict	**Heuristic**	**Unordered**
Sequential	CG-1 (Karlsson, 1990) VISL CG-3 (Didriksen, 2014) Peltonen (2011) Yli-Jyrä (2011) Hulden (2011)	CG-2 (Tapanainen, 1996) Weighted CG-3 (Pirinen, 2015)	–
Parallel	SAT-CG$_{\text{Ord}}$	SAT-CG$_{\text{Max}}$ *FSIG (Voutilainen, 1994)* *Voting constraints (Oflazer and Tür, 1997)*	*Lager (1998)* *FSIG (Koskenniemi, 1990)*

Table 2.1: Combinations of rule ordering and execution strategy.

shall all the rules also act in parallel? If we allow rules to be ordered, then the result will not be any different from the same grammar in sequential execution; that is, the later rule (later by any metric) will not apply. The only difference is the reason why not: "context does not match" in sequential, and "do not remove the last reading" in parallel.

However, usually parallel execution is combined with *unordered* rules. In order to express the result of these two rules in an unordered scheme, we need a concept that has not been discussed so far, namely, disjunction: "the allowed combinations are *det+n* or *pron+v*". If we wanted to keep the purely list-based ontology of CG, but combine it with a parallel and unordered execution, then the result would have to inconclusive and keep both readings; both cannot be removed because that would leave *question* without any readings. The difference between the list-based and disjunction-based ontologies, corresponding to CG and FSIG respectively, is explained with further detail in Lager and Nivre (2001).

Table 2.1 shows different systems of the constraint rule family, with rule order (strict vs. heuristic) on one axis, and execution strategy (sequential vs. parallel) on other. Traditional CG implementations are shown in a normal font; other, related systems in cursive font and lighter colour. SAT-CG$_{\text{Max}}$and SAT-CG$_{\text{Ord}}$refer to the systems by the author; they are presented in Listenmaa and Claessen (2015) and in Chapter 3 of this thesis.

2.2 Grammatical Framework

Grammatical Framework (GF) (Ranta, 2011) is a framework for building multilingual grammar applications. Its main components are a functional programming language for writing grammars and a resource library that contains the linguistic details of many natural languages. A GF program consists of an *abstract syntax* (a set of functions and their categories) and a set of one or more *concrete syntaxes* which describe how the abstract functions and categories are turned into surface strings in each respective concrete language. The resulting grammar describes a mapping between concrete language strings and their corresponding abstract trees (structures of function names). This mapping is bidirectional—strings can be *parsed* to trees, and trees *linearised* to strings. As an abstract syntax can have multiple corresponding concrete syntaxes, the respective languages can be automatically *translated* from one to the other by first parsing a string into a tree and then linearising the obtained tree into a new string.

Another main component of GF is the Resource Grammar Library (RGL) (Ranta, 2009), which, as of October 2018, contains a range of linguistic details for over 40 natural languages. The library has had over 50 contributors, and it consists of 1900 program modules and 3 million lines of code. As the name suggests, the RGL modules are primarily used as libraries to build smaller, domain-specific application grammars. In addition, there is experimental work on using the RGL as an interlingua for wide-coverage translation, aided by statistical disambiguation (Ranta et al., 2014).

2.2.1 Related work

GF comes from the theoretical background of type theory and logical frameworks. The prime example of a system which combines logic and linguistic syntax is Montague grammar (Montague, 1974); in fact, GF can be seen as a general framework for Montague-style grammars.

The notion of abstract and concrete syntax appeared in both computer science, specifically compiler construction (McCarthy, 1962), and linguistics, introduced by Curry (1961) as *tectogrammatical* (abstract) and *phenogrammatical* (concrete) structure.

GF is analogous to a multi-source multi-target compiler—a program in any programming language can be parsed into the common abstract syntax, and linearised into any of the other programming languages that the compiler supports. In the domain of linguistics, Ranta (2011) mentions a few grammar formalisms that also build upon abstract and concrete syntax, such as de Groote (2001); Pollard (2004); Muskens (2001). However, none of these systems have focused on multilinguality. The inspiration for a large resource grammar used as a library to build smaller applications comes from CLE (Core Language Engine) (Alshawi, 1992; Rayner et al., 2000).

```
abstract Foods = {
  flags startcat = Comment ;
  cat
    Comment ; Item ; Kind ; Quality ;
  fun
    Pred : Item -> Quality -> Comment ;               -- this wine is good
    This, That, These, Those : Kind -> Item ;         -- this wine
    Mod : Quality -> Kind -> Kind ;                   -- Italian wine
    Wine, Cheese, Fish, Pizza : Kind ;
    Warm, Good, Italian, Vegan : Quality ;
 }
```

Figure 2.1: Abstract syntax of a GF grammar about food

2.2.2 Abstract syntax

Abstract syntax describes the constructions in a grammar without giving a concrete implementation. Figure 2.1 shows the abstract syntax of a small example grammar in GF, slightly modified from Ranta (2011), and Figure 2.2 shows a corresponding Spanish concrete syntax. We refer to this grammar throughout the chapter.

Section `cat` introduces the categories of the grammar: `Comment`, `Item`, `Quality`, and `Kind`. `Comment` is the *start category* of the grammar: this means that only comments are complete constructions in the language, everything else is an intermediate stage. `Quality` describes properties of foods, such as `Warm` and `Good`. `Kind` is a basic type for foodstuffs such as `Wine` and `Pizza`: we know what it is made of, but everything else is unspecified. In contrast, an `Item` is *quantified*: we know if it is singular or plural (e.g. 'one pizza' vs. 'two pizzas'), definite or indefinite ('the pizza' vs. 'a pizza'), and other such things ('your pizza' vs. 'my pizza').

Section `fun` introduces functions: they are either lexical items without arguments, or syntactic functions which manipulate their arguments and build new terms. Of the syntactic functions, `Pred` constructs an `Comment` from an `Item` and a `Quality`, building trees such as `Pred (This Pizza) Good` 'this pizza is good'. `Mod` adds an `Quality` to a `Kind`, e.g. `Mod Italian Pizza` 'Italian pizza'. The functions `This, That, These` and `Those` quantify a `Kind` into an `Item`, for instance, `That (Mod Italian Pizza)` 'that Italian pizza'.

```
concrete FoodsSpa of Foods = {
  lincat
    Comment = Str ;
    Item = { s : Str ; n : Number ; g : Gender } ;
    Kind = { s : Number => Str ; g : Gender } ;
    Quality = { s : Number => Gender => Str ; p : Position } ;
  lin
    Pred np ap = np.s ++ copula ! np.n ++ ap.s ! np.n ! np.g ;
    This cn = mkItem Sg "este" "esta" cn ;
    These cn = mkItem Pl "estos" "estas" cn ;
    -- That, Those defined similarly
    Mod ap cn = { s = \\n => preOrPost ap.p (ap.s ! n ! cn.g) (cn.s ! n) ;
                  g = cn.g } ;
    Wine = { s = table { Sg => "vino" ; Sg => "vinos" } ;
             g = Masc } ;
    Pizza = { s = table { Sg => "pizza" ; Sg => "pizzas" } ;
              g = Fem } ;
    Good = { s = table { Sg => table { Masc => "bueno" ; Fem => "buena" } ;
                         Pl => table { Masc => "buenos" ; Fem => "buenas" } } ;
             p = Pre } ;
    --Fish, Cheese, Italian, Warm and Vegan defined similarly
  param
    Number = Sg | Pl ;
    Gender = Masc | Fem ;
    Position = Pre | Post ;
  oper
    mkItem num mascDet femDet cn =
     let det = case cn.g of { Masc => mascDet ; Fem => femDet } ;
      in { s = det ++ cn.s ! num ; n = num ; g = cn.g } ;
    copula = table { Sg => "es" ; Pl => "son" } ;
    preOrPost p x y = case p of { Pre => x ++ y ; Post => y ++ x } ;
 }
```

Figure 2.2: Spanish concrete syntax of a GF grammar about food

2.2.3 Concrete syntax

Concrete syntax is an implementation of the abstract syntax. The section `lincat` corresponds to `cat` in the abstract syntax: for every abstract category introduced in `cat`, we give a concrete implementation in `lincat`.

Figure 2.2 shows the Spanish concrete syntax, in which `Comment` is a string, and the rest of the categories are more complex records. For instance, `Kind` has a field `s` which is a table from number to string (SG $\Rightarrow$ *pizza*, PL $\Rightarrow$ *pizzas*), and another field `g`, which contains its gender (feminine for `Pizza`). We say that `Kind` has *inherent* gender, and *variable* number.

The section `lin` contains the concrete implementation of the functions, introduced in `fun`. Here we handle language-specific details such as agreement: when `Pred (This Pizza) Good` is linearised in Spanish, 'esta pizza es buena', the copula must be singular (*es* instead of plural *son*), and the adjective must be in singular feminine (*buena* instead of masculine *bueno* or plural *buenas*), matching the gender of `Pizza` and the number of `This`. If we write an English concrete syntax, then only the number of the copula is relevant: this pizza/wine *is* good, these pizzas/wines *are* good.

2.2.4 PMCFG

GF grammars are compiled into parallel multiple context-free grammars (PMCFG), introduced by Seki et al. (1991). The connection between GF and PMCFG was established by Ljunglöf (2004), and further developed by Angelov (2011). After the definition, which follows Angelov (2011), we explain three key features for the test suite generation.

Definition A PMCFG is a 5-tuple:

$$G = \langle N^C, F^C, T, P, L \rangle.$$

- N^C is a finite set of concrete categories.
- F^C is a finite set of concrete functions.
- T is a finite set of terminal symbols.
- P is a finite set of productions of the form:

  $$A \rightarrow f[A_1, A_2, \ldots, A_{a(f)}]$$

  where $a(f)$ is the arity of f, $A \in N^C$ is called result category, $A_1, A_2, \ldots, A_{a(f)} \in N^C$ are called argument categories and $f \in F^C$ is a function symbol.
- $L \subset N^C \times F^C$ is a set which defines the default linearisation functions for those concrete categories that have default linearisations.

Concrete categories For each category in the original grammar, the GF compiler introduces a new *concrete category* in the PMCFG for each combination of inherent parameters. These concrete categories can be linearised to strings or vectors of strings. The start category (`Comment` in the Foods grammar) is in general a single string, but intermediate categories may have to keep several options open.

Consider the categories `Item`, `Kind` and `Quality` in the Spanish concrete syntax. Firstly, `Item` has inherent number and gender, so it compiles into four concrete categories: $\texttt{Item}_{\texttt{sg,masc}}$, $\texttt{Item}_{\texttt{sg,fem}}$, $\texttt{Item}_{\texttt{pl,masc}}$ and $\texttt{Item}_{\texttt{pl,fem}}$, each of them containing one string. Secondly, `Kind` has an inherent gender and variable number, so it compiles into two concrete categories: $\texttt{Kind}_{\texttt{masc}}$ and $\texttt{Kind}_{\texttt{fem}}$, each of them a vector of two strings (singular and plural). Finally, `Quality` needs to agree in number and gender with its head, but it has its position as an inherent feature. Thus `Quality` compiles into two concrete categories: $\texttt{Quality}_{\texttt{pre}}$ and $\texttt{Quality}_{\texttt{post}}$, each of them a vector of four strings.

Concrete functions Each syntactic function from the original grammar turns into multiple syntactic functions into the PMCFG: one for each combination of parameters of its arguments.

- $\texttt{Mod}_{\texttt{pre,fem}}$: $\texttt{Quality}_{\texttt{pre}} \to \texttt{Kind}_{\texttt{fem}} \to \texttt{Kind}_{\texttt{fem}}$
- $\texttt{Mod}_{\texttt{post,fem}}$: $\texttt{Quality}_{\texttt{post}} \to \texttt{Kind}_{\texttt{fem}} \to \texttt{Kind}_{\texttt{fem}}$
- $\texttt{Mod}_{\texttt{pre,masc}}$: $\texttt{Quality}_{\texttt{pre}} \to \texttt{Kind}_{\texttt{masc}} \to \texttt{Kind}_{\texttt{masc}}$
- $\texttt{Mod}_{\texttt{post,masc}}$: $\texttt{Quality}_{\texttt{post}} \to \texttt{Kind}_{\texttt{masc}} \to \texttt{Kind}_{\texttt{masc}}$

Coercions As we have seen, `Quality` in Spanish compiles into $\texttt{Quality}_{\texttt{pre}}$ and $\texttt{Quality}_{\texttt{post}}$. However, the difference of position is meaningful only when the adjective is modifying the noun: "la *buena* pizza" vs. "la pizza *vegana*". But when we use an adjective in a predicative position, both classes of adjectives behave the same: "la pizza es *buena*" and "la pizza es *vegana*". As an optimization strategy, the grammar creates a *coercion*: both $\texttt{Quality}_{\texttt{pre}}$ and $\texttt{Quality}_{\texttt{post}}$ may be treated as $\texttt{Quality}_{*}$ when the distinction doesn't matter. Furthermore, the function $\texttt{Pred} : \texttt{Item} \to \texttt{Quality} \to \texttt{S}$ uses the coerced category $\texttt{Quality}_{*}$ as its second argument, and thus expands only into 4 variants, despite there being 8 combinations of $\texttt{Item} \times \texttt{Quality}$.

- $\texttt{Pred}_{\texttt{sg,fem},*}$: $\texttt{Item}_{\texttt{sg,fem}} \to \texttt{Quality}_{*} \to \texttt{Comment}$
- $\texttt{Pred}_{\texttt{pl,fem},*}$: $\texttt{Item}_{\texttt{pl,fem}} \to \texttt{Quality}_{*} \to \texttt{Comment}$
- $\texttt{Pred}_{\texttt{sg,masc},*}$: $\texttt{Item}_{\texttt{sg,masc}} \to \texttt{Quality}_{*} \to \texttt{Comment}$
- $\texttt{Pred}_{\texttt{pl,masc},*}$: $\texttt{Item}_{\texttt{pl,masc}} \to \texttt{Quality}_{*} \to \texttt{Comment}$

2.3 Software testing and verification

We can approach the elimination of bugs in two ways: reveal them by constructing tests, or build safeguards into the program that make it more difficult for bugs to occur in the first place. In this thesis, we concentrate on the first aspect: our starting point is two particular grammar formalisms that already have millions of lines of code written in them. Rather than change the design of the programming languages, we want to develop methods that help finding bugs in existing software. In the present section, we introduce some key concepts from the field of software testing, as well as their applications to grammar testing.

Unit testing Unit tests are particular, concrete test cases: assume we want to test the addition function (+), we could write some facts we know, such as "1+2 should be 3". In the context of grammars and natural language, we could assert translation equivalence between a pair of strings, e.g. "*la casa grande*" ⇔ "*the big house*", or between a tree and its linearisation, e.g. "`DetCN the_Det (AdjCN big_A house_N)` ⇔ *the big house*". Whenever a program is changed or updated, the collection of unit tests are run again, to make sure that the change has not broken something that previously worked.

Property-based testing The weakness of unit testing is that it only tests concrete values that the developer has thought of adding. Another approach is to use property-based testing: the developer defines abstract properties that should hold for all test cases, and uses random generation to supply values. If we want to test the (+) function again, we could write a property that says "for all integers x and y, $x + y$ should be equal to $y + x$". A grammar-related property could be, for instance, "a linearisation of a tree must contain strings from all of its subtrees". We formulate these properties, and generate a large amount of test data—pairs of integers in the first case, syntax trees in the second—and assert that the property holds for all of them.

Model-based testing The programs we test are often large and complex, and their main logic is hard to separate from less central components, such as graphical user interface, or code dealing with input and output. In order to make testing easier, we can introduce a *model* of the real system, which we use to devise tests. To give a concrete example, suppose we want to test a goal-line system: a program that checks whether a goal has been scored or not, in a game such as football. The real program includes complex components such as cameras and motion sensors, but our model can abstract away such details, and consider only the bare bones: the ball and the goal line. For our model, we define possible *states* for the ball and goal line: the ball can be inside or outside the line, and the goal line may be idle or in action. Furthermore, we define legal *transitions* between the states: the ball starts outside, can stay outside, or go inside, but once it has entered the goal, it must get out before another goal can be scored.

With such a model in place, we can query another program, called *model checker*, for possible outcomes in the model, or ask whether a particular outcome would be possible. For our goal-line system, we would like to be assured that it is possible to score a goal; on the other hand, when a ball has entered the goal, it is only registered as one goal, not repeated goals every millisecond until the ball is removed.

What do we gain from building such a model? We describe the (intended) behaviour of the real program in the form of a simplified model, and ask for logical conclusions: sometimes we discover that our original description wasn't quite as waterproof as we wanted. Of course, we still need to execute tests for the real system—the model (together with model checker) has only given us inspiration what to test.

There is no great theoretical distinction between model-based and property-based testing. A model can be seen as a specific variant of a property; alternatively, a collection of properties can be seen as a model. Furthermore, we can see grammars as models for language: what is a grammar if not a human's best attempt to describe the behaviour of a complex system?

In Chapter 4, we will take the approach of a model checker to the individual Constraint Grammar rules: "you claimed that *X* happens, now let's try applying *X* in all possible contexts", where the contexts are exhaustively created combinations of morphological readings, not real text. The goal of this procedure is to find out whether the grammar rules contradict each other. The desired outcomes are therefore very simple: for each rule, the outcome is "this rule can apply, given that all other rules have applied", and the task of our program is to find out if such an outcome is possible.

In Chapter 5, we are interested in the much more difficult outcome "this grammar rule produces correct language". It is clear that a model checker cannot check such a high-level concept as grammatical correctness. Therefore, we need humans as the ultimate judges of quality. Because human time is expensive, we need to make sure that the test cases are as minimal and representative as possible.

Deriving test cases Unit tests, as well as properties, can be written by a human or derived automatically from some representation of the program. The sources of tests can range from informal descriptions, such as specifications or user stories, to individual statements in the source code. Alternatively, tests can be generated from an abstract model of the program, as previously explained.

In the context of grammar testing, the specification is the whole natural language that the grammar approximates—hardly a formal and agreed-upon document. Assuming no access to the computational grammar itself (generally called *black-box testing*), we can treat traditional grammar books or speaker intuition as an inspiration for properties and unit tests. For example, we can test a feature such as "pronouns must be in an accusative case after a preposition" by generating example sentences where pronouns occur after a preposition, and reading through them to see if the rule holds.

If we have access to the grammar while designing tests (*white-box testing*), we can take advantage of the structure—to start with, only test those features that are implemented in the grammar. For example, if we know that word order is a parameter with 2 values, direct statement and question, then we need to create 2 tests, e.g. "I am old" and "am I old". If the parameter had 3 values, say third for indirect question, then we need a third test as well, e.g. "I don't know if I am old".

Coverage criteria Beizer (2003) describes testing as a simple task: "all a tester needs to do is find a graph and cover it". The flow of an imperative program can be modelled as a graph with start and end points; multiple branches at conditional statements and back edges at loops. Take a simple program that takes a number and outputs "even" for even numbers and "odd" for odd numbers. The domain of this program is, in theory, infinite, as there is an infinite number of integers. But in practice, the program only has two branches, one for even and one for odd numbers. Thus in order to test the program exhaustively, i.e. cover both paths, one needs to supply two inputs: one even and one odd number.

Simulating the run of the program with all feasible paths is called *symbolic evaluation*, and a constraint solver is often used to find out where different inputs lead into. It is often not feasible to simulate all paths for a large and complex program; instead, several heuristics have been created for increasing code coverage.

Symbolic evaluation works well for analysing (ordered) CG grammars, at least up to an input space of tens of thousands of different morphological analyses. The range of operations is fairly limited, and the program flow is straightforward: execute rule 1, then execute rule 2, and so on.

For GF grammars, the notion of code coverage is based on individual grammatical functions, rather than a program flow. We want to test linearisation, not parsing, and thus our "inputs" are just syntax trees. We need to test all syntactic functions (e.g. putting an adjective and a noun together), with all the words that make a difference in the output (some adjectives come before the noun, others come after). Thus, even if the grammar generates an infinite amount of sentences, we still have only a finite set of constructions and grammatical functions to cover.

2.4 Boolean satisfiability (SAT)

Imagine you are in a pet shop with a selection of animals: *ant*, *bat*, *cat*, *dog*, *emu* and *fox*.

These animals are very particular about each others' company. The dog has no teeth and needs the cat to chew its food. The cat, in turn, wants to live with its best friend bat. But the bat is very aggressive towards all herbivores, and the emu is afraid of anything lighter than 2 kilograms. The ant hates all four-legged creatures, and the fox can only handle one flatmate with wings.

You need to decide on a subset of pets to buy—you love all animals, but due to their restrictions, you cannot have them all. You start calculating in your head: "If I take the ant, I cannot take cat, dog, nor fox. How about I take the dog, then I must take the cat and the bat as well." After some time, you decide on bat, cat, dog and fox, leaving the ant and the emu in the pet shop.

Definition This conveniently contrived situation is an example of *Boolean satisfiability (SAT)* problem. The animals translate into *variables*, and the cohabiting restrictions of each animal translate into *clauses*, such that "dog wants to live with cat" becomes an implication $dog \Rightarrow cat$. Under the hood, all of these implications are translated into even simpler constructs: lists of disjunctions. For instance, "dog wants to live with cat" as a disjunction is $\neg dog \vee cat$, which means "I don't buy the dog or I buy the cat". The representation as disjunctions is easier to handle algorithmically; however, for the rest of this thesis, we show our examples as implications, because they are easier to understand. The variables and the clauses are shown in Figure 2.3.

The objective is to find a *model*: each variable is assigned a Boolean value, such that the conjunction of all clauses evaluates into true. A program called *SAT-solver* takes the set of variables and clauses, and performs a search, like the mental calculations of the animal-lover in the shop. We can see that the assignment $\{ant = 0, bat = 1, cat = 1, dog = 1, emu = 0, fox = 1\}$ satisfies the animals' wishes. Another possible assignment would be $\{ant = 0, bat = 0, cat = 0, dog = 0, emu = 1, fox = 1\}$: you only choose the emu and the fox. Some problems

Variable	Constraint	Explanation
	$ant \vee bat \vee cat \vee dog \vee fox$	"You want to buy at least one pet."
ant	$ant \Rightarrow \neg cat \wedge \neg dog \wedge \neg fox$	"Ant does not like four-legged animals."
bat	$bat \Rightarrow \neg ant \wedge \neg emu$	"Bat does not like herbivores."
cat	$cat \Rightarrow bat$	"Cat wants to live with bat."
dog	$dog \Rightarrow cat$	"Dog wants to live with cat."
emu	$emu \Rightarrow \neg ant \wedge \neg bat$	"Emu does not like small animals."
fox	$fox \Rightarrow \neg(bat \wedge emu)$	"Fox cannot live with two winged animals."

Figure 2.3: Animals' cohabiting constraints translated into a SAT-problem.

have a single solution, some problems have multiple solutions, and some are unsatisfiable, i.e. no combination of assignments can make the formula true.

History and applications SAT-solving as a research area dates back to 1970s. Throughout its history, it has been of interest for both theoretical and practical purposes. SAT is a well-known example of an *NP-complete* (Nondeterministic Polynomial time) problem (Cook, 1971): for all such problems, a potential solution can be *verified* in polynomial time, but there is no known algorithm that would *find* such a solution, in general case, in sub-exponential time. This equivalence means that we can express any NP-complete problem as a SAT-instance, and use a SAT-solver to solve it. The class includes problems which are much harder than the animal example; nevertheless, all of them can be reduced into the same representation, just like $\neg bat \vee \neg emu$.

The first decades of SAT-research were concentrated on the theoretical side, with little practical applications. But things changed in the 90s: there was a breakthrough in the SAT-solving techniques, which allowed for scaling up and finding new use cases. As a result, modern SAT-solvers can deal with problems that have hundreds of thousands of variables and millions of clauses (Marques-Silva, 2010).

What was behind these advances? SAT, as a general problem, remains NP-complete: it is true that there are still SAT-problems that cannot be solved in sub-exponential time. However, there is a difference between a general case, where the algorithm must be prepared for any input, and an "easy case", where we can expect some helpful properties from the input. Think of a sorting algorithm: in the general case, it is given truly random lists, and in the "easy case", it mostly gets lists with some kind of structure, such as half sorted, reversed, or containing bounded values. The general time complexity of sorting is still $O(n\, log\, n)$, but arguably, the easier cases can be expected to behave in linear time, and we can even design heuristic sorting algorithms that exploit those properties.

Analogously, the 90s breakthrough was due to the discovery of right kind of heuristics. Much of the SAT-research in the last two decades has been devoted to optimising the solving algorithms, and finding more efficient methods of encoding various real-life problems into SAT. This development has led to an increasing amount of use cases since the early 2000s (Claessen et al., 2009). One of the biggest success stories for a SAT-application is model checking (Sheeran and Stålmarck, 1998; Biere et al., 1999; Bradley, 2011), used in software and hardware verification. Furthermore, SAT has been used in domains such as computational biology (Claessen et al., 2013) and AI planning (Selman and Kautz, 1992), just to pick a few examples. In summary, formulating a decision problem in SAT is an attractive approach: instead of developing search heuristics for each problem independently, one can transform the problem into a SAT-instance and exploit decades of research into SAT-solving.

2.5 Summary

In this section, we have presented the theoretical background used in this thesis. We have introduced Constraint Grammar and Grammatical Framework as examples of computational grammars. On the software testing side, we have presented key concepts such as unit testing and Boolean satisfiability. In the following chapters, we will connect the two branches. Firstly, we encode CG grammars as a SAT-problem, which allows us to apply symbolic evaluation and find potential conflicts between rules. Secondly, we use methods for creating minimal and representative test data, in order to find a set of trees that test a given GF grammar in an optimal way.

Chapter 3

CG as a SAT-problem

This explicitly reductionistic approach does not seem to have any obvious counterparts in the grammatical literature.
Fred Karlsson, 1995

You should do it because it solves a problem, not because your supervisor has a fetish for SAT.
Koen Claessen, 2016

In this chapter, we present CG as a Boolean satisfiability (SAT) problem, and describe an implementation using a SAT-solver. This is attractive for several reasons: formal logic is well-studied, and serves as an abstract language to reason about the properties of CG. Despite the wide adoption of the formalism, there has never been a single specification of all the implementation details of CG, particularly the rule ordering and the execution strategy. Furthermore, the translation into a SAT-problem makes it possible to detect conflicts between rules—we will see an application for grammar analysis in Chapter 4.

Applying logic to reductionist grammars has been explored earlier by Lager (1998) and Lager and Nivre (2001), but there has not been, to our knowledge, a full logic-based CG implementation; at the time, logic programming was too slow to be used for tagging or parsing. Since those works, SAT-solving techniques have improved significantly (Marques-Silva, 2010), and they are used in domains such as microprocessor design and computational biology—these problems easily match or exceed CG in complexity. In addition, SAT-solving brings us more practical tools, such as maximisation, which enables us to implement a novel conflict resolution method for parallel CG.

The content in this chapter is based on "Constraint Grammar as a SAT problem" (Listenmaa and Claessen, 2015). As in the original paper, we present a translation of CG rules into logical formulas, and show how to encode it into a SAT-problem. This work is implemented as an open-source software SAT-CG[1]. It uses the high-level library SAT+[2], which is based on MiniSAT (Eén and Sörensson, 2004). We evaluate SAT-CG against the state of the art, VISL CG-3. The experimental setup is the same, but we ran the tests again for this thesis: since the writing of Listenmaa and Claessen (2015), we have optimised our program and fixed some bugs; this makes both execution time and F-scores better than we report in the earlier paper.

3.1 Related work

Our work is inspired by previous approaches of encoding CG in logic (Lager, 1998; Lager and Nivre, 2001). Lager (1998) presents a "CG-like, shallow and reductionist system" translated into a disjunctive logic program. Lager and Nivre (2001) build on that in a study which reconstructs four different formalisms in first-order logic. CG is contrasted with Finite-State Intersection Grammar (FSIG) (Koskenniemi, 1990) and Brill tagging (Brill, 1995); all three work on a set of constraint rules which modify the initially ambiguous input, but with some crucial differences. On a related note, Yli-Jyrä (2001) explores the structural correspondence between FSIG and constraint-solving problems. In addition, logic programming has been applied for automatically inducing CG rules from tagged corpora (Eineborg and Lindberg, 1998; Sfrent, 2014; Lager, 2001).

3.2 CG as a SAT-problem

In this section, we translate the disambiguation of a sentence into a SAT-problem. We demonstrate our encoding with an example in Spanish, shown in Figure 3.1: *la casa grande*. The first word, *la*, is ambiguous between a definite article ('the') or an object pronoun ('her'), and the second word, *casa*, can be a noun ('house') or a verb ('(he/she) marries'). The subsegment *la casa* alone can be either a noun phrase, $la_{\textsc{Det}}$ $casa_{\text{N}}$ 'the house' or a verb phrase $la_{\textsc{Prn}}$ $casa_{\text{V}}$ '(he/she) marries her'. However, the unambiguous adjective, *grande* ('big'), disambiguates the whole segment into a noun phrase: 'the big house'. Firstly, we translate input sentences

[1] `https://github.com/inariksit/cgsat`
[2] `https://github.com/koengit/satplus`

into variables and rules into clauses. Secondly, we disambiguate the sentence by asking for a solution. Finally, we consider different ordering schemes and conflict handling.

3.2.1 Encoding the input

Original analysis	Variables	Default rule
`"<la>"`		“do not remove the last reading”
`"el" det def f sg`	la_{Det}	
`"lo" prn p3 f sg`	la_{Prn}	$la_{\text{Det}} \vee la_{\text{Prn}}$
`"<casa>"`		
`"casa" n f sg`	$casa_{\text{N}}$	
`"casar" v pri p3 sg`	$casa_{\text{V}}$	$casa_{\text{N}} \vee casa_{\text{V}}$
`"<grande>"`		
`"grande" adj mf sg`	$grande_{\text{Adj}}$	$grande_{\text{Adj}}$

Figure 3.1: Ambiguous segment in Spanish: translation into SAT-variables.

Reading The readings of the word forms make a natural basis for variables. We translate a combination of a word form and a reading, such as `"<la>" ["el" det def f sg]`, into a variable la_{Det}, which represents the possibility that *la* is a determiner. This example segment gives us five variables: $\{la_{\text{Det}}, la_{\text{Prn}}, casa_{\text{N}}, casa_{\text{V}}, grande_{\text{Adj}}\}$, shown in 3.1.

Cohort As in the original input, the readings are grouped together in cohorts. We need to keep this distinction, for instance, to model SELECT rules and cautious context: `SELECT "casa" n` means, in effect, “remove $casa_{\text{V}}$”, and `IF (-1C prn)` means “if la_{Prn} is true and la_{Det} false”. Most importantly, we need to make sure that the last reading is not removed. Hence we add the default rule, “do not remove the last reading”, as shown in the third column of 3.1. These disjunctions ensure that at least one variable in each cohort must be true.

Sentence In order to match conditions against analyses, the input needs to be structured as a sentence: the cohorts must follow each other like in the original input, indexed by their absolute position in the sentence. Thus when we apply `REMOVE v IF (-1 det)` to the cohort $2 \to [casa_{\text{N}}, casa_{\text{V}}]$, the condition will match on la_{Det} in cohort 1.

Rule Next, we formulate a rule in SAT. A single rule, such as `REMOVE v IF (-1 det)`, is a template for forming an implication; when given a concrete sentence, it will pick concrete variables by the following algorithm.

1. Match rule against all cohorts

   la: No target found

   casa: Target found in $casa_{\text{V}}$, match conditions to *la*

   – Condition found in la_{Det}

   – Create a clause: $la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}$ 'if *la* is a determiner, *casa* is not a verb'

   grande: No target found

2. Solve with all clauses: $\{\overbrace{la_{\text{Det}} \vee la_{\text{Prn}},\ casa_{\text{N}} \vee casa_{\text{V}},\ grande_{\text{Adj}}}^{\text{given by the default rule}},\ \overbrace{la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}}^{\texttt{REMOVE v IF (-1 det)}}\}$

In Section 3.3 , we have included a translation of all the rule types that SAT-CG supports: REMOVE and SELECT rules, with most of the operations from CG-2 (Tapanainen, 1996), and a couple of features from VISL CG-3 (Didriksen, 2014). The following examples in this section do not require reading Section 3.3.

3.2.2 Applying a rule

Finally, we have all we need to disambiguate the segment: the sentence and the constraints encoded as SAT-variables and clauses. The SAT-solver returns a model that satisfies all the clauses presented in step 2. We started off with all the variables unassigned, and required at least one variable in each cohort to be true. In addition, we gave the clause $la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}$. We can see with a bare eye that this problem will have a solution; in fact, multiple ones, shown in Figure 3.2. The verb analysis is removed in the first two models, as required by the presence of la_{Det}. However, the implication may as well be interpreted "if $casa_{\text{V}}$ may not follow la_{Det}, better remove la_{Det} instead"; this has happened in Models 3–4. We see a third interpretation in Model 5: $casa_{\text{V}}$ may be removed even without the presence of la_{Det}. This is possible, because $la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}$ is only an implication, not an equivalence.

It seems like SAT-CG does worse than any standard CG implementation: the latter would just remove the verb, not give 5 different interpretations for a single rule. In fact, the rule `REMOVE v IF (-1 det)` alone behaves exactly like `REMOVE det IF (1 v)`. But there is power to this property. Now, we add a second rule: `REMOVE n IF (-1 prn)`, which will form the clause $la_{\text{Prn}} \Rightarrow \neg casa_{\text{N}}$. The new clause prohibits the combination $la_{\text{Prn}}\ casa_{\text{N}}$, which rules out three models out of five. The disambiguation is shown in Figure 3.3.

After two rules, we only have two models: one with $la_{\text{Det}}\ casa_{\text{N}}$ and other with $la_{\text{Prn}}\ casa_{\text{V}}$. In fact, we have just implemented parallel CG (PCG), introduced in Section 2.1.3: the rules act

Model 1	**Model 2**	**Model 3**	**Model 4**	**Model 5**
la_{DET}	la_{DET}			
	la_{PRN}	la_{PRN}	la_{PRN}	la_{PRN}
$casa_{\mathrm{N}}$	$casa_{\mathrm{N}}$	$casa_{\mathrm{N}}$		$casa_{\mathrm{N}}$
		$casa_{\mathrm{V}}$	$casa_{\mathrm{V}}$	
$grande_{\mathrm{ADJ}}$	$grande_{\mathrm{ADJ}}$	$grande_{\mathrm{ADJ}}$	$grande_{\mathrm{ADJ}}$	$grande_{\mathrm{ADJ}}$

Figure 3.2: Possible models for `REMOVE v IF (-1 det)`.

Model 1	**Model 2**
la_{DET}	
	la_{PRN}
$casa_{\mathrm{N}}$	
	$casa_{\mathrm{V}}$
$grande_{\mathrm{ADJ}}$	$grande_{\mathrm{ADJ}}$

Figure 3.3: Possible models for `REMOVE v IF (-1 det)` and `REMOVE n IF (-1 prn)`.

in parallel, and if the sentence cannot be fully disambiguated, the remaining uncertainty is modelled as a disjunction of all possible combinations of readings. In contrast, a sequential CG (SCG) engine applies each rule individually, and it cannot handle disjunction; its only operation is to manipulate lists of readings in a cohort. The SCG engine would have just applied one of the rules—say, the first one, removed the verb and stopped there. If another rule later in the sequence removes the determiner, there is no way to restore the verb.

To finish our PCG example, let us add one more rule: `REMOVE v IF (1 adj)`, and the corresponding clause $grande_{\mathrm{ADJ}} \Rightarrow \neg casa_{\mathrm{V}}$. This clause will rule out Model 2 of Figure 3.3, and we will get Model 1 as the unique solution. We can see another benefit in allowing connections between rules: none of the three rules has targeted *la*, still it has become unambiguous.

3.2.3 Solving conflicts in the parallel scheme

As described in Section 2.1.3, PCG behaves differently from SCG: the rules are dependent on each other, and the order does not matter. This prevents too hasty decisions, such as removing $casa_{\mathrm{V}}$ before we know the status of *la*. However, ignoring the order means that we miss significant information in the rule set. The truth is that pure PCG is very brittle: each and every rule in the set must fit together, without the notion of order. The rule sequence in Figure 3.4, taken from a Dutch grammar[3], will be well-behaved in an SCG with strict rule order. The grammar will behave as intended also in a heuristic variant of SCG, because the rules with a longer context are matched first. But in PCG, the rule set will definitely cause a conflict, rendering the whole grammar useless.

The order clearly demonstrates the course of action: "If a potential imperative starts a sentence and is followed by an object pronoun, select the imperative reading; then, move on to other rules; finally, if any imperative is still ambiguous, remove the imperative reading." Comparing the success of SCG to PCG in practical applications, one may speculate that the sequential order is easier to understand—undeniably, its behaviour is more transparent. If two rules target the same cohort, the first mentioned gets to apply, and removes the target. When the first rule has acted, the second rule is not even considered, because it would remove the last reading.

```
SECTION

  # Zeg me
  SELECT Imp IF (-1 BOS) (1 (prn obj)) ;

  # . Heb je
  SELECT (vbhaver pres p2 sg) IF (-1 BOS) (1 (prn obj uns p2 mf sg)) ;

  [--]

SECTION

  # remove all imperative readings that have not been explicitly selected
  REMOVE Imp ;

  # remove informal 2nd person singular reading of "heb"
  REMOVE (vbhaver pres p2 sg) ;
```

Figure 3.4: Example from a Dutch grammar

Ideally, both ways of grammar writing should yield similar results: sequential CG rules are more imperative, and parallel CG rules are more declarative. But the problem of conflicts in PCG still remains. In the following, we present two solutions: in the first one, we emulate

[3] https://github.com/apertium/apertium-nld/blob/master/apertium-nld.nld.rlx, commit d13fdaa

ordering in choosing which clauses to keep, and in the second one, we maximise the number of rule applications.

Emulating order We keep the parallel base, but use ordering as information for solving conflicts. This means that all the benefits of parallel execution still hold: the three rules, which all target *casa*, may still disambiguate *la*, without *la* ever being the target. If all the rules play well together, or if the earlier rules do not match any cohorts, then no rule applications need to be removed. However, if we have the grammar from Figure 3.4, and imperative is the right analysis for a given context, then the clauses created by `REMOVE Imp` would be ignored, in favour of the clauses that are created by `SELECT Imp IF (-1 BOS) (1 (prn obj))`.

In this modified scheme, we introduce the clauses to the SAT-solver one by one, and attempt to solve after each clause. If the SAT-problem after the 50^{th} rule has a solution, we accept all the clauses created by rule 50. If rule 51 causes a conflict, we prioritise the previous, well-behaving subset of 50 rules, and discard the conflicting clauses created by rule 51.

If a rule matches multiple cohorts, it creates a separate clause for each instance. Thus, it is no problem if the rule causes a conflict in only one cohort—say, we have another potential imperative in the sentence, but there is no other rule which targets its other readings. We can discard only the conflicting instances: we prevent `REMOVE Imp` from applying to *Zeg* in the sequence *# Zeg me*, but it still may apply to other ambiguous tokens with imperative reading.

Let us demonstrate the procedure with the Spanish segment *la casa grande*. Assuming our rule set is {`REMOVE v IF (-1 det)`, `REMOVE v IF (1 adj)`, `REMOVE n`}, the revised algorithm goes as follows:

1. Apply `REMOVE v IF (-1 det)`
   - Create a clause: $la_{\text{DET}} \Rightarrow \neg casa_{\text{V}}$
   - Solve with previous clauses: $\{\overbrace{la_{\text{DET}} \vee la_{\text{PRN}},\ casa_{\text{N}} \vee casa_{\text{V}},\ grande_{\text{ADJ}}}^{\text{default rule}},\ \overbrace{la_{\text{DET}} \Rightarrow \neg casa_{\text{V}}}^{\texttt{REMOVE v IF (-1 det)}}\}$
   - Solution found: add new clause to the formula
2. Apply `REMOVE v IF (1 adj)`
   - Create a clause: $grande_{\text{ADJ}} \Rightarrow \neg casa_{\text{V}}$
   - Solve with previous clauses: $\{\ldots,\ la_{\text{DET}} \Rightarrow \neg casa_{\text{V}},\ \overbrace{grande_{\text{ADJ}} \Rightarrow \neg casa_{\text{V}}}^{\texttt{REMOVE v IF (1 adj)}}\}$
   - Solution found: add new clause to the formula

3. Apply `REMOVE n`

   - Create a clause: $\neg casa_{\text{N}}$
   - Solve with previous clauses: $\{..., la_{\text{Det}} \Rightarrow \neg casa_{\text{V}},\ grande_{\text{Adj}} \Rightarrow \neg casa_{\text{V}},\ \overbrace{\neg casa_{\text{N}}}^{\texttt{REMOVE n}}\}$
   - No solution: discard clause

With this procedure, we use ordering to decide which clauses to include, and then apply all of them in parallel. After going through all the rules, the final formula to the SAT-solver will contain the clauses $la_{\text{Det}} \vee la_{\text{Prn}}$, $casa_{\text{N}} \vee casa_{\text{V}}$, $grande_{\text{Adj}}$, $la_{\text{Det}} \Rightarrow \neg\ casa_{\text{V}}$ and $grande_{\text{Adj}} \Rightarrow \neg\ casa_{\text{V}}$.

Maximisation Solving conflicts means that we have multiple rules that target the same reading, and we must choose which rule to apply. Strict ordering substitutes the question with a simpler one: "which rule comes first in the grammar?" Heuristic rule order asks "out of all the rules that target this cohort, which one has the best matching context?" If the competitors are `REMOVE n IF (-1 prn)` and `REMOVE v IF (-1 det) (1 adj)`, then the second one will win. However, if the rules are both as good a match, which happens in Figure 3.3, we need to resort to mere guessing, or fall back to ordering.

However, we can ask yet another question: "Out of all the rules that target this cohort, which one is a best fit *with other rules that will apply to this whole sentence*?" As opposed to heuristic or weighted approaches (Voutilainen, 1994; Oflazer and Tür, 1997), here all the individual rule applications are equally important; we just want to find the largest possible subset of rule applications that can act together without conflict. We will explain the procedure in the following.

Each rule application to a concrete cohort produces a clause, and the whole rule set applied to the whole sentence produces a large formula. In an ideal case, all the rules are well-behaved, and the whole formula is satisfiable. However, if the whole formula is unsatisfiable, we may still ask for an assignment that satisfies the maximum number of the clauses; that is, rule applications. If the grammar is good, we hope that the interaction between the appropriate rules would make a large set of clauses that fit together, and the inapplicable rule would not "fit in".

We keep the Spanish segment and the rule set {`REMOVE v IF (-1 det)`, `REMOVE v IF (1 adj)`, `REMOVE n` }. Now the procedure goes as follows:

1. Apply `REMOVE v IF (-1 det)`

- Create a clause: $la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}$

2. Apply `REMOVE v IF (1 adj)`
   - Create a clause: $grande_{\text{Adj}} \Rightarrow \neg casa_{\text{V}}$
3. Apply `REMOVE n`
   - Create a clause: $\neg casa_{\text{N}}$
4. Solve with all clauses: $\{\overbrace{la_{\text{Det}} \vee la_{\text{Prn}}, \ldots,}^{\text{default rule}}\ \overbrace{la_{\text{Det}} \Rightarrow \neg casa_{\text{V}},}^{\texttt{REMOVE v IF (-1 det)}}\ \overbrace{grande_{\text{Adj}} \Rightarrow \neg casa_{\text{V}},}^{\texttt{REMOVE v IF (1 adj)}}\ \overbrace{\neg casa_{\text{N}}}^{\texttt{REMOVE n}}\}$
5. No solution for all clauses: try to find a solution that satisfies maximally many rule applications; however, default rule cannot be overridden.

Similarly to the previous, order-based scheme, we create a clause for each instance of the rule application. In the case of a conflict, we can only discard the clauses targeting the offending cohort, but the rule may apply elsewhere in the sentence.

The problem of satisfying maximum amount of clauses is known as *Maximum Satisfiability* (MaxSAT). Whereas SAT is a decision problem, MaxSAT is an optimisation problem. However, optimisation can be expressed as a sequence of decision problems: first, we compute a solution, then we add a constraint "the solution must be better than the one just found", and ask for another one. This process repeats until a better solution cannot be found; then we accept the latest solution.

Now let us define how is one solution "better" than other, by using a simple trick. For each clause c, we create a new variable v. Instead of the original clause, we give the SAT-solver an implication $v \Rightarrow c$. This means that if v is false, the SAT-solver can ignore the actual clause c—the part that comes from the rule application. Conversely, if v is true, then the SAT-solver must handle the original clause. Then, we ask for a solution where maximally many of these vs are true, and the question for improvement becomes "can we make any more of the vs true"? The method of maximising the variables is described in Eén and Sörensson (2006).

As a alternative to creating a helper variable, we could also separate the variables into contexts and targets, and maximise the set of contexts: for $la_{\text{Det}} \Rightarrow \neg casa_{\text{V}}$ and $grande_{\text{Adj}} \Rightarrow \neg casa_{\text{V}}$, maximise the set of $\{la_{\text{Det}}, grande_{\text{Adj}}\}$. This variant would bring back the distinction between targets and contexts; given the design of most actually used CGs, it may be better suited for a practical implementation.

3.3 SAT-encoding

In this section, we show the SAT-encoding for all the rule types that SAT-CG supports: remove and select rules, with the contextual tests and set operations described in CG-2 (Tapanainen, 1996), except for the operators `**` (continue search if linked tests fail) and `@` (absolute position). In addition, the software SAT-CG supports a couple of new constructions from VISL CG-3: cbarrier, which requires the barrier tag to be unambiguous in order to function as a barrier; negate, which inverts a whole chain of contextual tests; and subreadings, which make it possible to include e.g. a clitic in the same cohort, but distinguish it from the main reading.

The encoding in this section is independent of the implementation of the software SAT-CG; it does not describe how the sentence is processed in order to find the variables. The purpose of this section is to give a full description of the different rule types; the method is introduced with more pedagogical focus in Section 3.2.

3.3.1 SAT-encoding of sentences

As in Section 3.2, we demonstrate the SAT-encoding with a concrete segment in Spanish: *sobre una aproximación más científica*. It has the virtue of being ambiguous in nearly every word: for instance, *sobre* is either a preposition ('above' or 'about') or a noun ('envelope'); *una* can be a pronoun, determiner or a verb. The full analysis, with the initial ambiguities, is shown in Figure 3.5.

```
"<sobre>"
            "sobre" pr                   "<aproximación>"
            "sobre" n m sg                                   "aproximación" n f sg
"<una>"                                  "<más>"
            "uno" prn tn f sg                                "más" adv
            "uno" det ind f sg                               "más" adj mf sp
            "unir" v prs p3 sg           "<científica>"
            "unir" v prs p1 sg                               "científico" adj f sg
            "unir" v imp p3 sg                               "científico" n f sg
```

Figure 3.5: Ambiguous segment in Spanish.

We transform each reading into a SAT-variable:

$$
\begin{array}{ccccc}
\textit{sobre}_{\textsc{Pr}} & \textit{una}_{\textsc{Prn}} & \textit{aproximación}_{\mathrm{N}} & \textit{más}_{\textsc{Adv}} & \textit{científica}_{\textsc{Adj}} \\
\textit{sobre}_{\mathrm{N}} & \textit{una}_{\textsc{Det}} & & \textit{más}_{\textsc{Adj}} & \textit{científica}_{\mathrm{N}} \\
 & \textit{una}_{\textsc{Prs}\mathrm{P3}} & & & \\
 & \textit{una}_{\textsc{Prs}\mathrm{P1}} & & & \\
 & \textit{una}_{\textsc{Imp}\mathrm{P3}} & & &
\end{array}
\tag{3.1}
$$

CG rules may not remove the last reading, even if the conditions hold otherwise. To ensure that each cohort contains at least one true variable, we add the clauses in 3.2; later referred to as the "default rule". The word *aproximación* is already unambiguous, thus the clause $\textit{aproximación}_{\mathrm{N}}$ is a unit clause, and the respective variable is trivially assigned true. The final assignment of the other variables depends on the constraint rules.

$$
\begin{array}{c}
\textit{sobre}_{\textsc{Pr}} \vee \textit{sobre}_{\mathrm{N}} \\
\textit{una}_{\textsc{Prn}} \vee \textit{una}_{\textsc{Det}} \vee \textit{una}_{\textsc{Prs}\mathrm{P3}} \vee \textit{una}_{\textsc{Prs}\mathrm{P1}} \vee \textit{una}_{\textsc{Imp}\mathrm{P3}} \\
\textit{aproximación}_{\mathrm{N}} \\
\textit{más}_{\textsc{Adv}} \vee \textit{más}_{\textsc{Adj}} \\
\textit{científica}_{\textsc{Adj}} \vee \textit{científica}_{\mathrm{N}}
\end{array}
\tag{3.2}
$$

3.3.2 SAT-encoding of rules

In order to demonstrate the SAT-encoding, we show variants of REMOVE and SELECT rules, with different contextual tests. We try to craft rules that make sense for this segment; however, some variants are not likely encountered in a real grammar, and for some rule types, we modify the rule slightly. We believe this makes the encoding overall more readable, in contrast to using more homogeneous but more artificial rules and input.

No conditions

The simplest rule types remove or select a target in all cases. A rule can target one or multiple readings in a cohort. We demonstrate the case with one target in rules 3.3–3.4, and multiple target in rules 3.5–3.6.

REMOVE adj Unconditionally removes all readings which contain the target.

$$\begin{gathered}\neg \textit{más}_{\textsc{Adj}} \\ \neg \textit{científica}_{\textsc{Adj}}\end{gathered} \tag{3.3}$$

SELECT adj Unconditionally removes all other readings which are in the same cohort with the target. We do not add an explicit clause for $\textit{más}_{\textsc{Adj}}$ and $\textit{científica}_{\textsc{Adj}}$: the default rule 3.2 already contain $\textit{más}_{\textsc{Adv}} \vee \textit{más}_{\textsc{Adj}}$ and $\textit{científica}_{\textsc{Adj}} \vee \textit{científica}_{\textsc{N}}$, and the combination $\neg \textit{más}_{\textsc{Adv}}$ and $\textit{más}_{\textsc{Adv}} \vee \textit{más}_{\textsc{Adj}}$ implies that $\textit{más}_{\textsc{Adj}}$ must be true.

$$\begin{gathered}\neg \textit{más}_{\textsc{Adv}} \\ \neg \textit{científica}_{\textsc{N}}\end{gathered} \tag{3.4}$$

REMOVE verb As 3.3, but matches multiple readings in a cohort. All targets are negated.

$$\neg \textit{una}_{\textsc{PrsP3}} \wedge \neg \textit{una}_{\textsc{PrsP1}} \wedge \neg \textit{una}_{\textsc{ImpP3}} \tag{3.5}$$

SELECT verb As 3.4, but matches multiple readings in a cohort. All other readings in the same cohort are negated. As previously, we need no explicit clause for $\textit{una}_{\textsc{PrsP3}} \vee \textit{una}_{\textsc{PrsP1}} \vee \textit{una}_{\textsc{ImpP3}}$: the default rule 3.2 guarantees that at least one of the target readings is true.

$$\neg \textit{una}_{\textsc{Det}} \wedge \neg \textit{una}_{\textsc{Prn}} \tag{3.6}$$

Positive conditions

Rules with contextual tests apply to the target, if the conditions hold. This is naturally represented as implications. We demonstrate both REMOVE and SELECT rules with a single condition in 3.7; the rest of the variants only with REMOVE. All rule types can be changed to SELECT by changing the consequent from $\neg \textit{más}_{\textsc{Adj}}$ to $\neg \textit{más}_{\textsc{Adv}}$.

$$\begin{array}{ll}\textbf{REMOVE adj IF (1 adj)} & \textbf{SELECT adj IF (1 adj)} \\ \textit{científica}_{\textsc{Adj}} \implies \neg \textit{más}_{\textsc{Adj}} & \textit{científica}_{\textsc{Adj}} \implies \neg \textit{más}_{\textsc{Adv}}\end{array} \tag{3.7}$$

REMOVE adj IF (-1 n) (1 adj) Conjunction of conditions.

REMOVE adj IF (-1 n LINK 2 adj) Linked conditions—identical to the above.

$$\textit{científica}_{\textsc{Adj}} \wedge \textit{aproximación}_{\textsc{N}} \implies \neg \textit{más}_{\textsc{Adj}} \tag{3.8}$$

REMOVE adj IF ((-1 n) OR (1 adj)) Disjunction of conditions (template).

$$\textit{científica}_{\text{Adj}} \vee \textit{aproximación}_{\text{N}} \implies \neg \textit{más}_{\text{Adj}} \tag{3.9}$$

REMOVE adj IF (1C adj) Careful context. Condition must be must be unambiguously adjective.

$$\textit{científica}_{\text{Adj}} \wedge \neg \textit{científica}_{\text{N}} \implies \neg \textit{más}_{\text{Adj}} \tag{3.10}$$

REMOVE adj IF (-1* n) Scanning. Any noun before the target is a valid condition.

$$\begin{aligned} \textit{sobre}_{\text{N}} \vee \textit{aproximación}_{\text{N}} &\implies \neg \textit{más}_{\text{Adj}} \\ \textit{sobre}_{\text{N}} \vee \textit{aproximación}_{\text{N}} &\implies \neg \textit{científica}_{\text{Adj}} \end{aligned} \tag{3.11}$$

REMOVE adj IF (-1* n LINK 1 v) Scanning and linked condition. Any noun before the target, followed immediately by a verb, is a valid condition.

$$\begin{aligned} \textit{sobre}_{\text{N}} \wedge (\textit{una}_{\text{PrsP3}} \vee \textit{una}_{\text{PrsP1}} \vee \textit{una}_{\text{ImpP3}}) &\implies \neg \textit{más}_{\text{Adj}} \\ \textit{sobre}_{\text{N}} \wedge (\textit{una}_{\text{PrsP3}} \vee \textit{una}_{\text{PrsP1}} \vee \textit{una}_{\text{ImpP3}}) &\implies \neg \textit{científica}_{\text{Adj}} \end{aligned} \tag{3.12}$$

REMOVE adj IF (-1* n LINK 1* adv) Scanning and linked condition. Any noun before the target, followed anywhere by an adverb, is a valid condition.

$$\begin{aligned} (\textit{sobre}_{\text{N}} \wedge \textit{más}_{\text{Adv}}) \vee (\textit{aproximación}_{\text{N}} \wedge \textit{más}_{\text{Adv}}) &\implies \neg \textit{más}_{\text{Adj}} \\ (\textit{sobre}_{\text{N}} \wedge \textit{más}_{\text{Adv}}) \vee (\textit{aproximación}_{\text{N}} \wedge \textit{más}_{\text{Adv}}) &\implies \neg \textit{científica}_{\text{Adj}} \end{aligned} \tag{3.13}$$

REMOVE adj IF (-1* n BARRIER det) Scanning up to a barrier. Any noun before the target, up to a determiner, is a valid condition: $\textit{sobre}_{\text{N}}$ is a valid condition only if $\textit{una}_{\text{Det}}$ is false.

$$\begin{aligned} (\textit{sobre}_{\text{N}} \wedge \neg \textit{una}_{\text{Det}}) \vee \textit{aproximación}_{\text{N}} &\implies \neg \textit{más}_{\text{Adj}} \\ (\textit{sobre}_{\text{N}} \wedge \neg \textit{una}_{\text{Det}}) \vee \textit{aproximación}_{\text{N}} &\implies \neg \textit{científica}_{\text{Adj}} \end{aligned} \tag{3.14}$$

REMOVE adj IF (-1* n CBARRIER det) Scanning up to a careful barrier. Any noun before the target, up to an unambiguous determiner, is a valid condition. The variable una_{Det} fails to work as a barrier, if any of the other analyses of *una* is true. Let una_{Any} denote the disjunction $una_{\text{Prn}} \vee una_{\text{PrsP3}} \vee una_{\text{PrsP1}} \vee una_{\text{ImpP3}}$.

$$\begin{aligned}(sobre_{\text{N}} \wedge una_{\text{Any}}) \vee aproximación_{\text{N}} &\implies \neg más_{\text{Adj}}\\ (sobre_{\text{N}} \wedge una_{\text{Any}}) \vee aproximación_{\text{N}} &\implies \neg científica_{\text{Adj}}\end{aligned} \tag{3.15}$$

REMOVE adj IF (-1C* n) Scanning with careful context. Any unambiguous noun before the target is a valid condition: $sobre_{\text{N}}$ is a valid condition only if it is the only reading in its cohort, i.e. $sobre_{\text{Pr}}$ is false.

$$\begin{aligned}(sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}}) \vee aproximación_{\text{N}} &\implies \neg más_{\text{Adj}}\\ (sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}}) \vee aproximación_{\text{N}} &\implies \neg científica_{\text{Adj}}\end{aligned} \tag{3.16}$$

REMOVE adj IF (-1C* n BARRIER det) Scanning with careful context, up to a barrier. Any unambiguous noun before the target, up to a determiner, is a valid condition: $sobre_{\text{N}}$ is a valid condition only if it is the only reading in its cohort ($sobre_{\text{Pr}}$ is false), and there is no determiner between *sobre* and the target (una_{Det} is false).

$$\begin{aligned}(sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}} \wedge \neg una_{\text{Det}}) \vee aproximación_{\text{N}} &\implies \neg más_{\text{Adj}}\\ (sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}} \wedge \neg una_{\text{Det}}) \vee aproximación_{\text{N}} &\implies \neg científica_{\text{Adj}}\end{aligned} \tag{3.17}$$

REMOVE adj IF (-1C* n CBARRIER det) Scanning with careful context, up to a careful barrier. Like above, but *una* fails to work as a barrier if it is not unambiguously determiner.

$$\begin{aligned}(sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}} \wedge una_{\text{Any}}) \vee aproximación_{\text{N}} &\implies \neg más_{\text{Adj}}\\ (sobre_{\text{N}} \wedge \neg sobre_{\text{Pr}} \wedge una_{\text{Any}}) \vee aproximación_{\text{N}} &\implies \neg científica_{\text{Adj}}\end{aligned} \tag{3.18}$$

Inverted conditions

In the following, we demonstrate the effect of two inversion operators. The keyword NOT inverts a single contextual test, such as **IF (NOT 1 noun)** , as well as linked conditions, such as **IF (-2 det LINK NOT *1 noun)** . The keyword NEGATE inverts a whole conjunction of contextual tests, which may have any polarity: **IF (NEGATE -2 det LINK NOT 1 noun)** means “there may not be a determiner followed by a not-noun”; thus, *det noun* would be allowed, as well as *prn adj*, but not *det adj*. Inversion cannot be applied to a BARRIER condition. If one wants to express **IF (*1 foo BARRIER ¬bar)** , that is, "try to find a *foo* until you see the first item that is not *bar*", a set complement operator must be used: **(*) - bar** .

There is a crucial difference between matching positive and inverted conditions. If a positive condition is out of scope or the tag is not present in the initial analysis, the rule simply does not match, and no clauses are created. For instance, the conditions '10 adj' or '-1 punct', matched against our example passage, would not result in any action. In contrast, when an inverted condition is out of scope or inapplicable, that makes the action happen unconditionally. As per VISL CG-3, the condition **NOT 10 adj** applies to all sentences where there is no 10th word from target that is adjective; including the case where there is no 10th word at all. If we need to actually have a 10th word to the right, but that word may not be an adjective, we can, again, use the set complement: **IF (10 (*) - adj)** .

REMOVE adj IF (NOT 1 adj) Single inverted condition. There is no word following *científica*, hence its adjective reading is removed unconditionally.

$$\begin{aligned} \neg \textit{científica}_{\text{ADJ}} &\implies \neg \textit{más}_{\text{ADJ}} \\ &\implies \neg \textit{científica}_{\text{ADJ}} \end{aligned} \tag{3.19}$$

REMOVE adj IF (NOT -1 n) (NOT 1 adj) Conjunction of inverted conditions.

$$\begin{aligned} \neg\ \textit{aproximación}_{\text{N}} \wedge \neg \textit{científica}_{\text{ADJ}} &\implies \neg \textit{más}_{\text{ADJ}} \\ &\implies \neg \textit{científica}_{\text{ADJ}} \end{aligned} \tag{3.20}$$

REMOVE adj IF (NEGATE -3 pr LINK 1 det LINK 1 n) Inversion of a conjunction of conditions.

$$\begin{aligned} \neg(\textit{sobre}_{\text{PR}} \wedge \textit{una}_{\text{DET}} \wedge\ \textit{aproximación}_{\text{N}}) &\implies \neg \textit{más}_{\text{ADJ}} \\ &\implies \neg \textit{científica}_{\text{ADJ}} \end{aligned} \tag{3.21}$$

REMOVE adj IF (NOT 1C adj) Negated careful context. Condition cannot be unambiguously adjective.

$$\begin{aligned} \neg(\textit{científica}_{\text{Adj}} \wedge \neg\textit{científica}_{\text{N}}) &\implies \neg\textit{más}_{\text{Adj}} \\ &\implies \neg\textit{científica}_{\text{Adj}} \end{aligned} \tag{3.22}$$

REMOVE adj IF (NOT -1* n) Scanning. There must be no nouns before the target.

$$\begin{aligned} \neg(\textit{sobre}_{\text{N}} \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{más}_{\text{Adj}} \\ \neg(\textit{sobre}_{\text{N}} \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{científica}_{\text{Adj}} \end{aligned} \tag{3.23}$$

REMOVE adj IF (NOT -1* n BARRIER det) Scanning up to a barrier. There must be no nouns before the target, up to a determiner.

$$\begin{aligned} \neg((\textit{sobre}_{\text{N}} \wedge \neg\textit{una}_{\text{Det}}) \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{más}_{\text{Adj}} \\ \neg((\textit{sobre}_{\text{N}} \wedge \neg\textit{una}_{\text{Det}}) \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{científica}_{\text{Adj}} \end{aligned} \tag{3.24}$$

REMOVE adj IF (NOT -1* n CBARRIER det) Scanning up to a careful barrier. There must be no nouns before the target, up to an unambiguous determiner.

$$\begin{aligned} \neg((\textit{sobre}_{\text{N}} \wedge \neg\textit{sobre}_{\text{Pr}} \wedge \textit{una}_{\text{Any}}) \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{más}_{\text{Adj}} \\ \neg((\textit{sobre}_{\text{N}} \wedge \neg\textit{sobre}_{\text{Pr}} \wedge \textit{una}_{\text{Any}}) \vee \textit{aproximación}_{\text{N}}) &\implies \neg\textit{científica}_{\text{Adj}} \end{aligned} \tag{3.25}$$

Non-matching inverted conditions Here we demonstrate a number of inverted rules, in which the contextual test does not match the example sentence. As a result, the action is performed unconditionally.

REMOVE adj IF (NOT 1 punct) Single inverted condition, not present in initial analysis.

REMOVE adj IF (NOT 10 adj) Single inverted condition, out of scope.

REMOVE adj IF (NOT 1 n) (NOT 10 n) Conjunction of inverted conditions, one out of scope.

REMOVE adj IF (NEGATE -3 pr LINK 1 punct) Inversion of a conjunction of conditions, some not present in initial analysis.

$$\implies \neg\textit{más}_{\text{Adj}} \tag{3.26}$$

3.4 Experiments

In this section, we report experiments on the two modifications to the parallel scheme, presented in the previous section. We evaluate the performance against VISL CG-3 in accuracy and running time; in addition, we offer some preliminary observations on the effect of grammar writing.

For these experiments, we implemented another variant for both schemes: we force all the literals that are not targeted by the original rules to be true. This modification destroys the "rules may disambiguate their conditions" property, which we hypothesised to be helpful; however, turns out that this variation slightly outperforms even VISL CG-3 in terms of precision and recall. Conversely, the original SAT-CG scheme, where all variables are left unassigned, fares slightly worse for accuracy. Overall, the accuracy is very low with the tested grammars, thus it is hard to draw conclusions. As for execution time, all variants of SAT-CG perform significantly worse than VISL CG-3—depending on the sentence length and the number of rules, SAT-CG ranges from 10 times slower to 100 times slower. The results are presented in more detail in the following sections.

3.4.1 Performance against VISL CG-3

	19 rules			**261 rules**		
	F-score		Time	F-score		Time
SAT-CG$_{\text{Max}}$	80.22^{U} %	83.28^{F} %	4s	78.15^{U} %	79.03^{F} %	8s
SAT-CG$_{\text{Ord}}$	81.14^{U} %	83.12^{F} %	4s	79.03^{U} %	79.17^{F} %	8s
VISL CG-3	81.52 %		0.39s	78.83 %		0.64s

Table 3.1: F-scores and execution times for the subset of the Spanish grammar, tested on a gold standard corpus of 20,000 words.

We took a manually tagged corpus[4] containing approximately 22,000 words of Spanish news text, and a small constraint grammar[5] from the Apertium repository. We kept only SELECT and REMOVE rules, which left us 261 rules. In order to test the grammars, we produced an ambiguous version of the tagged corpus: we discarded the tags, and analysed the text again with the Apertium morphological analyser and the morphological dictionary for Spanish. Ideally, we should have used the same analyser, but since the gold standard was produced, there have been some changes in the dictionary. Due to these differences, we had

[4]https://svn.code.sf.net/p/apertium/svn/branches/apertium-swpost/apertium-en-es/es-tagger-data/es.tagged
[5]https://github.com/apertium/apertium-spa/blob/master/apertium-spa.spa.rlx, commit 1d4f290d

to discard some 2000 words from the gold standard—in most cases, there was a change of just one word, but we excluded the whole sentence nevertheless. With this setup, we ran both SAT-CG and VISL CG-3 on the newly ambiguous corpus. In addition, we wrote a small grammar of 19 rules, optimised to be very compact and disambiguate effectively, taking advantage of the parallel execution. Some of the rules were selected from the original grammar, and some were written by the author.

Table 3.1 shows F-scores and execution times for these two grammars. SAT-CG$_{\text{Max}}$ is the parallel scheme with maximisation, and SAT-CG$_{\text{Ord}}$ is the parallel scheme where rule applications are introduced one by one; superscript U refers to the unassigned variant and F to the version where we force non-targeted literals true. The original grammar performs very poorly, with an F-score between 78–79 % for all three setups—even with an empty grammar, the F-score would be around 70 %. Due to the unrepresentative example, we cannot draw many conclusions. We had access to other grammars of the same size range, such as Portuguese, Dutch and Russian; however, no sufficiently large gold standard corpora were available.

The experiment with the 19-rule grammar was more interesting. We intended to write a grammar that would perform exceptionally well for the parallel execution; it was heavily tuned to the gold standard corpus, and tested only with SAT-CG (leaving all variables unassigned) during development. To our surprise, this small grammar turned out to outperform the original, 261-rule grammar, even when run with VISL CG-3. The rule ordering was not optimised by automatic means, and sections were not used; hence it is possible that with another order, the numbers could be even better for VISL CG-3 and the ordered scheme.

3.4.2 Execution time

	Spanish (380k words)		Finnish (19k words)
	19 rules	**261 rules**	**1185 rules**
SAT-CG$_{\text{Max}}$	25s	1m 5s	4m 56s
SAT-CG$_{\text{Ord}}$	19s	1m 2s	4m 17s
VISL CG-3	2.7s	4.8s	4.3s

Table 3.2: Execution times for Spanish and Finnish grammars of different sizes, disambiguating Don Quijote (384,155 words) and FinnTreeBank (19,097 words).

Number of rules vs. number of words In addition to the 20,000-word corpus of news text, we tested the performance by parsing Don Quijote (384,155 words) with the same Spanish

	261 rules		**1185 rules**	
	283 tokens	Split	251 tokens	Split
SAT-CG$_{\text{Max}}$	2.26s	2.06s	5.34s	2.36s
SAT-CG$_{\text{Ord}}$	2.20s	1.99s	4.70s	2.36s
VISL CG-3	0.05s	0.03s	0.04s	0.04s

Table 3.3: Experiments with sentence length. On the left, Spanish 261-rule grammar parsing the complete 283-token sentence, parsed as one unit vs. split at semicolons into four parts. On the right, Finnish 1185-rule grammar parsing an artificially constructed 251-token sentence.

grammars as in the previous experiment. More importantly, we wanted to test a grammar of a realistic size, so we took a Finnish grammar, originally written in 1995 for CG-1, and updated into the VISL CG-3 standard by Pirinen (2015). We discarded rules and constructions that SAT-CG does not support, and ended up with 1185 rules. Then, we parsed around 19,000 words of text from FinnTreeBank (Voutilainen, 2011). Table 3.2 shows the results for both Finnish and Spanish tests.

For both systems, the number of rules in the grammar affects the performance more than the raw word count. However, the performance of SAT-CG gets worse faster than VISL CG-3s. From the SAT-solving side, maximisation is the most costly operation. Emulating order performs slightly faster, but still in the same range: maximisation is not needed, but the solve function is performed after each clause—this gets costlier after each rule. However, we believe that SAT is not necessary the bottleneck: VISL CG-3 is a product of years of engineering effort, whereas SAT-CG is still, despite the improvements from Listenmaa and Claessen (2015), a rather naive implementation, written in Haskell and BNFC.

Sentence length In addition, we explored the effect of sentence length further. When split by just full stops, the longest sentence in Don Quijote consists of 283 tokens, including punctuation. In Table 3.3, we see this sentence parsed, on the right as a single unit, and on the left, we split it in four parts, at the semicolons. In Table 3.2, it was already split at semicolons. We tested the effect of sentence length also for the Finnish grammar. All the sentences in FinnTreeBank were very short, so we created an 251-token "sentence" by pasting together 22 short sentences and replacing sentence boundaries with commas.

The results in Table 3.3 are promising: the execution time does not grow unbearably, even with the unusually long sentence. To give some context, let us consider the performances of the sequential CG-2 and the parallel FSIG in 1998. Voutilainen (1998) reports the performance of a 3,500-rule CG: "On a 266 MHz Pentium running Linux, EngCG-2 tags around 4,000 words per second". In contrast, a 2,600-rule FSIG grammar is unable to find a correct parse in the allowed time, 100 seconds per sentence, for most sentences longer than 15 words. As another CG result from the late 1990s, Tapanainen (1999) reports an average time of 5 seconds for parsing 15-word sentences in CG-2.

Compared to the parsing times of sequential CG and FSIG nearly 20 years ago, our results demonstrate a smaller difference, and much less steep growth in the execution time related to sentence length. Of course, this remark should be taken with caution: to start with, the described experiments were conducted on different systems and in different centuries[6]. Importantly, the parallel participants in the experiments were using different grammars—the FSIG grammar used in Voutilainen (1998) contains much more difficult operations, which makes the constraint problem larger. Our experiments have just shown the lack of blowup for a small grammar. Nevertheless, it would be interesting to conduct this experiment properly: try a SAT-based approach for executing the FSIG grammars from the 1990s, and compare the results to the current state of the art.

3.4.3 Effect on grammar writing

Throughout the implementation of SAT-CG, we have predicted that the more declarative features would influence the way rules are written. We hoped to combine the best parts of SCG and PCG: the rules would be more expressive and declarative, but we could still fall back to heuristics: eventual conflicts would not render the whole grammar useless. On the one hand, getting rid of ordering and cautious context could ease the task of the grammar writer, since it removes the burden of estimating the best sequence of rules and whether to make them cautious. On the other hand, lack of order can make the rules less transparent, and might not scale up for larger grammars. Without an actual CG writer, it is hard to say whether this prediction holds or not.

As stated earlier, we got the best F-score with a modification which prevents the rules from disambiguating their context: in the case with *la casa grande* and the three rules which all target *casa*, we would assign both la_{DET} and la_{PRN} true. We tried also a second variant, where we would assign true to only those variables which are neither targets nor conditions to any rule. Table 3.4 shows in more detail what happens to the precision and recall with all the three variants, with the 19-rule grammar.

[6] Eckhard Bick (personal communication) points out that CG-2 run on a modern system is around 6 times faster than VISL CG-3.

We notice that the SAT-based solution loses severely in recall, when all variables are left unassigned. This is understandable: given that the grammar is extremely small, only 19 rules, the majority of the tokens will never appear in the clauses, neither as targets nor conditions. So the SAT-solver is free to arbitrarily choose the readings of those free words, whereas VISL CG-3 does nothing to them. There is no universal behaviour what to do if a variable is not included in any of the clauses; it is likely that many solvers would just assign false by default. In our case, we still require at least one reading of every cohort to be true; this applies even for cohorts which do not appear in the clauses. Given the lack of other information, the SAT-solver is likely to just choose the first one in alphabetical order. It would not be difficult to modify the scheme based on another heuristic, such as the most likely unigram reading, or dispreference for analyses with many compound splits. However, we did not add any heuristics; the results just show the SAT-solver performing its default actions.

	NoAss		NoAff		NoTar	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
SAT-CG$_{\text{Max}}$	79.97	80.47	77.24	82.94	**80.36**	**86.41**
SAT-CG$_{\text{Ord}}$	**80.96**	81.31	**78.42**	84.07	80.22	86.28
VISL CG-3	78.29	**85.02**	78.29	**85.02**	78.29	85.02

Table 3.4: Showing the effect of three variants on the 19-rule grammar: (NoAss) Leave all variables unassigned; (NoAff) Assign true to variables which do not appear in any rule; (NoTar) Assign true to variables which are not targeted by any rule.

The precision in SAT-CG is better already in the unassigned scheme. The percentage is calculated by overall number of analyses, but even at the level of different words, the advantage is at SAT-CG. Out of the gold standard corpus, VISL CG-3 differs in the analyses of some 4,200 words, whereas SAT-CG, in the unassigned variant, disagrees only on 3700-3900 words. Of course, the differences are still very small, and this effect could also be due to randomness: sometimes the SAT-solver happens to disambiguate correctly just by removing a random analysis.

We move on to the second variant, where the completely unaffected variables are assigned true. Unsurprisingly, the recall grows: we have just added some thousands of true variables, some of which would have been otherwise randomly removed by the SAT-solver. The ordered variant resembles most the behaviour of VISL CG-3, both in precision, recall and the amount of words where the analysis disagrees; now the number is around 4,200 for both.

The third variant, where all non-targeted analyses are assigned true, performs the best. High recall is expected; now we force some more thousands of variables true, among them must be some actually true readings. But at the same time, this limits the power of rules: they may only disambiguate targets, not conditions—why does precision go up? The most plausible explanation for this phenomenon, aside from mere chance, is the particular features of the tested grammar. When tested with the original 261-rule grammar, shown in Table 3.5, the pattern does not repeat: assigning more variables true just brings the performance closer to VISL CG-3.

	NOASS		NOAFF		NOTAR	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
SAT-CG$_{\text{Max}}$	77.03	79.32	70.85	86.46	**71.91**	87.72
SAT-CG$_{\text{Ord}}$	**77.91**	80.18	70.79	86.40	71.78	87.62
VISL CG-3	71.17	**88.34**	**71.17**	**88.34**	71.17	**88.34**

Table 3.5: Evaluation of the three variants, for the original 261-rule grammar.

```
# la casa (en que vivo)
SELECT:i01_s_det Det IF (NOT 1C VerbFin) ;

# (el sacerdote) la casa (con el novio)
SELECT:i08_s_pro PrnIndep IF (1C VerbFin) ;
```

Figure 3.6: Excerpt from the 19-rule Spanish grammar.

Now, what are these features of the tiny grammar? One reason may be that many of the rules come in pairs. Figure 3.6 shows an excerpt from the grammar: due to these rules, both *la*$_{\text{DET}}$ and *la*$_{\text{PRN}}$ would be targeted, and remain unassigned. Another peculiarity in the 19-rule grammar is the high number of SELECT rules—this means that all the other readings in the cohort become the actual targets. Thus, if a rule addresses some phenomenon, most of the readings in the target cohort are saved from the forced assignment.

To conclude, this experiment has been very inconclusive, due to the extremely low coverage of the tested grammars. A high-quality grammar would target significantly more individual words in the test corpus; this means less risk for the SAT-solver to randomly decide the values for untargeted readings. We would hope that the power of rules disambiguating their condition would help such a grammar to go that extra mile; however, based on the experiments so far, this is mostly speculation.

It would be interesting to repeat the experiment in writing a serious, large-scale grammar. Some questions to address would be (a) do we need fewer rules; (b) do we need careful context; (c) do we write more select rules; and (d) will a grammar that performs well with parallel CG also perform well with sequential CG?

3.5 Summary

In this section, we started from the known parallels between CG and logic: the rules express what is true (select) and false (remove), under certain conditions (if). Expressing this as a SAT-problem, we get a working CG implementation, which executes the rules in a parallel and unordered manner. However, a straight-forward parallel encoding means that the rules of a given grammar should contain no contradictions. To alleviate the brittleness of PCG, we developed two approaches to discard conflicting rule applications. The first one is based on order; introducing clauses one by one, and assuming that the previous clauses are true, and the second one is based on maximisation; trying to make as many clauses as possible apply.

As a conclusion, SAT-solver provides an interesting alternative for a CG engine, but loses in practicality: performance and lack of transparency for grammar writers. We see more potential in contributing to the FSIG community, with a practical implementation and novel ways to get around conflicts. It would be interesting to test an actual syntactic FSIG grammar encoded in SAT, to see if a SAT-based solution is powerful enough for the more complex constraints.

Chapter 4

Analysing Constraint Grammar

Another desirable facility in the grammar development environment would be a mechanism for identifying pairs of rules that contradict each other.
Atro Voutilainen, 2004

In the previous chapter, we presented a tool. In the current chapter, we will solve a problem.

Recall the design principles of CG from Section 2.1.2: by design, the grammars are shallow and low-level. There is no particular hierarchy between lexical, morphological, syntactic or even semantic tags: individual rules can be written to address any property, such as "verb", "auxiliary verb in first person singular", or "the word form *sailor*, preceded by *drunken* anywhere in the sentence". This makes it possible to treat very particular edge cases without touching the more general rule: we would simply write the narrow rule first ("if noun AND *sailor*"), and introduce the general rule ("if noun") later.

However, this design is not without problems. As CGs grow larger, it gets harder to keep track of all the rules and their interaction. Despite this well-known issue, there has not been a tool that would help grammar writers to detect conflicting rules. Following the idea further, the tool could give feedback that is not restricted to conflicts, but also other features that are helpful in the process of writing grammar. Given the rules in Figure 4.1, a grammar writer may ask the following questions.

- Are all the Portuguese rules distinct? (e.g. `Para` and `De` may be included in `Prep`)
- Could two or more rules be merged? (e.g. `SELECT Inf IF -1 Prep OR Vai OR Vbmod ...`)

```
SELECT Inf IF ...
  (-1 Prep) (0C V) ;
  (-1 Para OR De) (0C V) ;
  (-1C Vbmod) (0C V) ;
  (-1C Vai) ;
  (-1C Vbmod) (0 Ser) ;
  (-1C Ter/de) ;
```

```
                 SELECT V + Prs/Imprt + Act + Neg IF ...
(*-1C Negv LINK NOT *1 Vfin)     (NOT *-1 Niin OR Neg)
(NOT 0 N) (NOT 0 Pron)           (*-1C Negv
(NOT *1 Neg) (NOT *-1 Neg)         LINK NOT 0 Imprt
(NOT 0 Pass) (NOT *-1 Niin)        LINK NOT *1 Vfin OR CLB?)
(*-1C Negv LINK NOT *1 CLB?)     (NOT 0 N OR Pron OR Pass)
(*-1C Negv LINK NOT 0 Imprt) ;   (NOT *1 Neg) ;
```

Figure 4.1: Left: rules to select infinitive in Portuguese. Right: two versions of a condition in Finnish.

- What is the best order for the rules?
- Can the Finnish rule on the left be rewritten in the form shown on the right?
- Generate an example sequence that triggers rule(s) R but not rule(s) R'.

The chapter follows with introduction of related work: namely, corpus-based methods to aid grammar writing, and automatic optimisation of a complete, human-written grammar. We continue by presenting our solution, along with a working implementation, and finally, evaluate its performance.

Most of the content in this chapter is based on "Analysing Constraint Grammars with a SAT-solver" (Listenmaa and Claessen, 2016). In addition, we include two later developments. Section 4.4, based on the article "Cleaning up the Basque grammar: a work in progress" (Listenmaa et al., 2017), describes a practical application of our tool and collaboration with CG writers. Section 4.5 presents the main ideas from "Exploring the Expressivity of Constraint Grammar" (Kokke and Listenmaa, 2017)—a side effect following the creation of *symbolic sentences*, it allowed us to pose the question "what if CG was a generative formalism?" Section 4.5 is only loosely related to the main line of work, and can be safely skipped by the uninterested reader.

4.1 Related work

There has been previous research on corpus-based methods in manual grammar development (Voutilainen, 2004), as well as optimisation of hand-written CGs (Bick, 2013). In addition, there is a large body of research on automatically inducing rules, e.g. Samuelsson et al. (1996); Eineborg and Lindberg (1998); Lager (2001); Sfrent (2014). However, since our work is aimed to aid the process of hand-crafting rules, we omit those works from our discussion.

Due to adding Section 4.5, we also include work addressing the expressivity and complexity of CG.

Corpus-based methods in manual grammar development Hand-annotated corpora are commonly used in the development of CGs, because they give immediate feedback whether a new rule increases or decreases accuracy. Voutilainen (2004) gives a detailed account about best practices of grammar writing and efficient use of corpora to aid the grammar development. For a language with no free or tagset-compatible corpus available, Reynolds and Tyers (2015) describe a method where they apply their rules to unannotated Wikipedia texts and pick 100 examples at random for manual check.

CG rules are usually arranged in sections, and run in the following manner. First apply rules from section 1, and repeat until nothing changes in the text. Then apply rules from sections 1–2, then 1–3 and so on, until the set includes all rules. The best strategy is to place the safest and most effective rules in the first sections, and leave the more heuristic (i.e. less safe) rules as a last resort; only to act on the parts that haven't been disambiguated yet by the safer rules. A representative corpus is arguably the best way to get concrete numbers—how many times a rule applied and how often it was correct—and to arrange the rules in sections based on that feedback.

Voutilainen (2004) states that the around 200 rules are probably enough to resolve 50–75 % of ambiguities in the corpus used in the development. This figure is very much thanks to Zipf's law: we can add rules that target the most frequent *tokens*, thus disambiguating a high number of word forms. However, this method will not notice a missed opportunity or a grammar-internal conflict, nor suggest ways to improve; neither does it guarantee a coherent whole of rules. While the coverage information is easy to obtain from a corpus, there is no tool that would aid grammar writers in including wide coverage of different linguistic phenomena.

Automatic optimisation of hand-written grammars The corpus-based method can tell the effect of each single rule at their place in the rule sequence, and leaves the grammar writer to make changes in the grammar. As a step further, Bick (2013) modifies the grammar automatically, by trying out different rule orders and altering the contexts of the rules. Bick reports error reduction of 7–15% compared to the original grammars. This is a valuable tool, especially for grammars that are so big that it's hard to keep track manually. A program can try all combinations whereas trying to make sense out of a huge set of rules would be hard for humans. As a downside, the grammar writer will likely not know why exactly does the tuned grammar perform better.

Expressivity and complexity of CG Tapanainen (1999) gives an account of the expressivity of the contextual tests for 4 different constraint formalisms, including CG. In addition, parsing complexity can be easily defined for a given variant and implementation of CG; see for instance Nemeskey et al. (2014). Yli-Jyrä (2017) relates CG to early formal language theory, and provides a proof of non-monotonic[1] CG being Turing-complete.

4.2 Analysing CGs

We start by defining a conflict, and present requirements for a solution. Then, we introduce a logical translation of sequential CG, corresponding to Lager and Nivre (2001), and modify it into a SAT-problem about the *original* sentence before applying the rules. We refine our solution by restricting what kind of sentences we can create. The whole method requires only a morphological lexicon, no corpus.

Conflict We define *conflict* as follows: a list of rules R is in conflict with the rule r, if applying R makes it impossible to apply r, regardless of input. Some examples of conflicts follow:

- If two equivalent rules r and r' occur in the grammar, the second occurrence will be disabled by the first
- A list of rules R selects something in a context, and r' removes it
- A list of rules R removes something in a context, and r' selects it
- A list of rules R removes something from the context of a rule r', so r' can never apply
- A rule r has an internal conflict, such as non-existent tag combination, or contradictory requirements for a context word

This definition is very similar to the concept *bleeding order* in generative phonology (Kiparsky, 1968); however, since we are talking about an explicit list of human-written rules, we include also rule-internal conflicts in our classification. The conflicting (or "bleeding") R can be a single rule or a list of rules: for instance, if one rule removes a verb in context C, and another

[1] A monotonic variant of CG may only remove readings from cohorts, whereas a non-monotonic variant may add readings or cohorts.

in context $\neg C$, together these rules remove a verb in all possible cases, disabling any future rule that targets verbs.

Solution How do we find out if a rule r can act? We could apply the grammar to a large corpus and count the number of times each rule fires; if some rule never fires, we can suspect there is something wrong in the rule itself, or in the interaction with previous rules. But it can also be that r just targets a rare phenomenon, and there was no sentence in the corpus that would trigger it.

Of course, tagged corpora are extremely useful for many questions in CG analysis. A corpus can show which rules are the most used, or which rules give false analyses. Luckily, we have already methods for finding such statistics—the question about conflicts is orthogonal to those, and the solution needs to address only the part about conflicts, not about rules being frequent or accurate. Therefore, we can take a more artificial approach. Remember the definition of conflict: "applying R makes it impossible to apply r *regardless of input*". In a nutshell, we are going to show the absence of a conflict by trying to create a sequence where r can apply after R; conversely, we detect a conflict by showing that such sequence cannot be created.

4.2.1 From disambiguation to generation

SAT-encoding of sequential CG In order to analyse real-life CGs, we need to model the semantics of sequential CG: rule application takes effect immediately, is irreversible[2], and only allows changes in the target. We restrict ourselves to the operations on morphological disambiguation, and only SELECT and REMOVE rules. The following encoding does not support dependency relations, nor rule types which add readings or cohorts. Last time, we only used ordering as a way to choose the which rules to apply. But there was no state: all clauses operated in parallel, on the same variables. Now, we need a new variable for a reading every time when it is targeted; that is, when its state potentially changes. As before, a sentence is a vector of cohorts, which is, in turn, a vector of variables representing the readings of the current cohort. At each rule application, we create a new variable $\mathit{word}'_{\text{RD}}$ for each targeted reading $\mathit{word}_{\text{RD}}$. The new variable $\mathit{word}'_{\text{RD}}$ is true iff
We keep the same example sequence from Chapter 3, *la casa grande*. In the following, we apply the rule `REMOVE v IF (-1 det)` to it.

[2]As pointed out by Eckhard Bick, VISL CG-3 allows reference to deleted readings. In principle, this would not be an obstacle for our implementation, because no variables are removed.

(a) *word*$_{\text{RD}}$ was true, and (b) the rule cannot apply: this can be because
– its conditions do not hold, or
– it would remove the last reading.

$$\text{New variable } casa'_{\text{V}} \iff casa_{\text{V}} \land (\overbrace{\neg la_{\text{Det}}}^{\text{invalid condition}} \lor \overbrace{(casa_{\text{V}} \land \neg casa_{\text{N}})}^{\text{only target left}})$$

After the application, we replace each *word*$_{\text{RD}}$with *word*$'_{\text{RD}}$in the sentence vector; those variables that are not touched by the rule, will be carried over to the next round unchanged. For example, if we apply two rules which target verbs and one which targets pronouns, the sentence vector will look like the following.

$$\begin{aligned} la &\rightarrow \{la_{\text{Det}}, la'_{\text{Prn}}\} \\ casa &\rightarrow \{casa''_{\text{V}}, casa_{\text{N}}\} \\ grande &\rightarrow \{grande_{\text{Adj}}\} \end{aligned}$$

Symbolic sentence We saw how $casa'_{\text{V}}$ was created, based on the status of $casa_{\text{V}}$ and the conditions at the time. Similarly, $casa''_{\text{V}}$ will be created based on $casa'_{\text{V}}$. But what is the status of the variables in the beginning? In fact, that decision makes a crucial difference. If all variables start off as true, then we have, in effect, reimplemented the logical encoding by Lager and Nivre (2001). This option would not make an interesting SAT-problem: there is no search involved, just manipulation of Boolean expressions in a completely deterministic way. All the new variables, created from the rule applications, would get their value immediately: $casa'_{\text{V}} \Leftrightarrow grande_{\text{Adj}} \land \neg(casa_{\text{V}} \land \neg casa_{\text{N}})$ just translates into $casa'_{\text{V}} \Leftrightarrow True \land \neg(True \land \neg True)$.

In order to turn this problem into a SAT-problem, we will have the readings start off unassigned. All the other variables computed along the way depend on the original variables, so the question to the SAT-solver becomes: *"Which readings were originally true?"* For implementing a CG engine, this question does not make much sense: we know which readings were given by the morphological analyser. But our task now is to *generate* the original sentence, which will pass through the rules. Here comes the most important modification: we will apply the rules to something we call *symbolic sentence*. Every cohort, called *symbolic word*, contains every possible reading, and rule applications are responsible for shaping the sentence into a concrete one.

Before we can do any analysis any of the rules, we need to find out what the set of all possible readings of a word is. We can do this by extracting this information from a lexicon, but there are other ways too; we will explain this in more detail in Section 4.2.2. In our experiments, the number of readings has ranged from about 300 to about 9000.

Width of a rule Furthermore, when we analyse a rule r, we need to decide the *width* $w(r)$ of the rule r: How many different words should there be in a sentence that can trigger r? Most often, $w(r)$ can be easily determined by looking at how far away the rule context indexes in the sentence relative to the target. For example, in the rule `REMOVE v IF (-1 det)`, the width is 2.

If the context contains a `*` (context word can be anywhere), we need to make an approximation of $w(r)$, which may result in false positives or negatives later on in the analysis; this indeed happened in the Finnish grammar in Section 4.3.3. For example, given the rule `REMOVE v IF (-1* det)`, we tried first a sentence of width 2; if there was a conflict, we tried width 3, and then 4. For a rule with multiple `*`s, we create combinations in the range of $\pm$ 2 symbolic words for each `*`-condition; in addition, we need to specify where the target is in the symbolic sentence. For instance, the rule `REMOVE v IF (-1* det) (1* det)` would be tested with the following combinations, where C means condition and T means target: (C,T,C); (C,C,T,C); (C,T,C,C); (C,C,T,C,C); (C,C,C,T,C,C); (C,C,T,C,C,C) and (C,C,C,T,C,C,C).

Rule application Finally, we define what does it mean for a sentence to "pass through" a rule. Let us enumerate the cases:

1. The sentence is out of scope; target not removed
2. The conditions of the rule do not hold; target not removed
3. The target is the only possible reading left; target not removed
4. The target was never in place
5. The target is removed

Let us illustrate the difference between the last two cases with a rule that targets pronouns, say, `REMOVE prn IF (1 v)`. If the target was never in place, then both the original la_{PRN} and the newly created la'_{PRN} end up as false. If the target is removed, then la_{PRN} will be true and la'_{PRN} false. This means that a rule with the condition `IF (-1 prn)` behaves differently depending on where in the rule sequence it is placed: if it is applied before `REMOVE prn IF (1 v)`, then it will match the original reading la_{PRN}. Any rule applied after that will get the newly created variable la'_{PRN}.

Now we have the definitions in place. In the following, we are going to show two examples; first one a very simple conflict, and the second one with more complicated interaction.

Example 1: conflict Let us look at a conflicting case. For simplicity, we assume that the full set of readings in the language is {*det def, noun sg, noun pl, verb sg, verb pl*}. We ignore all lexical readings for now, and assume dummy words of the form w_i, where i refers to the index of the word. The rules are as follows:

```
r1 = REMOVE verb IF (-1 det) ;
r2 = REMOVE verb IF (-1 det) (1 noun) ;
```

We want to know if the last rule is able to apply, after going through the rules that come before it—in this case, there is only one such rule. The width of the last rule is three: one condition to the left of the target, one to the right, so we create a symbolic sentence of three words. Below, we show the symbolic sentence, consisting of the symbolic words *w1, w2* and *w3*.

```
"<w1>"                 "<w2>"                 "<w3>"
       "w1" det def           "w2" det def           "w3" det def
       "w1" noun sg           "w2" noun sg           "w3" noun sg
       "w1" noun pl           "w2" noun pl           "w3" noun pl
       "w1" verb sg           "w2" verb sg           "w3" verb sg
       "w1" verb pl           "w2" verb pl           "w3" verb pl
```

After applying the first rule, we have a situation which looks much like the one in the previous chapter, Figure 3.2. If the symbolic word *w1* is a determiner, then *w2* cannot be a verb; the only exception is the case where verb is the only analysis of *w2*. As before, the verb analysis *w2* can also be false, even if *w1* is not a determiner. The exact same dependencies are created between the symbolic words *w2* and *w3*. The first word *w1* is out of scope of the condition, because it has no preceding word. Below we see the affected readings after the first rule application, with newly created variables:

$$\begin{aligned}
1 &\rightarrow \{w1_{\text{DetDef}},\ w1_{\text{NSg}},\ w1_{\text{NPl}},\ w1_{\text{VSg}},\ w1_{\text{VPl}}\} \\
2 &\rightarrow \{w2_{\text{DetDef}},\ w2_{\text{NSg}},\ w2_{\text{NPl}},\ w2'_{\text{VSg}},\ w2'_{\text{VPl}}\} \\
3 &\rightarrow \{w3_{\text{DetDef}},\ w3_{\text{NSg}},\ w3_{\text{NPl}},\ w3'_{\text{VSg}},\ w3'_{\text{VPl}}\}
\end{aligned}$$

There is no “final sentence” yet, simply constraints on the possible combinations of determiners and verbs. If we asked for a solution now, it could be anything; for instance, all three words are determiners. Just asking for a freely chosen sequence is not very interesting; there are so many possibilities, and even after applying more rules, most of the combinations are not explicitly prohibited.

We want to know if there is a sequence that can trigger the last rule. In order to trigger the rule, the sequence must have the following three features:

- Target readings: at least one reading with a *verb* tag in the target position *w2*.
- Ambiguity: at least one reading without a *verb* tag in the target position; if there are only *verb* readings in the target, the rule would not fire.
- Conditions: at least one reading with a *det* tag in the cohort preceding the target, and at least one reading with a *noun* tag in the cohort following the target.

We can see that some of these requirements can be fulfilled: the first rule allows models where *w2* contains *target readings*. If the rule had been `REMOVE verb`, that is, unconditionally remove all verbs, then this requirement could not be fulfilled. But so far we have not run into the wall. The second requirement, for *ambiguity* is no problem either: other readings of *w2* have not been targeted, so we are free to choose anything.

As for *conditions*, the part about `(1 noun)` can be fulfilled; nothing prevents *w3* from being a noun. But the first condition, `(-1 det)`, cannot be true, if the target *w2* has to be a verb—the first rule has prohibited exactly that combination. Therefore, when we try to ask for a solution where all these requirements hold, we will get a conflict. The result shows us that the rule `REMOVE verb IF (-1 det) (1 noun)` cannot ever apply, if the rule `REMOVE verb IF (-1 det)` has been already applied.

Example 2: no conflict The previous was a simple example of a conflict: two rules target the same reading, and the first one had broader conditions. Since we had only one preceding rule, we could not demonstrate why is it important to apply all the rules in order—even with a longer list of rules, the previous example could have also worked unordered, as long as the last rule is separated from rules before it. Now we will show another example, where we illustrate why is it important to create new variables, and how conditions can affect the state of the variables, but in a more limited way. The set of readings is the same, and the rules are as follows.

```
r1 = REMOVE verb IF (-1C det) ;
r2 = SELECT det  IF ( 1 verb) ;
r3 = REMOVE verb IF (-1 det) ;
```

The width of the last rule is 2, thus we create a symbolic sentence with only *w1* and *w2*. The sentence is shown below:

```
"<w1>"                  "<w2>"
       "w1" det def            "w2" det def
       "w1" noun sg            "w2" noun sg
       "w1" noun pl            "w2" noun pl
       "w1" verb sg            "w2" verb sg
       "w1" verb pl            "w2" verb pl
```

To begin, let us only look at *r1* and *r3*. Like in the previous example, they target the same reading, but now the order is good: the first rule is the one with a narrower condition, which means that it is possible for a sequence to pass through it, and still have the verb reading in *w2* intact—it just needs to have something else in addition to the determiner in *w1*. The following is an example of such sequence (false readings not shown).

```
"<w1>"                  "<w2>"
       "w1" det def            "w2" noun pl
       "w1" noun sg            "w2" verb sg
```

So far, we have passed through *r1* with the case "condition does not hold, target not removed". Now, let us add *r2*: `SELECT det IF (1 verb)`. We must, in fact, select the determiner and remove everything else: the condition of *r2* happens to be the target of *r3*, so it must hold. But we have the notion of state now, so this is no problem. On arriving to *r1*, the symbolic word *w1* had to be ambiguous—that is, one of the variables $w1_{\text{NSG}}$, $w1_{\text{NPL}}$, $w1_{\text{VSG}}$, $w1_{\text{VPL}}$ must be true. But after passing through *r2*, the state of *w1* has changed: it contains a set of new variables $w1'_{\text{NSG}}$, $w1'_{\text{NPL}}$, $w1'_{\text{VSG}}$, $w1'_{\text{VPL}}$, and all of them must be false. Since the clauses formed by *r1* and *r3* get access to different variables, there is no conflict.

4.2.2 Towards realistic language

Creating realistic readings Earlier we have shown an example with 5 readings ("det def", "noun sg", ...). In a realistic case, we operate between hundreds and thousands of possible readings. In order to find the set of readings, we expand a morphological lexicon[3], ignore the word forms and lemmas, and take all distinct analyses. However, many grammar rules target a specific lemma or word form. A simple solution is to retain the lemmas and word

[3]We used the lexica from Apertium, found in `https://github.com/apertium/apertium-languages`.

forms only for those entries where it is specified in the grammar, and otherwise leave them out. For example, the Dutch grammar contains the following rule:

```
REMOVE ("zijn" vbser) IF (-1 Prep) (1 Noun) ;
```

This hints that there is something special about the verb *zijn*, compared to the other verbs. Looking at the lexicon, we find *zijn* in the following entries:

```
zijn:zijn<det><pos><mfn><pl>     zijn:zijn<vbser><inf>
zijn:zijn<det><pos><mfn><sg>     zijn:zijn<vbser><pres><pl>
```

Thus we add special entries for these: in addition to the anonymous "det pos mfn pl" reading, we add "*zijn* det pos mfn pl". The lemma is treated as just another tag.

However, for languages with more readings, this approach is not feasible. For instance, Spanish has a high number of readings, not only because of many inflectional forms, but because it is possible to add 1–2 clitics to the verb forms. The number of verb readings without clitics is 213, and with clitics 1572. With the previously mentioned approach, we would have to duplicate 1572 entries for each verb. We experimented with small variations with Spanish, and report slightly different results (and significantly different running time) in Section 4.3. Later on, when we applied the tool for Basque (Listenmaa et al., 2017), we were forced to modify the approach even further. We explain the changes in Section 4.4.1.

The readings in a grammar can be underspecified: for example, the rule `REMOVE (verb sg) IF (-1 det)` gives us "verb sg" and "det". In contrast, the lexicon only gives us fully specified readings, such as "verb pres p2 sg". We implemented a version where we took the tag combinations specified in the grammar directly as our readings, and we could insert them into the symbolic sentences as well. The shortcut works most of the time, but if we only take the readings from the grammar and ignore the lexicon, it is possible to miss some cases: e.g. the rule `SELECT (pron rel) IF (0 nom)` may require "pron rel nom" in one reading, but this method only gives "pron rel" and "nom" separately.

In addition, we found that the tag lists in the grammars sometimes contain errors, such as using a nonexistent tag or using a wrong level in a subreading. If we accept those lists as readings, we will generate symbolic sentences that are impossible, and not discover the bug in the grammar. However, if we are primarily interested in rule interaction, then using the underspecified readings from the grammar may be an adequate solution.

Creating realistic ambiguities In the previous section, we have created realistic *readings*, by simply hardcoding legal tag combinations into variables. The next step in creating realistic

	`n nt sg`	`n f pl`	`vblex sep inf`	`det pos mfn`
uitgaven	0	1	1	0
toespraken	0	1	1	0
haar	1	0	0	1

Table 4.1: Ambiguity classes

ambiguities is to constrain which readings can go together. For instance, the case of *zijn* shows us that "determiner or verb" is a possible ambiguity. In contrast, there is no word form in the lexicon that would be ambiguous between an adjective and a comma, hence we do not want to generate such ambiguity in our symbolic sentences.

We solve the problem by creating *ambiguity classes*: groups of readings that can be ambiguous with each other. We represent the expanded morphological lexicon as a matrix, as seen in Table 4.1: word forms on the rows and analyses on the columns. Each distinct row forms an ambiguity class. For example, one class may contain words that are ambiguous between plural feminine nouns and separable verb infinitives; another contains masculine plural adjectives and masculine plural past participles. Then we form SAT-clauses that allow or prohibit certain combinations. These clauses will interact with the constraints created from the rules, and the end result will be closer to real-life sentences.

Our approach is similar to Cutting et al. (1992), who use ambiguity classes instead of distinct word forms, in order to reduce the number of parameters in a Hidden Markov Model. They take advantage of the fact that they don't have to model "bear" and "wish" as separate entries, but they can just reduce it to "word that can be ambiguous between noun and verb", and use that as a parameter in their HMM.

There are two advantages of restricting the ambiguity within words. Firstly, we can create more realistic example sentences, which should help the grammar writer. Secondly, we can possibly detect some more conflicts. Assume that the grammar contains the following rules:

```
REMOVE adj IF (-1 aux) ;
REMOVE pp  IF (-1 aux) ;
```

With our symbolic sentence, these rules will be no problem; to apply the latter, we only need to construct a target that has a realistic ambiguity with a past participle; the adjective will be gone already. However, it could be that past participles (`pp`) only ever get confused with adjectives—in that case, the above rules would contradict each other. By removing the `adj` reading, the first rule effectively selects the `pp` reading, making it an instance of "r selects something in a context, r' removes it". The additional constraints will prevent the SAT-solver from creating an ambiguity outside the allowed classes, and such a case would be caught as a conflict.

4.2.3 Use cases

In the beginning of this chapter, we gave a list of questions, regarding the rules in Figure 4.1. After describing our implementation, we return to these questions, and explain how a SAT-solver can answer them. Section 4.3 shows the evaluation of the conflict detection, which we tried out for four grammars of different sizes. The use cases presented in this section have not been tried in practice; however, our setup makes them fairly straight-forward to implement.

Are all the rules distinct? We gave the example of two rules in the Portuguese grammar, where one had the condition `(-1 Prep)` and the other `(-1 Para OR De)`. A grammarian who has some knowledge of Portuguese could tell that *para* and *de* are both prepositions, so these two rules seem to do almost the same thing.

How can we verify this? To start, we apply all candidate rules to the same initial symbolic sentence. For the resulting symbolic sentence, we can ask for solutions with certain requirements. For instance, "give me a model where a reading $a \notin$ `Inf` is true". There are fewer such models allowed after `SELECT Inf IF (-1 Prep)`—if a non-infinitive reading is in place, it means that the previous word cannot be any determiner—and they are all in the set of models allowed `SELECT Inf IF (-1 Para OR De)`. Thus, we can show that the first rule implies the second. It is in the hands of the grammar writer to decide whether to keep the stronger or the weaker rule, or if they should be placed in different sections.

Could two or more rules be merged? This example concerns two or more rules with the same target but different conditions in the same position. All the six rules in Figure 4.1 have conditions in position -1; four of them also in 0. Some of them have an additional careful context, and some do not. A grammar writer might think that `IF (-1C Vbmod) (0C V)` and `IF (-1C Vbmod) (0 Ser)` are very similar, and could be merged into `SELECT Inf IF (-1C Vbmod) (0C V OR Ser)`. But could there be some unintended side effects in merging these two rules? Why is there a C in the first rule, but not in the second rule?

The procedure is similar to the previous one. We initialise two symbolic sentences; on one we run the original rules in a sequence, and on the other the merged rule. Then, we can perform tests on the resulting symbolic sentences. Assuming that the ambiguity class constraints are in place, the result may show that C is not needed for the condition about *ser*, simply because *ser* is not ambiguous with anything else in the lexicon; hence the merged rule can safely have the condition `(0C V OR Ser)`. However, if *ser* can be ambiguous with something else, then this merged rule is stricter than the two original rules. If the grammar writer is still unsure whether there would be any meaningful sequences that would be missed, they can ask the SAT-solver to generate all those cases—this requires that the ambiguity classes are implemented, otherwise the number of solutions would blow up due to all the readings that are irrelevant to the rules in question.

Generate a sequence that triggers rule(s) R but not rule(s) R'. For any given rule, we can extract three requirements that would make it trigger. We illustrate them for the rule `REMOVE v IF (-1 det)`. In order to trigger the rule, the sequence must have

- target readings: at least one reading with the v tag in the target position
- ambiguity: at least one reading without the v tag in the target position; if there are only v-readings in the target, the would not fire.
- conditions: at least one reading with the *det* tag in the cohort preceding the target.

We can extract these requirements from both the rules that must be triggered, and the rules that must be not triggered. Then, we can request a solution from the SAT-solver.

What is the best order for the rules? For a small number of rules, we can generate variants of the rule order, and run them to a set of symbolic sentences. Then, for the resulting symbolic sentences, we can query which one is the most restricted, that is, allows the least models. As a variant, we may be interested in the placement of just one rule in the sequence.

Does a rewritten rule correspond to the original? We can initialise two symbolic sentences, then run the original rule and the rewritten rule, and check if they allow the same models.

4.3 Evaluation

We tested two grammars with the setup described in Section 4.2.2: readings as combinations of lexical and morphological tags, and ambiguity class constraints in place. These grammars were very small: Dutch[4], with 59 rules and Spanish[5], with 279 rules.

[4]`https://github.com/apertium/apertium-nld/blob/master/apertium-nld.nld.rlx`, commit d13fdaa
[5]`https://github.com/apertium/apertium-spa/blob/master/apertium-spa.spa.rlx`, commit 1d4f290d

In addition, we applied variations of the method to two grammars, which were both larger and targeted more complex languages: Finnish[6], with 1185 rules, and Basque[7] with 1261 rules. Due to the size of the grammars and tagsets, we had to apply some shortcuts; these are described in the relevant sections.

We left out MAP, SUBSTITUTE and other rule types introduced in CG-3, and only tested REMOVE and SELECT rules. The results for Dutch and Spanish are shown in Table 4.5, and the results for Finnish in Table 4.3. For Basque, we were unable to analyse the whole grammar; instead we worked in smaller batches.

The experiments revealed problems in all grammars. For the smaller grammars, we were able to verify manually that nearly all detected conflicts were true positives—we found one false positive and one false negative, when trying out different setups for the Spanish grammar. We did not systematically check for false negatives in any of the grammars, but we kept track of a number of known tricky cases; mostly rules with negations and complex set operations. As the tool matures and we add new features, a more in-depth analysis will be needed. Another natural follow-up evaluation will be to compare the accuracy of the grammar in the original state, and after removing the conflicts found by our tool.

	NLD	SPA	SPA $^{\text{sep. lem.}}$
# rules	59	279	279
# readings	336	3905	1735
# true positives $^{\text{AC}}$ (internal + interaction)	7 (6 + 1)	45 (21 + 24)	44 (20 + 24)
# true positives $^{\text{no AC}}$ (internal + interaction)	7 (6 + 1)	43 (18 + 25)	42 (17 + 25)
# false positives	0	0	1
🕒 with amb. classes	7 s	1h 46m	23 min
🕒 no amb. classes	3 s	44 min	16 min

Table 4.2: Results for Dutch and Spanish grammars.

4.3.1 Dutch

The Dutch grammar had two kinds of errors: rule-internal and rule interaction. As for rule-internal conflicts, one was due to a misspelling in the list definition for personal pronouns, which rendered 5 rules ineffective. The other was about subreadings: the genitive *s* is anal-

[6] https://github.com/flammie/apertium-fin/blob/master/apertium-fin.fin.rlx, commit 49866f3b

[7] https://github.com/inariksit/cgclean/blob/master/data/gramatika4.4.1_lem.rlx, commit 793e00d1

ysed as a subreading in the Apertium morphological lexicon, but it appeared in one rule as the main reading. There was one genuine conflict with rule interaction, shown below:

```
D1. REMOVE Adv IF (1 N) ;
    REMOVE Adv IF (-1 Det) (0 Adj) (1 N) ;
```

These two rules share a target: both remove an adverb. The problem is that the first rule has a broader condition than the second, hence the second will not have any chance to act. If the rules were in the opposite order, then there would be no problem.

We also tested rules individually, in a way that a grammar writer might use our tool when writing new rules. The following rule was one of them:

```
D2. SELECT DetPos IF (-1 (vbser pres p3 sg)) (0 "zijn") (1 Noun);
```

As per VISL CG-3, the condition `(0 "zijn")` does not require *zijn* to be in the same reading with the target `DetPos`. It just means that at index 0, there is a reading with any possessive determiner, and a reading with any *zijn*. However, the intended action is to select a "det pos *zijn*" all in one reading; this is expressed as `SELECT DetPos + "zijn"`. In contrast, the 0-condition in example D1 is used correctly: the adjective and the adverb are supposed to be in different readings.

Can we catch this imprecise formulation with our tool? The SAT-solver will not mark it as a conflict (which is the correct behaviour). But if we ask it to generate an example sequence, the target word may be either of the following options. Seeing interpretation a) could then direct the grammar writer to modify the rule.

```
a) "<w2>"
       "w2" det pos f sg
       "zijn" vbser inf

b) "<w2>"
       "zijn" det pos mfn pl
```

We found the same kind of definition in many other rules and grammars. To catch them more systematically, we could add a feature that alerts in all cases where a condition with 0 is used. As a possible extension, we could automatically merge the 0-condition into the target reading, then show the user this new version, along with the original, and ask which one was intended.

4.3.2 Spanish

The Spanish grammar had proportionately the highest number of errors. The grammar we ran is like the one found in the Apertium repository (linked on the previous page), apart from two changes: we fixed some typos (capital O for 0) in order to make it compile, and commented out two rules that used regular expressions, because support for these had not been implemented. For a full list of found conflicts, see the annotated log of running our program in `https://github.com/inariksit/cgsat/blob/master/data/spa/conflicts.log`.

We include two versions of the Spanish grammar in Table 4.5: in column SPA, we added the lemmas and word forms as described in Section 4.2.2, and in column SPA$^{\text{sep. lem.}}$, we just added each word form and lemma as individual readings, allowed to combine with any other reading. This latter version ran much faster, but failed to detect an internal conflict for one rule, and reported a false positive for another.

When we added ambiguity class constraints, we found three more internal conflicts. Interestingly, the version with ambiguity classes fails to detect an interaction conflict, which the simpler version reports, because one of the rules is first detected as an internal conflict. We think that neither of these versions is a false positive or negative; it is just a matter of priority. Sometimes we prefer to know that the rule cannot apply, given the current lexicon. However, we may know that the lexicon is about to be updated, and would rather learn about all potential interaction conflicts.

As an example of internal conflict, there are two rules that use `SET Cog = (np cog)`: the problem is that the tag "cog" does not exist in the lexicon. As another example, four rules require a context word tagged as NP with explicit number, but the lexicon does not indicate any number with NPs. It is likely that this grammar has been written for an earlier version, where such tags have been in place. One of the conflicts that was only caught by the ambiguity class constraints had the condition `IF (1 Comma) (..) (1 CnjCoo)`. The additional constraints correctly prevent commas from being ambiguous with anything else.

As for the 25 interaction conflicts, there were only 9 distinct rules that rendered 25 other rules ineffective. In fact, we can reduce these 9 rules further into 3 different groups: 4 + 4 + 1, where the groups of 4 rules are variants of otherwise identical rule, each with different gender and number. An example of such conflict is below (gender and number omitted for readability):

```
s1. # NOM ADJ ADJ
    SELECT A OR PP IF (-2 N) (-1 Adj_PP) (0 Adj_PP) (NOT 0 Det);
```

```
# NOM ADJ ADJ ADJ
SELECT A OR PP IF (-3 N) (-2 N) (-1 Adj_PP) (0 Adj_PP) (NOT 0 Det);
```

In addition, the grammar contains a number of set definitions that were never used. Since VISL CG-3 already points out unused sets, we did not add such feature in our tool. However, we noticed an unexpected benefit when we tried to use the set definitions from the grammar directly as our readings: this way, we can discover inconsistencies even in set definitions that are not used in any rule. For instance, the following definition requires the word to be all of the listed parts of speech at the same time—most likely, the grammar writer meant `OR` instead of +:

s_2. `SET NP_Member = N + A + Det + PreAdv + Adv + Pron ;`

If it was used in any rule, that rule would have been marked as conflicting. We noticed the error by accident, when the program offered the reading "$w2$ n adj det preadv adv prn" in an example sequence meant for another rule.

As with the Dutch grammar, we ran the tool on individual rules and examined the sequences that were generated. None of the following was marked as a conflict, but looking at the output indicated that there are multiple interpretations, such as whether two analyses for a context word should be in the same reading or different readings. We observed also cases where the grammar writer has specified desired behaviour in comments, but the rule does not do what the grammar writer intended.

s_3. `REMOVE Sentar IF (0 Sentar) (..) ;`

```
SELECT PP IF (0 "estado") (..) ;
```

The comments make it clear that the first rule is meant to disambiguate between *sentar* and *sentir*, but the rule does not mention anything about *sentir*. Even with the ambiguity class constraints, the SAT-solver only created an ambiguity where *sentar* in 1st person plural is ambiguous with a lexically unspecified 1st person plural reading. This does not reflect the reality, where the target is only ambiguous with certain verbs, and in certain conjugated forms.

The second case is potentially more dangerous. The word form $estado_W$ can be either a noun ($estado_L$, 'state'), or the past participle of the verb $estar_L$. The condition, however, addresses the lemma of the noun, $estado_L$, whereas the lemma of the PP is $estar_L$. This means that, in theory, there can be a case where the condition to select the PP is already removed.

Currently, the lexicon does not contain other ambiguities with the word form $estado_W$, but we could conceive of a scenario where someone adds e.g. a proper noun $Estado_L$ to the lexicon. Then, if some rule removes the lemma $estado_L$, the rule to select PP will not be able to trigger.

4.3.3 Finnish

The results for the Finnish grammar are shown separately, in Table 4.3. We encountered a number of difficulties and used a few shortcuts, which we did not need for the other grammars—most importantly, not using the ambiguity class constraints. Due to these complications, the results are not directly comparable, but we include Finnish in any case, to give an idea how our method scales up: both to more rules, and more complex rules.

Challenges with Finnish The first challenge is the morphological complexity of Finnish. There are more than 20,000 readings, when all possible clitic combinations are included. After weeding out the most uncommon combinations, we ended up with sets of 4000–8000 readings.

The second challenge comes from the larger size of the grammar. Whereas the Spanish and Dutch had only tens of word forms or lemmas, the Finnish grammar specifies around 900 of them. Due to both of these factors, the procedure described in Section 4.2.2 would have exploded the number of readings, so we simply took the lemmas and word forms, and added them as single readings. In cases where they were combined with another tag in the grammar, we took that combination directly and made it into an underspecified reading: for instance, we included both *aika* and "*aika* n" from the rule `SELECT "aika" + N`, but nothing from the rule `SELECT Pron + Sg`. This method gave us 1588 additional readings.

Finally, we were not able to create ambiguity class constraints—expanding the Finnish morphological lexicon results in 100s of gigabytes of word forms, which is simply too big for our method to work. For future development, we will see if it is possible to manipulate the finite automata directly to get hold of the ambiguities, instead of relying on the output in text.

Results The results are shown in Table 4.3. In the first column, we included only possessive suffixes. In the second column, we included question clitics as well. Both of these readings include the 1588 lemmas and word forms from the grammar. In the third column, we in-

	1 CL + LEM	2 CL + LEM	1 CL + RDS	ONLY RDS
# readings	5851	9494	6657	2394
lexicon + grammar	4263 + 1588	7906 + 1588	4263 + 2394	0 + 2394
# conflicts	214	214	22	22
internal + interaction	211 + 3	211 + 3	19 + 3	19 + 3
🕒 all rules (approx.)	~4h 30min	~9h 30min	~7h 45min	~2h 30min

Table 4.3: Results for Finnish (1185 rules). (1 CL + LEM) 1 clitic + lemmas from the grammar; (2 CL + LEM) 2 clitics + lemmas from the grammar; (1 CL + RDS) 1 clitic + all readings from the grammar; (ONLY RDS) all readings from the grammar.

cluded all the tag combinations specified in the grammar, and in the fourth, we took only those, ignoring the morphological lexicon.

The first two variants reported a high number of internal conflicts. These are almost all due to nonexisting tags. The grammar was written in 1995, and updated by (Pirinen, 2015); such a high number of internal conflicts indicates that possibly something has gone wrong in the conversion, or in our expansion of the morphological lexicon. As for accuracy, adding the question clitics did not change anything: they were already included in some of the 1588 sets with word forms or lemmas, and that was enough for the SAT-solver to find models with question clitics. We left the result in the table just to demonstrate the change in the running time.

The second two variants are playing with the full set of readings from the grammar. For both of these, the number of reported conflicts was only 22. Given the preliminary nature of the results, we did not do a full analysis of all the 214 reported conflicts. Out of the 22, we found 17 of them as true conflicts, but 5 seemed to be caused by our handling of rules with *: all of these 5 rules contain a LINK and multiple *s. On a positive note, our naive handling of the * seems to cover the simplest cases. Some examples of true positives are shown in the following.

```
F1. "oma" SELECT Gen IF (..) (0C Nom) ;

    SELECT Adv IF (NOT 0 PP) (..) ;
```

Both of these are internal conflicts, which may not be trivial to see. The first rule requires the target to be genitive and unambiguously nominative; however, these two tags cannot combine in the same reading. As for the second rule, the definition of `PP` includes `adv` among others—with the sets unfolded, this rule becomes `SELECT adv IF (NOT 0 pp|adv|adp|po|pr) (..)`.

The following two examples are interaction conflicts:

```
F2. REMOVE A (0 Der) ;
    REMOVE N (0 Der) ;
    REMOVE A/N (0 Der) ;
```

This is the same pattern we have already seen before, but with a set of rules as the reason for conflict. The first two rules together remove the target of the third, leaving no way for there to be adjective or noun.

```
F3. SELECT .. IF (-1 Comma/N/Pron/Q) ;
    SELECT .. IF (-2 ..) (-1 Comma) ;
```

The rules above have been simplified to show only the relevant part. The conflict lies in the fact that `Comma` is a subset of `Comma/N/Pron/Q`: there is no way to trigger the second rule without placing a comma in position -1, and thereby triggering the first rule.

4.3.4 Performance

The running time of the grammars ranges from seconds to hours. Note that the times in the Finnish table are not entirely comparable with each other: we were forced to run the tests in smaller batches, and it is possible that there are different overheads, unrelated to the size of the SAT-problem, from testing 50 or 500 rules at a time. Despite the inaccuracies, we can see that increasing the number of readings and adding the ambiguity class constraints slow the program down significantly.

However, many of the use cases do not require running the whole grammar. Testing the interaction between 5–10 rules takes just seconds in all languages, if the ambiguity class constraints are not included. A downside in the ambiguity classes is that generating them takes a long time, and while the overhead may be acceptable when checking the full grammar, it is hardly so when analysing just a handful of rules. We are working on an option to store and reuse the ambiguity class constraints.

4.4 Experiments on a Basque grammar

After the original experiment was done on the previously introduced three languages, we got a chance to test a Basque grammar (Aduriz et al., 1997). We worked together with grammarians, who improved the grammar in various ways: reordering, removing or modifying rules, and updating deprecated tag sets. It is hard to say how much exactly our tool influenced the improved F-score, but nevertheless, the tool was helpful in detecting both internal and interaction conflicts.

The size of the Basque tag set was much larger than for Finnish, and made it impossible to test the whole grammar with our setup. We explain our modifications and report preliminary results.

4.4.1 Changes in the setup

Work on groups of rules In order to get some use out of our tool, we grouped all the rules in the grammar by their targets, and sorted by the complexity of their contextual tests. For instance, the 5 rules that target the tag `ADOIN` will be in the order shown in Figure 4.2: from fewest to most contextual tests, and in the case of same number of tests, preferring those with fewer tagsets. Each of those groups is then tested with the SAT-based method. The original order is lost, but the method is likely to catch superfluous rules, because shorter contexts come first. For instance, the two rules originally on lines 6423 and 6422 will cause a conflict in this new order. The original order wasn't a conflict, according to our definition in Section 4.2, but it certainly looks redundant.

```
SELECT ADOIN IF (1 ARAZI) ;                          # line 7412
REMOVE ADOIN IF (0 IZE) (1C ADJ) ;                   # line 6423
REMOVE ADOIN IF (0 IZE) (1 DET | ADJ | IZE) ;        # line 6433
REMOVE ADOIN IF (0 EZEZAG + ADJ + GEN) (-2C IZE) ;   # line 6319
REMOVE ADOIN IF (0 IZE) (-1C IZE) (1C ADJ) ;         # line 6422
```

Figure 4.2: Rules grouped by target, and ordered by their contextual tests.

Modified handling of lexical forms We implemented an alternative scheme for readings, with variables split in three layers: word forms, lexemes and morphological tags. The layer with morphological tags comes from the lexicon as usual, but now, instead of creating new variables with appropriate lexical tags included (e.g. the word form `"<going>"` added to the morphological tag list `verb present gerund`), we allowed lexical tags to attach to anything. A reading in a symbolic sentence has to have at least morphological tags, and optional lexical tags. The method made it possible to run the tool for Basque, but the resulting readings were mostly nonsensical—imagine a situation in English where a reading combines the word form `"<going>"` with the lexeme `"dog"` and claims that it is a plural possessive pronoun. The method worked for detecting conflicts between rules, but of course, it may have missed some that would have been found by saner readings.

Improve order of the rules with an external tool In addition, we applied a tool for automatically tuning grammars (Bick, 2013): the tool modifies and reorders rules, testing against a gold standard to keep only those changes which improve the results. Our initial hypothesis is that the human-cleaned version will benefit more from tuning than the original grammar. Some bad rules may have only a minor problem, such as a single tag name having changed meaning, and they would be better fixed by updating the obsolete tag, instead of the whole rule being demoted or killed. To test our assumptions, we tune both the original grammar and the human-cleaned versions, continuously comparing the new versions to the original.

4.4.2 Results

We evaluate the grammars with a manually disambiguated corpus of 65,153 tokens/53,429 words, compiled from different sources (Aduriz et al., 2006). We report the original score, and the result from tuning the original grammar, as well as the result of preliminary cleanup. The scores are given using two metrics, differing on the granularity of the tagset.

The Basque tag set is structured in four levels of granularity. As explained in Ezeiza et al. (1998), the first level contains only the main POS, 20 distinct tags, and the fourth level contains several hundreds of tags with fine-grained distinctions, including semantic tags such as animacy. Table 4.4 shows a simplified example of the levels for nouns. On the 4th level, the initial ambiguity is very high: the test corpus has, on average, 3.96 readings per cohort. On the 2nd level, when readings that differ only in higher-level tags are collapsed into one, the initial ambiguity is 2.41 readings per cohort. We follow the scheme for evaluation: assume that we are left with two readings, "Common noun, singular" and "Common noun, plural", and one of them is correct. Evaluation on levels 3 and 4 reports 100 % recall and 50 % precision. Evaluation on levels 1 and 2 ignores the tags from the higher levels, and regards any common noun or noun as correct, hence 100 % for both measures.

Analysis of the results The results of the preliminary evaluation are in Table 4.5. The drop in performance after the preliminary cleanup is most certainly due to ordering—the alphabetical ordering seems to block many rules from firing. We could have done the preliminary fixes on the grammar in the old format, to get more representative scores after the initial cleanup, but we found it easier to work on the grammar directly after grouping and sorting

Level 1	**Level 2**	**Level 3**	**Level 4**
Noun	Common noun Proper noun	Common noun, plural absolutive Common noun, singular ergative Proper noun, plural absolutive Proper noun, singular ergative	Common noun, plural absolutive, animate Common noun, plural absolutive, inanimate ... Proper noun, singular ergative, animate Proper noun, singular ergative, inanimate

Table 4.4: Levels of granularity in the Basque grammar

	All tags (Level 4)			**48 main categories** (Level 2)		
	Rec.	*Prec.*	*F-score*	*Rec.*	*Prec.*	*F-score*
Original grammar	**95.61**	62.99	75.94	**97.48**	84.37	90.45
Tuned original	93.87	68.06	78.91	96.66	86.82	91.48
Preliminary cleanup	94.81	56.56	70.85	96.82	84.13	90.03
Tuned prel.cleanup	93.41	**68.61**	**79.11**	96.41	**87.19**	**91.57**

Table 4.5: Preliminary evaluation on words, excluding punctuation, for levels 4 and 2 of granularity.

the rules, as shown in Figure 4.2. Tuning the cleaned grammar brings the precision up, indicating that more rules get to fire in the tuned order. The difference is most dramatic in the sorted and grouped grammar on the 4th level: the original precision drops from 62 % to 56 %, and goes up to 68 % with the tuning.

The fairest test at this stage is to compare the tuned results of the original and the cleaned grammar. We see the cleaned and tuned grammar slightly outperforming the tuned original; the difference is not large, but we see it as a promising start.

Conflict check Our main problem is the size of the tag set: all possible combinations of tags on level 4 amount to millions of readings, and that would make the SAT-problems too big. We cannot just ignore all tags beyond level 2 or 3, because many of the rules rely on them as contexts.

As a first approximation, we have created a reduced set of 21000 readings, which allows the program to check up to 200 rules at a time before running out of memory. The reduced readings, along with the relaxed lexicon handling (as explained in Section 4.4.1), finds some nontrivial conflicts and redundancies, such as the following:

```
SELECT ADB IF
  (0 POSTPOSIZIOAK-9)
  (-1 IZE-DET-IOR-ADJ-ELI-SIG + INE) ;
SELECT ADB IF
  (0 ("<barrena>")) (-1 INE) ;
```

The problem is that the set `POSTPOSIZIOAK-9` contains the word form "barrena", and the other set contains the tag `INE`; in other words, the latter rule is fully contained in the first rule and hence redundant.

Our second strategy is to reduce the rules themselves: from a rule such as `SELECT Verb + Sg IF (1 Noun + Sg)`, we just remove all tags higher than level 2, resulting in `SELECT Verb IF (1 Noun)`. We also keep all lexical tags intact, but as explained in Section 4.4.1, we allow them to attach to any morphological tags; this may lead to further false negatives, but reduces the size of the SAT-problem. This setup analyses the whole grammar, in the given order, in approximately 1 hour. With the reduced rules, the program would not find the redundancy described earlier, because the problem lies in the 3rd-level tag `INE`. But this approximation found successfully 11 duplicates or near-duplicates in the whole grammar, such as the following:

```
SELECT IZE IF   # line 817
  (0 POSTPOSIZIOAK-10IZE LINK 0 IZE_ABS_MG)
  (-1 IZE-DET-IOR-ADJ-ELI-SIG + GEN) ;
SELECT IZE IF   # line 829
  (0 POSTPOSIZIOAK-10IZE + IZE_ABS_MG)
  (-1 IZE-DET-IOR-ADJ-ELI-SIG + GEN) ;
```

Both of the contextual tests contain 3rd-level tags (`ABS, MG, GEN`), but removing them keeps the sets identical, hence it is not a problem for the conflict check.

Finally, all setups have found some internal conflicts. In order to get a more reliable account, we would need more accurate tagset, beyond the 21000. To be fair, many internal conflicts can be detected by simpler means: using the STRICT-TAGS flag of VISL CG-3 would reveal illegal tags, which are the reason for a large number of internal conflicts. But some cases are due to a mistake in logic, rather than a typo; examples such as the following were easily found by the tool.

```
REMOVE ADI IF (NOT 0 ADI) (1 BAT) ;
SELECT ADI IF (0C ADI LINK 0 IZE) ;
```

The first rule is clearly an error; it is impossible to remove an `ADI` from a reading that does not have one. The conflict likely stems from a confusion between `NOT X` and `(*) - X`. The second rule is not obvious to the eye: the interplay of `0C` and `LINK 0` requires `ADI` and `IZE` in the same *reading*, not just in the same cohort. According to the lexicon, this is an impossible combination, and the rule is thus flagged as an internal conflict.

Tuning So far, the most important use of the tuning has been to overcome the differences in ordering. Given the preliminary nature of the work, we have not tried multiple variations. We used a development corpus of 61,524 tokens and a test corpus of 65,153 tokens; the same which we used to obtain the scores in Table 4.5. We stopped the tuning after 5 iterations, and used an error threshold of 25 % to consider a rule as "good" or "bad".

In the future, as the grammar cleanup advances, we are interested in trying out different settings. Already in our current stage, tuning has clearly improved the precision, for both original and preliminarily cleaned grammars, and for both levels of granularity; it is likely that experimenting with different parameters, we would find a combination that would also improve the recall, like Bick (2013) and Bick et al. (2015) report. However, while tuning improves the grammar's performance, it makes the grammar less human-readable, and continuing the development is harder. Keeping two versions of the grammar might be sensible: one for maintenance, kept in a human-readable order, and other for running, created automatically from the former.

4.5 Generative Constraint Grammar

Symbolic sentences have an attractive side effect: they let us model CG as a generative formalism. This section is a detour from the central theme of analysis and quality control of natural language grammars; the uninterested or time-constrained reader may well skip it.

We view a constraint grammar CG as generating a formal language $\mathcal{L}$ over an alphabet Σ as follows. We encode words $w \in \Sigma^\star$ as a sequence of cohorts, each of which has one of the symbols of w as a reading. A constraint grammar CG rejects a word if, when we pass its encoding through the CG, we get back the cohort `"<REJECT>"`. A constraint grammar CG accepts a word if it does not reject it. We generate the language $\mathcal{L}$ by passing every $w \in \Sigma^\star$ through the CG, and keeping those which are accepted.

As an example, consider the language $a^\star$ over $\Sigma = \{a, b\}$. This language is encoded by the following constraint grammar:

```
LIST A = "a";
LIST B = "b";
SET LETTER = A OR B;
SELECT A;
ADDCOHORT ("<REJECT>")
    BEFORE LETTER
```

```
        IF (-1 (>>>) LINK 1* B);
    REMCOHORT LETTER
```

We then encode the input words as a series of letter cohorts with readings (e.g. `"<l>" "a"`, `"<l>" "b"`), and run the grammar. For instance, if we wished to know whether either word in $\{aaa, aab\}$ is part of the language $a^\star$, we would run the following queries:

Input	**Output**
"<l>" "a"	"<l>" "a"
"<l>" "a"	"<l>" "a"
"<l>" "a"	"<l>" "a"
"<l>" "a"	"<REJECT>"
"<l>" "a"	
"<l>" "b"	

As CG is a tool meant for disambiguation, we can leverage its power to run both queries at once:

Input	**Output**
"<l>" "a"	"<l>" "a"
"<l>" "a"	"<l>" "a"
"<l>" "a" "b"	"<l>" "a"

This is a powerful feature, because it allows us disambiguate based on some formal language $\mathcal{L}$ if we can find the CG which generates it. However, the limitations of this style become apparent when we look at a run of a CG for the language $\{ab, ba\}$:

Input	**Output**
"<l>" "a" "b"	"<l>" "a" "b"
"<l>" "a" "b"	"<l>" "a" "b"

While the output contains the interpretations *ab* and *ba*, it also includes *aa* and *bb*. Therefore, while this style is useful for disambiguating using CGs based on formal languages, it is too limited to be used in defining the language which a CG generates.

In light of the idea of using CGs based on formal languages for disambiguating, it seems at odds with the philosophy of CG to reject by replacing the entire input with a single `"<REJECT>"` cohort. However, for the definition of CG as a formal language, we need some sort of distinctive output for rejections. Hence, we arrive at *two* distinct ways to run generative CGs: the method in which we input unambiguous strings, and output `"<REJECT>"`, which is used in the definition of CG as a formal language; and the method in which we input ambiguous strings, and simply disambiguate as far as possible.

It should be noted that VISL CG-3 (Bick and Didriksen, 2015; Didriksen, 2014) supports commands such as `EXTERNAL`, which runs an external executable. It should therefore be obvious that the complete set of VISL CG-3 commands, at least theoretically, can generate any recursively enumerable language. For this reason, we will investigate particular subsets of the commands permitted by CG. In sections 4.5.1 and 4.5.2, we will restrict ourselves to the subset of CG which only uses the `REMOVE` command with sections, and show this to at least cover all regular languages and some context-free and context-sensitive languages. In the original article (Kokke and Listenmaa, 2017), we address more subsets and complexity classes.

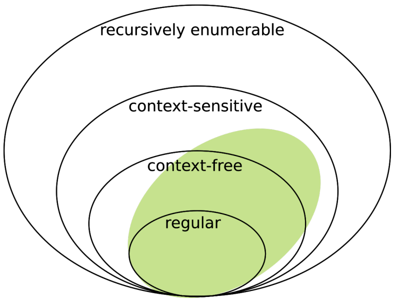


Figure 4.3: Lower bound on the expressivity of the subset of CG using only `REMOVE`.

4.5.1 A lower bound for CG

In this section, we will only use the `REMOVE` command with sections, in addition to a single use of the `ADDCOHORT` command to add the special cohort `"<REJECT>"`, and a single use of the `REMCOHORT` command to clean up afterwards. We show that, using only these commands, CG is capable of generating some context-free and context-sensitive languages, which establishes a lower bound on the expressivity of CG (see Figure 4.3).

Example grammar: $a^n b^n$ Below, we briefly describe the CG which generates the language $a^n b^n$. This CG is defined over the alphabet Σ, in addition to a hidden alphabet Σ'. These hidden symbols are meant to serve as a simple form of memory. When we encode our input words, we tag each cohort with *every* symbol in the hidden alphabet[8], e.g. for some symbol $\ell \in \Sigma$ and $\Sigma' = \{h_1, \ldots, h_n\}$ we would create the cohort `"<`ℓ`>" "h`$_1$`" ... "h`$_n$`"`.

[8] We can automatically add these hidden symbols to our cohorts using a single application of the `ADD` command.

The CG for $a^n b^n$ uses the hidden alphabet {odd, even, opt_a, opt_b}. These symbols mean that the cohort they are attached to is in an even or odd position, and that a or b is a legal option for this cohort, respectively. The CG operates as follows:

1. Is the number of characters even? We know the first cohort is odd, and the rest is handled with rules of the form `REMOVE even IF (NOT -1 odd)`. If the last cohort is odd, then discard the sentence. Otherwise continue…

2. The first cohort is certainly a and last is certainly b, so we can disambiguate the edges: `REMOVE opt_b IF (NOT -1 (*))`, and `REMOVE opt_a IF (NOT 1 (*))`.

3. Disambiguate the second cohort as a and second-to-last as b, the third as a and third-to-last as b, etc, until the two ends meet in the middle. If every "<a>" is marked with opt_a, and every "<b>" with opt_b, we accept. Otherwise, we reject.

The language $a^n b^n$ is context-free, and therefore CG must at least partly overlap with the context-free languages.

Example grammar: $a^n b^n c^n$ We can extend the approach used in the previous grammar to write a grammar which accepts $a^n b^n c^n$. Essentially, we can adapt the above grammar to find the middle of any input string. Once we have the middle, we can "grow" as from the top and bs up from the middle, and bs down from the middle and cs up from the bottom, until we divide the input into three even chunks. If this ends with all "<a>"s marked with opt_a, all "<b>"s marked with opt_b, and all "<c>"s marked with opt_c, we accept. Otherwise, we reject.

The language $a^n b^n c^n$ is context-sensitive, and therefore CG must at least partly overlap with the context-sensitive languages.

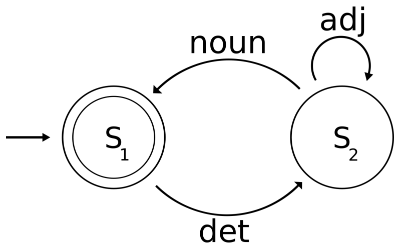


Figure 4.4: A finite-state automaton describing the regular language `det (adj)* noun`.

4.5.2 CG is regular

In the present section, we propose a method to transform arbitrary finite-state automata into CG. We show how any finite-state automaton can be expressed in a CG: encode the states and transitions as ambiguous cohorts, and the disambiguated result shows both the correct sequence and its path in the automaton. The translation is implemented in Haskell, and can be found on GitHub[9].

Finite-state automata Formally, a finite-state automaton is a 5-tuple

$$\langle \Sigma, S, s_0, \delta, F \rangle.$$

Σ is the alphabet of the automaton, S is a set of states, including a starting state s_0 and a set F of final states. δ is a transition function, which takes one state and one symbol from the alphabet, and returns the state(s) where we can get from the original state with that symbol. The automaton in Figure 4.4 is presented as follows:

$$\begin{aligned}
S &= \{\mathtt{s1}, \mathtt{s2}\} & \Sigma &= \{\textit{det, adj, n}\} \\
s_0 &= \mathtt{s1} & \delta &= \{\mathtt{s1} \xrightarrow{\textit{det}} \{\mathtt{s2}\}, \\
F &= \{\mathtt{s1}\} & &\quad\ \mathtt{s2} \xrightarrow{\textit{adj}} \{\mathtt{s2}\}, \\
& & &\quad\ \mathtt{s2} \xrightarrow{\textit{noun}} \{\mathtt{s1}\}\}
\end{aligned}$$

Informally, the automaton describes a simple set of possible noun phrases: there must be one determiner, one noun, and 0 or more adjectives in between. We implement a corresponding CG in the following sections.

Cohorts and sentences We encode our input as a sequence of *state cohorts* and *transition cohorts*. Initially, a state cohort contains the full set $S = \{\mathtt{s1}, \mathtt{s2}\}$ as its readings, and a transition cohort contains the alphabet $\Sigma = \{\textit{det, adj, noun}\}$, or some subset of it. As an example, we generate all 2-letter words recognised by the automaton in Figure 4.4. The initial maximally ambiguous input for length 2 looks as follows:

```
"<s>" "<w>" "<s>" "<w>" "<s>"
 s1    det   s1    det   s1
 s2    adj   s2    adj   s2
       noun        noun
```

[9]See https://github.com/inariksit/cgexp

The grammar disambiguates both transition cohorts and state cohorts. Thus the desired result shows both the accepted sequence(s)—*det noun* in this case—and their path(s) in the automaton.

```
"<s>"  "<w>"  "<s>"  "<w>"  "<s>"
 s1     det    s2    noun    s1
```

We can easily adapt the disambiguation scheme for real-world ambiguities, such as "the present". The state cohorts are identical, but the transition cohorts contain now some actual word form, and the initial ambiguity is not over the whole Σ, but some subset of it.

```
"<s>"  "<the>"  "<s>"  "<present>"  "<s>"
 s1      det     s1        adj        s1
 s2              s2        noun       s2
```

The disambiguation process goes exactly like in the first version, with full Σ in the transition cohorts. Depending on how much the initial input contains ambiguity, the result may be the same, or more disambiguated. For our example, the output is identical.

```
"<s>"  "<the>"  "<s>"  "<present>"  "<s>"
 s1      det     s2        noun       s1
```

Rules Given that every transition happens between two states, and every state has an incoming and outgoing transition, every rule needs only positions -1 and 1 in its contextual tests. The semantics of the rules are "remove a transition, if it is *not* surrounded by allowed states", and "remove a state, if it is *not* surrounded by allowed transitions". For the example automaton, the rules are as follows:

```
# Transition rules                       # State rules
REMOVE Det                               REMOVE S1
    IF (NEGATE -1 S1 LINK 2 S2) ;            IF (NEGATE -1 >>> OR Noun
REMOVE Adj                                           LINK 2 Det) ;
    IF (NEGATE -1 S2 LINK 2 S2) ;        REMOVE S2
REMOVE Noun                                  IF (NEGATE -1 Det OR Adj
    IF (NEGATE -1 S2 LINK 2 S1) ;                    LINK 2 Adj OR Noun) ;
```

The start and end states naturally correspond to the first and last state cohort, and can be trivially disambiguated, in this case both into s1. Once we remove a reading from either side of a cohort, some more rules can take action—the context "s2 on the left side and s1 on the

right side" may be broken by removing either `s2` or `s1`. One by one, these rules disambiguate the input, removing impossible states and transitions from the cohorts.
For the final result of the disambiguation, we consider three options: the cohorts may contain the whole alphabet, a well-formed subset or a malformed subset.

Full Σ If there is only one allowed word of length n in the language, then the result will contain only fully disambiguated transition cohorts. Furthermore, if there is only path in the automaton that leads to this word, then also the state cohorts are fully disambiguated.

If there are multiple words of the same length in the language, then we have to relax our criteria: every transition cohort and state cohort in the result may contain multiple readings, but all of them must contribute to some valid word of length n, and its path in the automaton.

Well-formed subset of Σ With well-formed subset, we mean that each cohort contains at least one of the correct readings: {`det`} for "the", and {`adj,noun`} for "present". If the initial input is well-formed, then the result will be correct, and may even be disambiguated further than with the full Σ in the transition cohorts.

Malformed subset of Σ Malformed subset has at least one cohort without any correct readings, for example, "the" is missing a `det` reading. This will lead to arbitrary disambiguations, which do not correspond to the automaton. Without a `det` reading in "the", the rule which removes `s2` would trigger in the middle state, leaving us with three `s1` states. `s1-s1-s1` is an impossible path in the automaton, so it would trigger all of the transition rules, and stop only when there is one, arbitrary, reading left in the transition cohorts.

4.5.3 Discussion

The FSA→CG conversion is still only an approximation. As Lager and Nivre (Lager and Nivre, 2001) point out, CG has no way of expressing disjunction. Unlike its close cousin FSIG (Koskenniemi, 1990), which would represent a language such as $\{ab, ba\}$ faithfully, CG substitutes uncertainty on the sentence level ("either ab or ba") with uncertainty in the cohorts: "the first character may be either a or b, and the second character may be either a or b". If we use such a CG to generate, by feeding it maximally ambiguous cohorts, the result will be overly permissive. We acknowledge that this is a limitation in the expressive power: many languages can only be approximated by CG, not reproduced exactly. Nevertheless, this limitation may not matter so much when disambiguating real-world text, because the cohorts are initially less ambiguous, and leaving genuine ambiguity intact is desired behaviour for CG.

4.6 Conclusions and future work

We set out to design and implement an automatic analysis of constraint grammars that can find problematic rules and rule combinations, without the need for a corpus. Our evaluation indicates that the tool indeed finds non-trivial conflicts and dead rules from actual grammars.

We did not have a volunteer to test the tool in the process of grammar writing, so we cannot conclude whether the constructed examples are useful for getting new insights on the rules. In any case, there are still a number of features to improve and add. Future work can be divided in roughly three categories: (a) general improvement of the tool (b) evaluation with users, and (c) integration as a part of CG development framework.

4.6.1 General improvement

Combining morphological and lexical tags Our solution to hardcode the tag combinations in the readings is feasible for simple morphology, but it can cause problems with more complex morphology. Currently, if we add one new lemma to the set of readings, we need to create as many new variables as there are inflectional forms for that lemma. The alternative scheme we implemented for Basque resulted in fewer combinations, but as a result, it created nonsensical combinations of wordform, lemma and morphological tags. Investigating whether this is a problem (i.e. does it miss actual conflicts, or suggest false positives) is left for future work.

Heuristic checks for common issues As mentioned earlier, some grammar design choices are common sources of misinterpretation. Many of these issues concern the case where the conditions include the target cohort—does `SELECT foo IF (0 bar)` mean that "foo" and "bar" should be in the same reading or in different readings? Lemmas and word forms are another source of confusion, which is easy to check automatically against the lexicon. Ideally, these checks should be included in a special "paranoid mode", to not clutter the analysis[10].

[10] As pointed out by Eckhard Bick, the program should act upon this only if the 0 relates to an existing ambiguity class.

Support for more features of VISL CG-3 As for longer-term goals, we want to handle more of the features in VISL CG-3, such as map, append and substitute rules, as well as dependency structure. This also means finding different kinds of conflicts, such as dependency circularity. In order to implement rules that may add new readings, or new tags to existing readings, we need to modify our approach in the SAT-encoding. Even if the lexicon gives all readings that exist in the lexicon, the user might give a nonexistent reading, or in the case of map, a syntactic tag, which is (by definition) not in the lexicon. We may need to move to a more scalable solution.

4.6.2 Evaluation with user base

Cleaning up the Basque grammar was our first chance to test the tool together with grammarians, complemented with machine learning (the tuning step). We got to a promising start, but due to the performance problems, we had to simplify our approach considerably. Below, we envision some properties that might be interesting to test further; however, we would be interested in getting feedback from actual grammarians and adding features based on what is needed.

Reformatting a rule Another possible feature is to suggest reformattings for a rule. Recall Figure 4.1 from the introduction; in the case on the right, the original rule was written by the original author, and another grammarian thought that the latter form is nicer to read. Doing the reverse operation could also be possible. If a rule with long disjunctions conflicts, it may be useful to split it into smaller conditions, and eliminate one at a time, in order to find the reason(s) for the conflict.

Suggesting alternative orders On a speculative note, it could be interesting to identify pairs for a potential "feeding order" that is missed in the grammar. Say we have the following rule sequence:

```
REMOVE:r1 x IF (-1C y)
SELECT:s2 y IF (...)
```

If $s2$ appears before $r1$, if makes way for $r1$ to act later on the same round. However, if the rules are ordered as shown, and y is not unambiguous from the beginning, then $r1$ has to wait for the next round to be applied.

Lager (2001) observed that the rule sequence learned by the μTBL system did not remove further ambiguities after its first run, and concluded that the sequence was "optimal". It would be interesting to recreate the goals in Lager (2001) and Bick (2013), to see if this semantic analysis of rule dependencies could lead also to better ordering within a grammar.

Of course, it remains to be seen if any of these improvements would make a difference in speed; VISL CG-3 is already very fast, when the grammars are run multiple times.

Alternative ways of deriving CGs Learning CG rules from a corpus has been a popular topic since the beginning of the formalism (Samuelsson et al., 1996; Eineborg and Lindberg, 1998; Lager, 2001; Sfrent, 2014). As an alternative, our method of transforming regular grammars into an approximate CG (see Section 4.5 could be of interest. To be fair, the resulting grammars are not very readable: they include extra cohorts and symbols, and the logic is spread across rules in a rather obscure way—in contrast to a human-written grammar, where each rule is a self-contained piece of truth about a language. Therefore we do not envision the generated grammars being used as is, but rather as compilation targets. Such CGs could be used as a part of a larger constraint grammar: some sections can be written manually, and others derived from existing grammars. So far we only have a working conversion tool for finite-state automata, but we are hoping to develop this further, to also include context-free or even mildly context-sensitive grammars.

4.6.3 Integration as a part of CG development environment

In order to attract the attention of the CG community, it would be desirable to incorporate the tools as a part of existing CG software. Currently, the described software consists of just under 3000 lines of Haskell code, including both the CG engine and the grammar analysis tool. The grammar used for parsing the original CG files is written in BNFC (Pellauer et al., 2004), and it is missing many constructs in CG-3. Given these factors, the preferred option would be a full reimplementation and integration as a part of VISL CG-3, or any other CG development framework. We believe this would make the tools easier to use, more cohesive and synchronised with the main CG engine, and likely much faster. Of course, it is up to the community and the developers to decide if these tools are of interest.

Chapter 5

Test Case Generation for Grammatical Framework

Fixpoint computation is the new SAT I'm afraid.

Koen Claessen, 2018

What is the *essence* of a language? When formalising and implementing a natural language grammar, which example sentences do we need to check in order to convince ourselves that the grammar is correct?

Imagine we are formalising a grammar for English, and in particular we are working on the reflexive construct. In order to check correctness for the 3rd person singular, we need to test for three different subjects, because the object has to agree with the subject: "he sees himself", "she sees herself" and "it sees itself". Without seeing all three examples (along with the rest of the pronouns), we cannot be certain that the reflexive construction is implemented correctly. In contrast, the general pattern of a transitive verb with a non-reflexive object is enough to test with only one third person subject: *he, she, it*, or any singular noun or proper name. The agreement only shows in the verb form, thus including both "she sees a dog" and "John sees a dog" in the test suite is redundant.

Now, what is minimal and representative is highly language-dependent. For instance, Basque transitive verbs agree with both subject and object, thus we need 6×6 examples just to cover all verb forms. In this chapter, we are not interested in the morphology per se—there are easier methods to test for that—but the correctness of the syntactic function: does the function pick the correct verb form for the correct combination of subject and object? For that purpose, it is enough to test the syntactic construction "transitive verb phrase" with just a single transitive verb.

We present a method that, given a grammar (that in general encompasses an infinite set of sentences), generates a finite set of sentences that can be used as test cases for the correctness of the grammar. Our concrete implementation is for a particular grammar formalism, namely parallel multiple context-free grammars (PMCFG) (Seki et al., 1991), which is the core formalism used by the Grammatical Framework (GF) (Ranta, 2004). However, the general method works for any formalism that is at most as expressive as PMCFG, including context-free grammars, multiple context-free grammars, which covers formalisms such as Tree-Adjoining Grammar (Joshi et al., 1975), Linear Context-free Rewriting Systems (Vijay-Shanker et al., 1987) and Combinatory Categorial Grammar (Steedman, 1988).

The following sections assume knowledge of GF and PMCFG formalisms; we direct the reader to Section 2.2 for a general GF introduction, and 2.2.4 especially for translation of a GF grammar into PMCFG.

5.1 Related work

Traditionally, GF grammars are tested by the grammarians themselves, much in the way described in the introduction of this article. An example human-written treebank can be found in Khegai (2006, p. 136–142). For testing the coverage of the grammars, grammarians have used treebanks such as the UD treebank (Nivre et al., 2016) and Penn treebank (Marcus et al., 1993), and for testing morphology, various open-source resources have been used, such as morphological lexica from the Apertium project (Forcada et al., 2011).

As an example of other grammar formalisms, Butt et al. (1999, pp. 212–213) describe common methods of testing the LFG formalism: similarly to GF, they use a combination of human-written test suites meant to cover particular phenomena, and external larger corpora to test the coverage. As a difference from GF testing tradition, their human-written test suites include also ungrammatical sentences: those that the grammar should *not* be able to parse. However, their tests are only meant for monolingual grammars, whereas GF tests are for multilingual grammars, so they are stored as trees. In other words, GF tests only what the grammar outputs, not what it parses.

Other related work in computational linguistics includes error mining for parsing grammars, such as Gardent and Narayan (2012). The setup includes triples of dependency tree, generated sentence and the gold standard sentence. For every triple where the generated sentence and the gold standard sentence are different, their algorithm finds the smallest subtrees that cause the problems. Gardent's algorithm fills a slightly different need than ours: it relies on the correct linearisation so be known, which we don't. Instead, we want to generate trees whose linearisations are then read by humans. On the other hand, Gardent's method would prove very useful once an error is found—it can be tricky to determine which function exactly caused the error. Further away from our approach, van Noord (2004) works on parsing grammars rather than generation, with the goal of detecting strings that cause parsing errors.

However, our biggest source of inspiration is automatic test case generation for general software, such as Celentano et al. (1980); Geist et al. (1996); Claessen and Hughes (2011). Software testing and grammar testing both deal with notions of coverage and compactness, and deriving test cases from a formal specification. Sometimes even software tests are generated with human oracles in mind, such as in Matinnejad et al. (2016).

5.2 Grammar

```
abstract NounPhrases = {
  flags startcat = NP ;
  cat
    S ; NP ; Adv ;                      -- Non-terminal categories
    CN ; Det ; Adj ; Prep ;             -- Terminal (lexical) categories
  fun
    UttNP   : NP -> S ;                 -- Single NP as an utterance
    PredAdj : NP -> Adj -> S ;          -- e.g. "this house is blue"
    PredAdv : NP -> Adv -> S ;          -- e.g. "this house is on a hill"
    DetNP  : Det -> NP ;                -- e.g. "this"; "yours"
    DetCN  : Det -> CN -> NP ;          -- e.g. "this house"
    PrepNP  : Prep -> NP -> Adv ;       -- e.g. "without the house"
    AdjCN  : Adj -> CN -> CN ;          -- e.g. "small house"
    AdvCN  : Adv -> CN -> CN ;          -- e.g. "house on a hill"

    a, the, this, these, your : Det ;
    good, small, blue, tired, ready : Adj ;
    house, hill : CN ;
    in, next_to, on, with, without : Prep ;
}
```

Figure 5.1: GF grammar for noun phrases

Figure 5.1 shows a small example of a GF abstract grammar. The grammar generates noun phrases for a lexicon of 17 words (*a, the, …, without*) in four lexical categories, and 8 functions to construct phrases. `CN` stands for common noun, and it can be modified by arbitrarily many adjectives (`Adj`), e.g. *small blue house* is an English linearisation of the abstract syntax tree `AdjCN small (AdjCN blue house)`. A `CN` is quantified into a noun phrase (`NP`) by adding a determiner (`Det`), e.g. *the small house* corresponds to tree `DetCN the (AdjCN small house)`. Alternatively, a `Det` can also become an independent noun phrase (as in, *(I like) this* instead of *(I like) this house*) using the constructor `DetNP`. Finally, we can form an adverb (`Adv`) by combining a preposition (`Prep`) with an `NP`, and those adverbs can modify yet another `CN`. We refer to this grammar throughout the chapter.

5.2.1 Examples to test

As examples that help illustrate different testing needs for different languages, let us take four language-specific phenomena in the scope of our small grammar: preposition contraction in Dutch, adjective agreement in Estonian, adjective placement and agreement in Spanish and determiner placement in Basque. Concrete syntaxes for all four languages are found in https://github.com/inariksit/GF-testing/tree/master/data/grammars; however, the chapter can be read without understanding the details of the concrete syntaxes.

Preposition contraction in Dutch In Dutch, some prepositions should merge with a single determiner or pronoun, e.g. *met dit* 'with this' becomes *hiermee* 'herewith', but stay independent when the determiner quantifies a noun, e.g. *met dit huis* 'with this house'. Other prepositions, such as *zonder* 'without', do not contract with any determiners: *zonder dit* 'without this' and *zonder dit huis* 'without this house'. When testing `PrepNP`, we would like to see one preposition that contracts and one that does not, as well as one `NP` that is a single determiner, and one that comes from a noun. Since the result of `PrepNP` is an adverb, which does not inflect any further, we are happy with just finding the right arguments to `PrepNP`, no need for contexts. In order to catch a bug in the function, or confirm there is none, we need the following 4 trees:

$$\texttt{PrepNP}\left\{\begin{matrix}\texttt{with}\\ \texttt{without}\end{matrix}\right\}\left\{\begin{matrix}\texttt{DetNP this}\\ \texttt{DetCN this house}\end{matrix}\right\}.$$

Adjective agreement in Estonian In Estonian, most adjectives agree with nouns in case and number in an attributive position. However, participles are invariable (singular nominative) as attributes but inflect regularly in a predicative position, and a set of invariable adjectives do not inflect in any position. Furthermore, in 4 of the 14 grammatical cases, even the regular adjectives only agree with the noun in number, but the case is always genitive. Table 5.1 shows the different behaviours in attributive position, with *sinine* 'blue' as an example of a regular adjective, and *valmis* 'ready' as an invariable.

Regular		**Genitive agreement**		**Invariable**	
sinises	majas	sinise	majaga	valmis	majas
blue-SG.INE	house-SG.INE	blue-SG.GEN	house-SG.COM	ready.SG.NOM	house-SG.INE
'in a blue house'		'with a blue house'		'in a finished house'	
sinistes	majades	siniste	majadega	valmis	majades
blue-PL.INE	house-PL.INE	blue-PL.GEN	house-PL.COM	ready.SG.NOM	house-PL.INE
'in blue houses'		'with blue houses'		'in finished houses'	

Table 5.1: Estonian adjective agreement

Since we are interested in adjectives, choosing `AdjCN` as the base sounds reasonable—but that only creates an inflection table, so we must think of a context too. Just like in English, number comes from the determiner, so we need to wrap the `CN` in a `DetCN` with two determiners of different number, for instance `this` and `these`. But we still need an example for one of the 10 cases with normal agreement, such as inessive (in something), and one of the 4 cases with restricted agreement, such as comitative (with something). These cases correspond to the English prepositions *in* and *with*, so in this abstract syntax we can use `PrepNP` with the arguments `in` and `with`. This is another showcase of the abstraction level of GF: in the English concrete syntax, `Prep` contains a string such as 'in' or 'with', and `PrepNP` concatenates the string from its `Prep` argument into the resulting adverb, but in Estonian, `Prep` contains a case, and `PrepNP` chooses that case from its `NP` argument. The following set of 8 trees creates all the relevant distinctions:

$$\texttt{PrepNP}\begin{Bmatrix}\texttt{in}\\\texttt{with}\end{Bmatrix}(\texttt{DetCN}\begin{Bmatrix}\texttt{this}\\\texttt{these}\end{Bmatrix}$$

$$\texttt{AdjCN}\begin{Bmatrix}\texttt{blue}\\\texttt{ready}\end{Bmatrix}\texttt{house}).$$

Adjective placement and agreement in Spanish Spanish adjectives agree in number and gender with the noun, in both attributive and predicative position. In attributive position, most adjectives (e.g. `small`) come after its head, but some adjectives (e.g. `good`) are placed before the head. In order to test `AdjCN`, with regards to both adjective placement and agreement, we need the following 8 trees:

$$\texttt{DetCN}\begin{Bmatrix}\texttt{this}\\\texttt{these}\end{Bmatrix}(\texttt{AdjCN}\begin{Bmatrix}\texttt{good}\\\texttt{small}\end{Bmatrix}\begin{Bmatrix}\texttt{house}\\\texttt{hill}\end{Bmatrix}).$$

Determiner placement in Basque In Basque, there are three different ways to place a determiner into a noun phrase. When a number (other than 1) or a possessive pronoun acts as a determiner in a complex noun phrase, it is placed between "heavy" modifiers, such as adverbials or relative clauses, and the rest of the noun phrase. Demonstrative pronouns, such as *this*, are placed after all modifiers as an independent word. Number 1, which functions as an indefinite article, acts like demonstratives, but the definite article is a suffix. If there is a "light" modifier, such as an adjective, the definite article attaches to the modifier; otherwise it attaches to the noun. In order to test the implementation of this phenomenon, we need the following 12 trees:

$$\texttt{DetCN} \left\{\begin{array}{c}\texttt{the}\\ \texttt{this}\\ \texttt{your}\end{array}\right\} \left\{\begin{array}{c}\texttt{AdvCN on (DetCN the hill)}\\ \varnothing\end{array}\right\} \left\{\begin{array}{c}\texttt{AdjCN small}\\ \varnothing\end{array}\right\} \texttt{house.}$$

5.3 Using the tool

Here is a typical use for the tool. Let us take the noun phrase grammar for Dutch, and pick a single function, say `AdvCN`. We generate test cases, which include the following trees:

```
AdvCN (PrepNP next_to (DetNP your)) hill
```
'hill next to yours'

```
AdvCN (PrepNP next_to (DetNP your)) house
```
'house next to yours'

In Dutch, the words *hill* and *house* have different genders, and the word *yours* has to agree in gender with the antecedent: *(de) heuvel naast de jouwe* and *(het) huis naast het jouwe*. Say that the test cases reveal a bug, where `DetNP your` picks a gender too soon—always neuter, "het jouwe", instead of leaving it open in an inflection table. To fix the bug, we add gender as a parameter to the `Adv` type, and have `AdvCN` choose the correct form based on the gender of the `CN`.

After implementing the fix, we run a second test case generation: this time, not meant for human eyes, but just to compare the old and new versions of the grammar. We want to make sure that our changes to `Adv`, `AdvCN` and `DetNP` have not caused new bugs in other categories and functions. The simplest strategy is to generate test cases for *all* functions in both grammars, and only show those outputs that differ between the grammars. After our changes, we get the following differences:

```
DetCN the (AdvCN (PrepNP next_to (DetNP your)) hill)
```

– ~~*de heuvel naast* **het** *jouwe*~~

+ *de heuvel naast* **de** *jouwe*

```
DetCN the (AdvCN (PrepNP without (DetNP this)) hill)
```

– ~~*de heuvel zonder* **dit**~~

+ *de heuvel zonder* **deze**

We notice a side effect that we may not have thought of: the gender is retained in all adverbs made of NPs made of determiners, so now it has become impossible to say "the hill next to *that*", where *that* is referring to a house. So we do another round of modifications, compute the difference (to the original grammar or to the intermediate), and see if something else has changed.

Additional analyses Aside from concrete language-dependent phenomena, there are more general questions a grammar writer may ask. For instance, say that our concrete type for `CN` in some language is an inflection table from case to string, we would like to know if (a) a given string field is unreachable from the start category; (b) any two fields always contain the same string; or (c) some fields are always the empty string. In addition, a whole argument may be erased by some function: say that `AdjCN : Adj` $\rightarrow$ `CN` $\rightarrow$ `CN` never adds the adjective to the new `CN`, in which case `AdjCN blue house` and `house` are linearised identically. Instead of testing every single function and finding out by accident, we can test the whole grammar to find out if there are any functions in the grammar that behave like this.

5.4 Generating the test suite

In this section, we explain how the test suites are built. Earlier in Section 2.2.4, we saw how a single abstract category compiles into multiple concrete categories N^C, depending on the combinations of parameters. Instead of dealing with parameters directly, we now have a set of new *types*, which is helpful for generating test cases.

We use one syntactic function as the base for one set of test cases. For lexical categories, it also makes sense to test the whole category, e.g. generate all trees that show that `AP` is defined and handled correctly in the functions. However, we only explain in detail the method with one syntactic function as a base.

We require that all test cases are trees with the same start (top-level) category, such as `S` in our example grammar. Furthermore, the start category must be linearised as one string only.

5.4.1 Enumerate functions

As we explained before, each syntactic function in the abstract syntax turns into multiple versions, one for each combination of parameters of its arguments. In the notation by Angelov (2011), presented in Section 2.2.4, these concrete functions are the set F^C.

In order to construct trees that use the concrete syntactic function $f \in F^C$, we need to supply the function with $a(f)$ *arguments*, as well as put the resulting tree $f(A_1, \ldots, A_{a(f)})$ into a *context* that produces a tree in the start category. In the best case, each f can be tested with only one tree, but as we will explain in the following section, sometimes we need more than one tree to test f, if no single argument list is discriminative enough.

5.4.2 Enumerate arguments

Some syntactic functions are simply a single lexical item (for example the word *good*); in this case just the tree `good` is our answer. If we choose a function with arguments, such as `PrepNP`, then we have to supply it with argument trees. Each argument needs to be a tree belonging to the right category (in the example, `Prep` and `NP`, respectively).

When we test a function, we want to see whether or not it uses the right material from its arguments, in the right way. The material that a syntactic function uses is any of the strings that come from linearising its arguments. In order to be able to see which string in the result comes from which field from which argument, we want to generate test cases that only contain unique strings. For example, the English pronoun *you* is a worse test case than *they*, because all forms are not unique: *you* is both nominative and accusative (*you* sleep vs. I saw *you*), whereas *they* has a different form for both (*they* sleep vs. I saw *them*).

Subsumption For our testing purposes, any other pronoun with all unique forms *subsumes* the pronoun *you*: if we can construct a test with {*they*,*them*,*their*}, we don't need the inferior {*you*,*you*,*your*} in addition. In general, we would like to find just one combination of arguments where all strings in the linearisations are different. However, it is not always possible to do this, which is why we in general aim to generate a set of combinations of arguments, where for each pair of strings from the arguments, there is always one test case where those strings are different. If all pronouns in English were non-unique, we could still get the effect by using both {*you*,*you*,*your*} and {*she*,*her*,*her*}, because their ambiguity manifests in different forms[1].

We define the function subsumes for a given function f and two candidate lists of argument trees. It is important to include the function f and not just its arguments, because any GF function may introduce new strings to the linearisation. If f introduces some string s, then any argument that contains s is worse than an otherwise identical argument without s.

In addition to Angelov (2011)'s definition, we use the notion of Tree as a generic type for any application of some $f \in F^C$ to all of its arguments $[A_1, \ldots, A_{a(f)}]$. Thus, a single $f \in F^C$ is also a Tree, if $a(f) = 0$. For an f with $a(f) > 0$, we need to apply f to all of its argument Trees of types $A_1, \ldots, A_{a(f)}$.

```
linAll :: Tree → [String]
linAll ($f(A_1, \ldots, A_{a(f)})$) = **evaluate the expression** $f(A_1, \ldots, A_{a(f)})$,
                                      **return all string fields.**

subsumes : $F^C$ → [Tree] → [Tree] → Bool
subsumes $f$ xtrees ytrees =
    **let** syncats = **strings introduced by** $f$
        xlins = syncats ++ concatMap linAll xtrees
        ylins = syncats ++ concatMap linAll ytrees
        len = min xs ys
    **in** and [ ylins !! $i$ == ylins !! $j$
            | $i$ ← [0..len-1], $j$ ← [0..len-1]
            , xlins !! $i$ == xlins !! $j$ ]
```

In words, we gather all strings from the function f and its arguments, and check that for all indices i and j, if the i^{th} element of xlins is the same as the j^{th} element of xlins, then also the i^{th} and j^{th} element from ylins must be the identical. If this property holds, then xtrees subsumes ytrees. For instance, the list [a,a,b] subsumes [c,c,d] (and vice versa), because the indices 0 and 1 are identical in both lists.

Argument generation and filtering For a given $f \in F^C$, we generate all lists of argument trees $[A_1, \ldots, A_{a(f)}]$. The function FEAT.generateAllTrees is implemented using the FEAT library (Duregård et al., 2012), and generates lazily all vectors of argument trees for the given

[1]The majority of all functions in all tested languages need 1 or 2 argument trees, and there are only a handful of cases that need more than 3. The highest number we encountered was 8 trees, for a single function in the Basque resource grammar.

concrete categories, starting from smallest size. The size of a tree is defined as the number of constructors; thus `DetNP this` is generated before `DetCN this house`. This ensures that we find the most minimal examples.

```
FEAT.generateAllTrees :: $[N^C] \to [[\mathsf{Tree}]]$
FEAT.generateAllTrees $[\mathsf{a}_1, \ldots, \mathsf{a}_n]$ = generates lazily all vectors of argument
                                          trees for the given concrete categories,
                                          starting from smallest.
```

Out of all argument lists, we find the ones to test *f* using the previously defined function subsumes. The function keepIfBetter takes a new tentative test case (t), a list of already good test cases (ts), and checks whether any of the test cases in ts subsumes t. If yes, we can ignore t and keep ts intact; if no, we add t to the list and remove from ts any test cases that t subsumes.

```
keepIfBetter :: $F^C \to [\mathsf{Tree}] \to [[\mathsf{Tree}]] \to [[\mathsf{Tree}]]$
keepIfBetter $f$ t ts = if any ($\lambda x.$ subsumes $f$ $x$ t) ts
                        then ts
                        else  t : filter (not $\circ$ subsumes $f$ t) ts
```

The main logic, in mostUniqueExamples, is implemented as a fold over a list of lists of argument trees. The algorithm starts with an empty list as an accumulator, and applies keepIfBetter repeatedly, until a desired combination of trees is found, or until the original list has run out.

```
mostUniqueExamples :: $F^C \to [[\mathsf{Tree}]] \to [[\mathsf{Tree}]]$
mostUniqueExamples $f$ vtrees = foldr (keepIfBetter $f$) [] vtrees
```

Putting it all together The set of most unique argument trees for some $f \in F^C$ is computed by the function bestTrees. For an *f* with arguments $[A_1, A_2, \ldots, A_{a(f)}]$, the Trees in the inner lists are of types $[A_1, A_2, \ldots, A_{a(f)}]$.

```
bestTrees :: $F^C \to [[\mathsf{Tree}]]$
bestTrees $f$ =
    let  args = $[A_1, A_2, \ldots, A_{a(f)}]$
                s.t. $A \to f[A_1, A_2, \ldots, A_{a(f)}]$ is a production in $P$.
    in   mostUniqueExamples $f$ (FEAT.generateAllTrees args)
```

The list coming from FEAT.generateAllTrees is infinite, but in practice, we stop after examining 10,000 lists of trees.

`AdjCN good house` (SG) buena casa (PL) buenas casas	`AdjCN good hill` (SG) buen cerro (PL) buenos cerros
`AdjCN small house` (SG) casa pequeña (PL) casas pequeñas	`AdjCN small hill` (SG) cerro pequeño (PL) cerros pequeños

Table 5.2: Agreement and placement of Spanish adjectives in attributive position

Example: Test cases using `AdjCN` Let us test the function `AdjCN : Adj → CN → CN` in Spanish concrete syntax. Firstly, we need a minimal and representative set of arguments: one premodifier and one postmodifier `AP` (`good` and `small`), as well as one feminine and one masculine `CN` (`house` and `hill`). Now, our full set of test cases are `AdjCN` applied to the cross product of $\{\substack{\texttt{good}\\\texttt{small}}\} \times \{\substack{\texttt{house}\\\texttt{hill}}\}$, as seen in Table 5.2.

5.4.3 Enumerate contexts

The third and last enumeration we perform when generating test cases is to generate all possible *uses* of a function. After we provide a function with arguments, we need to put the resulting tree into a context, so that we can generate a single string from the result. By *context* for a given category `C`, we mean a function of type `C → S`. From another point of view, a context can be seen as a tree in the start category (`S` in our example) with a *hole* of the same type as the function under test returns (denoted by _).

Figure 5.2 shows a concrete example for the category `CN` in our example grammar. Since `CN` is variable for number, we need two contexts: one that chooses the singular form and other that chooses the plural form. This means we should apply two different `CN → NP` functions

`UttNP (DetCN this (AdjCN good house))` esta buena casa	`UttNP (DetCN this (AdjCN good hill))` este buen cerro
`UttNP (DetCN these (AdjCN good house))` estas buenas casas	`UttNP (DetCN these (AdjCN good hill))` estos buenos cerros
`UttNP (DetCN this (AdjCN small house))` esta casa pequeña	`UttNP (DetCN this (AdjCN small hill))` este cerro pequeño
`UttNP (DetCN these (AdjCN small house))` estas casas pequeñas	`UttNP (DetCN these (AdjCN small hill))` estos cerros pequeños

Table 5.3: Complete test cases to test `AdjCN`

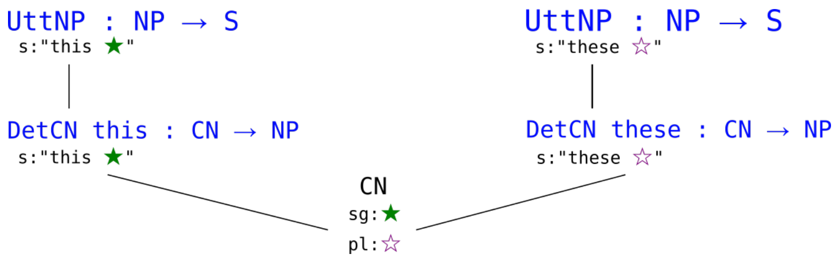


Figure 5.2: Two contexts for the category `CN`, which choose all forms from the inflection table.

(Figure 5.2 shows `DetCN this` and `DetCN these`), and give their results to the `UttNP` function, which constructs an `S`.

We could as well pick any of the `Pred*` functions, which also has the goal type `S`, but their second arguments don't make any difference as to which field we choose from the `NP`. In the name of minimalism, we choose `UttNP`, because it is the smallest tree. The concrete set of contexts is thus `UttNP (DetCN this _)` and `UttNP (DetCN these _)`. We insert the 4 test cases from Table 5.2 into the holes, and get 8 trees in total as shown in Table 5.3.

Strings' path to the start category A subtree in any category `C` gets to show its strings to the world by one way: the strings need to end up in a tree of the start category. If `C` is the start category, then the subtree of type `C` needs no further context. If `C` is not the start category, then the subtree depends on other categories in order to make it to the start category. We need to find a succession of function applications that take a string from some field in `C`, through all other intermediate categories, so that it finally ends up in the start category.

The word *depend* is a bit counterintuitive here: normally one would think that e.g. `NP` depends on `CN`, because we form `NP`s by using functions of type `CN → NP`. But when we think of contexts, we say instead that the *context of* `CN` depends on the *context of* `NP`. A tree of type `CN` would like to show its strings, but the function `UttNP`, which is the fastest way to the start category, only takes `NP`s. So it has to rely on applications of `DetCN this` and `DetCN these` to lift it up into an `NP`, as shown in Figure 5.2, and only then can `UttNP` get access to its strings.

Contexts as systems of equations Computing relevant contexts in a given start category `S` is done once, in advance, for all possible hole types H at the same time, using fixpoint iteration. We model this in a top-down manner, where in the beginning, only the start category `S` has a context, albeit a trivial one: "do nothing, you're already there!". The other categories don't

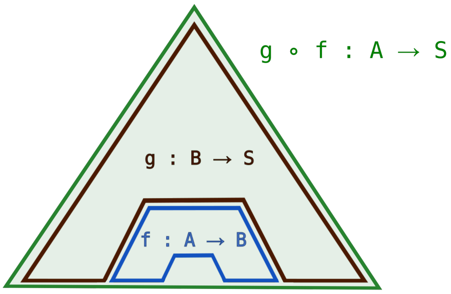


Figure 5.3: Context for `A` in start category `S` depends on a context for `B`.

have a context yet, but will get them during the process in the following manner, referencing Figure 5.3.

On the first round, we get to compute contexts for all categories `B` that are arguments to some function `g : B → S`: these contexts are simply the function `g` itself. On the second round, we compute contexts for all categories `A` that are arguments to some function `f : A → B`; the context for such an `A` is `g ∘ f`. Each round unlocks more categories further down from the start category, so we get to compute more contexts for categories that depend on the newly unlocked ones, until all categories have their contexts. Figure 5.3 shows the general pattern for categories `A` and `B` in start category `S`.

The definition of contexts is mutually recursive, and can be seen as a system of equations. The simple example with `f : A → B` and `g : B → S` is expressed as follows.

$$\begin{aligned}\mathsf{contexts}(\mathtt{S}) &= \{\ id\ \}\\ \mathsf{contexts}(\mathtt{B}) &= \{\ C[g(\_)] \mid C \in \mathsf{contexts}(\mathtt{S})\}\\ \mathsf{contexts}(\mathtt{A}) &= \{\ C[f(\_)] \mid C \in \mathsf{contexts}(\mathtt{B})\}\end{aligned}$$

Least solution In the general case, these sets can be infinite: think of functions of type `A → A` and loops of `A → B` and `B → A`. In such a case, we want to find the *least* solution to such systems of equations. The word *least* means two separate things: (a) the fewest trees that still use all the fields of the category, and (b) which are of the smallest size (e.g. choosing `UttNP` over `PredAdj` for the context of `NP`).

In contrast to the more intuitive bottom-up approach, top-down computation is more efficient: we retrieve the fastest way to the start category, because at each round, we consider contexts that we already know to lead into the start category. We avoid functions of type `B → A` when computing the context for `B`, because we only consider functions whose goal type *already has a context*; hence `g : B → S` is the only candidate.

Algorithm We express the set of relevant contexts for one hole type H in terms of the sets of relevant contexts for other hole types H', using the following definition:

$$\mathsf{contexts}(H) = \mathsf{prune}(\ \{C[F(\_)] \mid F \in H \to H',\ C \in \mathsf{contexts}(H')\}\ )$$

In words, to generate contexts with holes of type H, we enumerate all functions F that have a H as an argument, and enumerate all contexts C that have the result type H' of F as a hole type, and put C and F together to get a new context. Then, we apply a function prune to the result in order to filter out redundant contexts, i.e. contexts whose uses of the strings of H are already covered by other contexts in the same set.

In order to compute all sets of contexts for all possible hole categories H, we set up a system of equations. In general, this system of equations is recursive, and we use a fixpoint iteration to solve it, initialising the contexts as follows: id for the start category, and an empty set for all other categories. There is a guaranteed minimal solution, because the RHSs are monotonic in H' (Lassez et al., 1982).

$$\begin{aligned}
&\mathsf{contexts}(S) = \{\ id\ \} \\
&\mathsf{contexts}(H) = \{\ \} \quad : \forall H \neq S
\end{aligned}$$

It is important that we do not change the contexts for anything that uses fewer or the same amount of fields. In the English example, it would not serve any purpose to swap `PredAdj` for `AdjCN` in the context of `AP`, because both choose the same (the only) field. Furthermore, if we changed the contexts on an arbitrary basis, the sets would not converge, and we wouldn't find a fixpoint.

However, the functions can be replaced in the case that we find a function that covers strictly more fields than any of the previously found. Assume that we added a whole new abstract syntax construction of type `CN` $\to$ `NP`, which uses both singular and plural in one sentence. Then we get just one context which covers two fields, and thus we can throw away both `DetCN this` and `DetCN these` in favour of the new construction.

So, the set of contexts for a category H may change during the fixpoint computation. It grows as we unlock contexts for more categories, further down from the start category. It may also shrink, if we find a single context that covers the fields from multiple, previously computed, contexts. But most importantly, the list of fields from H that make it into the start category is growing throughout the fixpoint computation.

$NP_{sg,noncontr}$	$NP_{pl,noncontr}$	$NP_{sg,contr}$	$NP_{pl,contr}$
UttNP _	UttNP _	UttNP _ PredAdv <any NP> (PrepNP on _)	UttNP _ PredAdv <any NP> (PrepNP on _)

Table 5.4: Contexts to showcase all strings of different concrete NPs in the Dutch grammar

5.4.4 Pruning the trees within one set of test cases

Sometimes we can further reduce test cases created for a single function, taking advantage of the coercions in the grammar (explained in Section 2.2.4). Let us take as an example the function `DetNP : Det → NP` for Dutch. As usual, the category `Det` compiles into 8 concrete categories: combinations of singular vs. plural, contracting vs. non-contracting, and definite vs. indefinite. But our grammar only has 5 determiners in total, out of 8 possible combinations—does this mean that we cannot test `DetNP` exhaustively with the small lexicon? Firstly, we look for coercions of *arguments* (`Det`) in the different concrete versions of `DetNP`, and secondly, coercions of the *goal category* (`NP`) used in the contexts.

Coercions of `Det` used by `DetNP` Due to coercions, there are only 4 different type signatures for `DetNP`: definiteness is not relevant when a `Det` is made directly into an `NP` (it only matters when combining a determiner with an adjective). This brings us down to 4 arguments for `DetNP`, shown below.

- DetNP : $Det_{*,sg,noncontr} \rightarrow NP_{sg,noncontr}$
- DetNP : $Det_{*,pl,noncontr} \rightarrow NP_{pl,noncontr}$
- DetNP : $Det_{*,sg,contr} \rightarrow NP_{sg,contr}$
- DetNP : $Det_{*,pl,contr} \rightarrow NP_{pl,contr}$

Contexts for `NP` The next step is to look at the *contexts* for all goal categories of `DetNP`, as shown in Table 5.4. The contracting `NP`s get two contexts, one to choose the full form and other the contracted form. The non-contracting ones only get one context each, because their field for the contracted form is empty.

Coercions of `NP` used in contexts In addition, we look for *coercions* for all the goal categories. For instance, there is one that covers all `NP`s; let us call that NP_*. This tells us that there are functions that take `NP`, but don't care anything about its parameters. One such example is `UttNP`, which just takes an `NP`, chooses the nominative form and makes it into a standalone utterance. There is another useful coercion, which covers both singular and plural contracting `NP`s; we call that $\texttt{NP}_{*,\mathit{contr}}$. It is used by `PrepNP`[2], which doesn't care about number, just whether to merge its arguments or not. In order to reduce the number of test cases for any function `f : A → B`, two properties need to hold:

1. A number of concrete categories for `B` are interchangeable in a particular context;
2. That context is one of those chosen to showcase `B`.

With $\texttt{NP}_{*}$ and $\texttt{NP}_{*,\mathit{contr}}$, both of these properties hold. Thus, we only need to put one representative of *any* `NP` into the context `UttNP _`, and one representative of the contracting `NP`s into the context `PredAdv <any NP> (PrepNP on _)`. The following trees would fit the description:

```
UttNP (DetNP this)
```
deze
'this'

```
PredAdv (DetCN a house) (PrepNP on (DetNP this)
```
een huis hierop
'a house on this'

5.4.5 Pruning the trees to test the whole grammar

So far we have completely ignored that one tree can showcase more than one function. In the Estonian example, we start by testing `AdjCN` and end up with the following 8 trees, where `PrepNP` is in the context:

$$\texttt{PrepNP}\left\{\begin{matrix}\texttt{in}\\\texttt{with}\end{matrix}\right\}\texttt{(DetCN}\left\{\begin{matrix}\texttt{this}\\\texttt{these}\end{matrix}\right\}\texttt{AdjCN}\left\{\begin{matrix}\texttt{blue}\\\texttt{ready}\end{matrix}\right\}\texttt{house).}$$

In fact, these 8 trees cover all tests we would've needed for `PrepNP` itself. Thus, it is possible to shrink the test cases, if one wants to test the whole grammar at one go. However, such an approach has also a significant drawback: interpreting the test results may get more difficult, because a bug can come from many different functions.

Deterministic argument generation There is a simple way to detect redundancy in the generated trees. We make the generation of arguments and contexts completely deterministic, e.g. always choose the function that is alphabetically first. Then, we would get a sentence

[2]Along with a similar coercion $\texttt{NP}_{*,\mathit{noncontr}}$ for non-contracting `NP`s.

such as “the good house is good” for testing both `AdjCN` and `PredAdj`. The downside is that the sentences can get boring to read, or even confusing, in the style of “the house gives the house the house”.

However, we can split the generation in two stages: first stage is deterministic, where every noun is `house`, and we can eliminate redundancies by just eliminating copies of the same tree. Then, when we have a set of unique trees, we can substitute individual words in them with other words in the same concrete category. A sentence such as “the house gives the hill the ham” tests the same properties as the version with *house* in every role, but at least it is easier to keep track who does what, and compare the translations of the same tree.

In practice, we only apply the first step, and let the sentences be redundant. In Section 5.5, we refer to total and unique trees when generating tests for the whole grammar; “unique trees” means the number after removing duplicate trees that are generated to test different functions, but happen to be the same.

Alternative priorities for context generation The current implementation of context generation prioritises minimal trees, such as `UttNP` for `NP` instead of any other functions. As an alternative, we could choose contexts where the `NP` can also use its parameters, in addition to just showing all of its strings. Arguably, it would tell more to the user to see a sentence like “this house is big”, formed with `PredAdj`, rather than just “this house”, formed by `UttNP`.

However, these alternative priorities are not implemented yet. With the current scheme, we have found it a useful practice to hide functions such as `UttNP`, which just lift a category into the start category without introducing other arguments. For future, it would make sense for a context to serve double purpose: show all of its strings and (try to) use all of its parameters.

5.5 Evaluation

	Concrete syntax →		**Dutch**		**Spanish**		**Estonian**		**Basque**	
↓ Grammar	#funs+lex	#trees	#total	#uniq	#total	#uniq	#total	#uniq	#total	#uniq
Noun phrases	8+17	>10,000	21	18	16	15	33	27	40	36
Phrasebook	130+160	>480,000	513	419	504	382	610	505	538	503
Resource gr.	217+446	>500 billion	21,370	19,825	19,689	16,662	13,733	9,194	100,967	64,390

Table 5.5: Test cases for all functions in three grammars

In order to evaluate our method, we generate test cases for grammars of varying sizes, using the four languages presented earlier: Dutch, Spanish, Estonian and Basque. These

Resource grammar function		Dutch	Spanish	Estonian	Basque
`ComparA : A → NP → AP`	'stronger than you'	10	4	20	6
`RelNP : NP → RS → NP`	'a cat that I saw'	2	23	23	20
`ComplVS : VS → S → VP`	'say that I sleep'	89	206	171	104
`ReflVP : VPSlash → VP`	'see myself'	1034	890	343	1082

Table 5.6: Test cases for some individual functions in the resource grammar

languages come from different language families, and cover a wide range of grammatical complexity. Dutch and Spanish are both Indo-European languages, with fairly simple nominal morphology. Dutch features word order changes in subordinate and question clauses, and separable prefixes in verb phrases (imagine English behaving "look something *up* ~ *up*looked"). Spanish has a large number of tenses and moods, and clitics for direct and indirect objects, which attach to verbs. To add some linguistic variety, we include Estonian from the Finno-Ugric language family, and Basque, an isolate language. In contrast to Dutch and Spanish, Estonian and Basque have a rich nominal morphology, with 14 grammatical cases in each. In addition, Basque has the most complex verb morphology out of all the 4 languages, featuring agreement in subject, object and indirect object. The Dutch, Spanish and Estonian grammars behaved similarly to each other and Basque significantly worse, both in execution time and examples generated.

Table 5.5 shows the number of generated trees for in total for all syntactic functions in the three grammars, and Table 5.6 shows some example functions from the resource grammar. As stated earlier, we do not consider generating test cases for all functions an optimal way of testing a whole resource grammar from scratch; this gives merely a baseline reduction from all possible trees up to a reasonable depth. We introduce the grammars and comment on the results in the following sections.

5.5.1 Grammars

The first grammar is the toy example introduced earlier in this article: `NounPhrases` with 8syntactic functions and 17words in the lexicon. We wrote the concrete syntaxes from scratch for each of the languages, instead of using the full resource grammar and reducing it to only noun phrases. All four concrete syntaxes were completed in less than an hour, by an experienced grammarian with some knowledge in all the languages.

The second grammar is a mid-size application grammar: Phrasebook (Ranta et al., 2012), with 42 categories such as `Person`, `Currency`, `Price` and `Nationality`, 160-word lexicon and

130 functions with arguments. As opposed to the trees that we have seen so far, which only contain syntactic information, the trees in the Phrasebook are much more semantic: for example, the abstract tree for the sentence "how far is the bar?" in the Phrasebook is `PQuestion (HowFar (ThePlace Bar))`, in contrast to the resource grammar tree `UttQS (UseQCl (TTAnt TPres ASimul) PPos (QuestIComp (CompIAdv (AdvIAdv how_IAdv (PositAdvAdj far_A))) (DetCN (DetQuant DefArt NumSg) (UseN bar_N))))` for the same sentence. Limiting up to depth 3, the Phrasebook grammar produces over 480,000 trees[3].

The third grammar is a restricted version of the GF resource grammar, with 84 categories, 217 syntactic functions and 446 words in the lexicon. Since all the languages did not have a complete implementation, we simply took the subset of functions that was common, and removed manually a couple of rare constructions and words that are potentially distracting. This fragment produces hundreds of billions of trees already up to depth 5. None of the resource grammars has been tested systematically before—for the Estonian grammar (Listenmaa and Kaljurand, 2014), the morphological paradigms were tested extensively against existing resources, but syntactic functions were only tested with a treebank of 425 trees.

5.5.2 Execution time

We ran all the experiments on a MacBook Air with 1,7 GHz processor and 8 GB RAM. For Phrasebook, it took just seconds to generate the test suite for all languages. For the resource grammar, Dutch and Estonian finished in 3–4 minutes. However, the Basque resource grammar is noticeably more complex, and creating test trees for all functions took almost three hours.

5.5.3 Generated trees

We report both total and unique trees: total trees are simply the sum of all trees for all functions, and unique trees is the count after removing duplicates, as explained in Section 5.4.5 ("the pizza gives the pizza the pizza" style).

Grammar engineering vs. language typology As we can see in Table 5.5, the number of trees differs between languages. Table 5.6 shows some expected patterns in individual functions. For instance, Basque and Estonian have more complex noun morphology than Dutch, so `RelNP` needs more tests (20+ vs. 2): if the type for `NP` is a large inflection table, it needs to be

[3]Application grammars are usually much more compact than resource grammars, hence depth 3 covers already a lot of relevant trees.

put in many contexts. However, if the question was purely about language complexity, we would expect Spanish to behave more like Dutch: both have only 2 cases, nominative and accusative.

In fact, around 80 % of the code in the Spanish grammar is inherited from a common module to all Romance languages; thus the implementation is meant to be as general as possible, and support all distinctions that appear in some language, not necessarily in Spanish itself. In this case, the pan-Romance category for `NP` includes nominative, accusative, dative and genitive as cases in the inflection table, even though the latter two are realised as just prepositions (*a* and `de`) in Spanish. In addition, all cases include a stressed and unstressed form—this makes for 8 fields in total.

As another example, Basque, Spanish and Dutch have more complex verb phrases (for different reasons!) than Estonian, so they need more test cases to test `ReflVP`. This time it is not only about the contexts, but also the arguments: `ReflVP` takes a `VPSlash`, which has many parameters, so the tool needs to create many examples to cover all different concrete categories for `VPSlash`.

In general, we believe the number of test cases has both language typological and grammar engineering reasons. Other resource grammarians have reported significant differences in complexity between implementations: Enache et al. (2010) report a 200-time reduction in the number of concrete rules after changing the implementation of clitics in verb phrases.

To further explore the effect of grammar engineering vs. inherent language complexity, it would be interesting to generate test cases for two different implementations of the same language. However, there is not much material to explore—writing a full resource grammar is several months' effort, thus writing a second implementation for a language that already is in the RGL is hardly a justifiable use of resources. We could explore related languages, but in practice they are often based on one another, either by copying an existing one and modifying it (e.g. Afrikaans based on Dutch, Estonian on Finnish), or writing a parameterised module from the beginning, with the intention of fitting several languages (e.g. Romance and Scandinavian languages).

Two different implementations of Basque noun phrase grammar Given the lack of large-scale language resources with different implementations, we experimented with the noun phrase grammar for Basque. In the first implementation, the one described earlier in this chapter, we implemented nominal morphology using inflection tables, and syntactic functions choose the correct case for each purpose. In another grammar, we used the Basque resource grammar, which has implemented nouns as stems, and syntactic functions concatenate suffixes to the stems. In the stem-based grammar, nouns have phonological features as inherent parameters, in order to attach the correct form of the inflectional morphemes. For test case generation, this had the added benefit or curse of implicitly testing morphology as well. For instance, `AdjCN` needs to generate test cases with the arguments $\{\overset{\texttt{good}}{\texttt{small}}\}$ `house`, just because *good* ends in a consonant and *small* in a vowel.

Function	**Inflection table**	**Stems**
`PredAdj`, `PredAdv`, `UttNP`, `PrepNP`	1, 1, 1, 1	1, 1, 1, 1
`DetNP`	3	4
`DetCN`	3	4
`AdjCN`	7	4
`AdvCN`	7	4
Total:	**24**	**20**

Table 5.7: Two versions of Basque concrete syntax for noun phrase grammar

In the version based on inflection tables, `AdjCN` needs only one adjective and one noun to form the test cases, for instance `AdjCN small house`. This tree is then put in 7 different contexts, in order to showcase all the different fields in the `CN`.

```
UttNP (DetCN a _)
UttNP (DetCN your _)
UttNP (DetCN the _)
PredAdv <any NP> (PrepNP on (DetCN your _))
PredAdv <any NP> (PrepNP from (DetCN your _))
PredAdv <any NP> (PrepNP on (DetCN the _))
PredAdv <any NP> (PrepNP from (DetCN the _))
```

In Section 5.2.1, we described the placement of determiners in Basque. In the hand-written fragment for noun phrases, we introduced a parameter in the `Det` type, which controls the placement of the determiner, and which form it chooses from the `NP`. We can see clearly which part comes from where: `AdjCN` only creates an inflection table, and the rest of the context chooses different forms.

In contrast, the stem-based grammar creates much more unwieldy arguments for `AdjCN`. We explained the choice of good and small already, due to phonetic features as inherent parameters, but the full trees are more complex, as in the following:

`AdjCN` $\{{}^{\texttt{good}}_{\texttt{small}}\}$ `(AdvCN (PrepNP without (DetCN a house)) house)`
'good / small house without a house'.

This is because in the full resource grammar, there is no parameter for the modifier and determiner placement, instead, each `CN` just includes fields for heavy and non-heavy modifier, and all determiners include the fields for pre- and postdeterminer. This way the functions don't need to include any pattern matching, just concatenate `det.pre ++ cn.heavyMod ++ cn.noun ++ cn.lightMod ++ det.post`, and any of these fields may be empty. The downside is that, in order to generate a minimal number of test cases, the best strategy is to generate a test case where the fewest possible of these strings are empty. This leads into fewer examples, but they are harder to read.

The number of generated test cases doesn't differ much in this example. Furthermore, the version implemented with resource grammar includes many unnecessary distinctions for such a small grammar: for instance, each of the two trees generated for `AdjCN` are put in singular and plural contexts, which is completely irrelevant in the scope of our tiny grammar. Thus we cannot draw any conclusions which strategy leads to more test cases. However, just reading the generated 44 (20+24) trees suggests that the full resource grammar generates less readable test cases.

5.5.4 Qualitative analysis

The Basque resource grammar is still work in progress, and the test sentences showed serious problems in morphology. We thought it premature to get a fluent speaker to evaluate the grammar, because the errors in morphology would probably make it difficult to assess syntax separately. Phrasebook was implemented using the resource grammar, so it was equally error-ridden.

For Estonian, we read through all the Phrasebook test sentences. These 505 sentences showed a handful of bugs—all coming from Phrasebook itself, not from the resource grammar. Most were individual words having the wrong inflection paradigm (the right one exists in the resource grammar, but a wrong one was chosen by the application grammarian), but there were also some bugs in more general functions: for example, using a wrong form of nationality when applied to a human and when to an institution, along the lines of "Spaniard restaurant". As expected, Phrasebook sentences were easier to read, and made more sense semantically than sentences from the resource grammar.

We have been developing the tool by testing it on the Dutch resource grammar—this process is described in more detail in Section 5.6. During 6 months, we have committed 22 bugfixes on Dutch in the GF main repository. (In the name of honesty, a few of the bugs were caused by our earlier "fixes"—that was before we had implemented the comparison against an older version of the grammar.) One of the bugs found in Dutch was also present in other languages, so we fixed it in German and English.

5.6 Case study: Fixing the Dutch grammar

In this section, we describe the long-term project of fixing bugs in the Dutch resource grammar. The content of the section describes general principles, and is well understandable for readers who don't speak Dutch themselves.

5.6.1 Experimental setup

We describe a collaboration between a grammarian (the author), who is an expert in GF, and a native Dutch speaker. We wanted to investigate the feasibility of this division of labour: ideally, the tester can be any native speaker, with no skills in GF whatsoever. In our case, the tester also had GF skills, but did not have access to the grammar, only to the produced sentences.

We generated a set of test sentences for each function in Dutch, with English translations, and gave them to the native tester to read. The tester replied with a list of sentences that were wrong, along with suggestions for improvement. Communication between the grammarian and the tester was conducted via email.

5.6.2 Types of bugs

We can classify the bugs in two dimensions: how easy it is to understand what the problem is, and how easy it is to fix the grammar. Ease of understanding is relative to the grammarian: a trained linguist who is fluent in Dutch would have easy time pinpointing the error from the generated test cases, having both intuition and technical names for things. Ease of fixing is relative to the grammar: a given grammatical phenomenon can be implemented in a variety of ways, some of which are harder to understand.

In more concrete terms, *easy to fix* means just some local changes in a single function. In contrast, bugs that are *hard to fix* usually involve modifying several functions, restructuring the code or adding new parameters.

Easy to understand, easy to fix Perhaps the easiest bug to fix is to correct a wrong lexical choice. Below is an example feedback from the tester.

> “opschakelen” is not the right translation of “switch on”. “aanzetten” or “aandoen” is better.

Other examples include wrong inflection or agreement, e.g. the polite second person pronoun should take the third person singular verb form, but was mistakenly taking second person forms.

Typically, bugs that are due to an almost complete implementation are easy to fix. For instance, particle verbs were missing the particle in future tense. Looking at the generated sentences, we could see the particle being in the right place in all other tenses, except for the future. There was a single function that constructed all the tenses, and looking at the source code, we could see the line `++ verb.particle` in all other tenses except the future. In such a case, fixing the bug is fairly trivial.

Easy to understand, hard to fix Dutch negation uses two strategies: the clausal negation particle *niet* ‘not’, and the noun phrase negation *geen* ‘no’. There are some subtleties in their usage—the following quote comes from the tester:

> In any case, one can never say “eet niet wormen” (don’t eat worms, literally). That should always be “eet geen wormen” (don’t eat worms, correctly translated)

We sent three more sentences as follow-up, and got the following answer:

> eet niet deze wormen - maybe OK?, feels strange
> eet deze wormen niet - definitely OK
> eet niet 5 wormen - definitely OK

From the feedback, it was fairly easy to see the pattern: clauses with indefinite noun phrases (*a* worm, *worms*) use noun phrase negation, but if the noun phrase has any other determiner (*these* worms, *five* worms), then clausal negation is appropriate.

In the grammar, this fix required changes to 13 categories. Not all categories had to be changed manually, but e.g. a change in `NP` changes all categories that depend on it, such as `Comp` and `VP`. Depending on how modularly the grammar is implemented, this means that some functions that operate on `VP` or `Comp` need to be changed too, when `NP` changes.

Hard to understand, easy to fix The following two sentences were generated by the same function, which turns superlative adjectives and ordinal numbers into complements. The tester reported problems with both of them, as follows:

> ik wil roodst worden –> ik wil **het** roodst worden ('I want to become reddest')
> ik wil ~~tiend~~ worden –> ik wil **tiende** worden ('I want to become tenth')

We gave some more sentences to the tester, and got the following feedback:

> ik wil linker worden –> ik wil **de** linker worden ('I want to become left')
> ik wil 224e worden = OK ('I want to become 224th')

This small example gave at least three different ways of using these complements: for numerals, no article and -e at the end (*tiende* 'tenth'); for superlative adjectives, the article *het* and no -e at the end of the adjective (*het roodst* 'the reddest'), and for a class of adjectives like *left* and *right*, the article *de* (*de linker* 'the left one').

In addition, the grammar has a separate construction for combining a numeral and a superlative adjective, e.g. "tenth best". Since the tests were generated per function, the main tester didn't read those sentences at the same time. After noticing the additional function, we asked another informant how to say *Nth best*, and got an alternative construction *op (N-1) na best*. Eventually, we got an answer that the strategy used for superlative adjectives, i.e. with the article *het* and no -e in the number, is acceptable.

Once it was clear to the grammarian how to proceed, fixing the bug was easy. There was already a parameter for the adjective form: attributive in two forms (strong and weak) and one predicative, and the different classes of adjectives corresponded to the abstract syntax of the GF Resource Grammar Library (RGL). Thus it was easy to modify the predicative form in a different way for different adjective types. Earlier, the predicative was just identical to the other attributive form, but now the `AP` type actually contains 3 different strings for superlatives: *beste* and *best* for attributive and *het best* for predicative. Adjectives in positive or comparative don't get the article: *good* is just *goede*, *goed* and *goed* (not **het goed*).

If there hadn't been already a parameter for different adjective forms, or if the classes of words with different behaviours hadn't corresponded to the RGL categories, then this bug would have required more work to fix.

Hard to understand, hard to fix As an example of a problem that was hard to understand and hard to fix, we take the agreement of a reflexive construction in conjunction with a verbal complement. The example works as well for English, so for the convenience of the reader, we use English as an example language. Now, consider the following sentences:

- I like myself / you like yourself / …
- I help [you like yourself / them like themselves / …]

The choices of reflexive pronoun seem reasonable: if the object of liking was *I* in the second example, the pronoun wouldn't be *myself* but *me*: "I help you like me". In the GF grammar, these sentences are constructed in a series of steps:

```
PredVP (UsePron i_Pron)
       (ComplSlash
           (SlashV2V help_V2V
               (ReflVP
                 (SlashV2a like_V2)
               )
           )
           (UsePron they_Pron)
       )
```

The innermost subtree is `SlashV2a like_V2`: the transitive verb *like* is converted into a `VPSlash` (i.e. `VP\NP`). Right after, the function `ReflVP` fills the `NP` slot and creates a `VP`. However, no concrete string for the object is yet chosen, because the reflexive object depends on the subject. The status of the VP is as follows at the stage `ReflVP (SlashV2a like_V2)`:

```
    s = "like" ;
ncomp = table { I => "myself" ; You => "yourself" ; … } ;
vcomp = [] ;
```

If we added a subject at that point, the subject would choose the appropriate agreement: *I* like *myself*, *you* like *yourself*. But instead, we add another slash-making construction, `SlashV2V help_V2V`. Now the new verb `help_V2V`, which takes both a direct object and a verbal complement, becomes the main verb. The old verb *like* becomes a verbal complement.

```
    s = "help" ;
ncomp = table { I => "myself" ; You => "yourself"; … } ;
vcomp = "like" ;
```

The next stage is to add an NP complement they_Pron, using the function ComplSlash. The standard way for ComplSlash is to insert its NP argument into the ncomp table, taking the vcomp field along.

In the old buggy version, ComplSlash just concatenated the new object and the vcomp with the reflexive that was already in the ncomp table. But the scope of the reflexive was wrong: when adding an object to a VPSlash that has a verbal complement clause, the object should complete the verbal complement and pick the agreement. It is not in the scope for the subject. The old behaviour was as follows:

```
    s = "help" ;
ncomp = table { I => "them like myself" ; You => "them like yourself"; … } ;
vcomp = [] ;
```

After fixing the bug, the table was as follows:

```
    s = "help" ;
ncomp = table { _ => "them like themselves" } ;
vcomp = [] ;
```

But this turned out not to be a perfect solution. The exception to this is when the VPSlash is formed by VPSlashPrep : VP -> Prep -> VPSlash. With the changes to ComplSlash, we suddenly got sentences such as "[I like *ourselves*] without *us*". This would be a valid linearisation for a tree where [*ourselves* without *us*] is a constituent (such a tree is formed by another set of functions and was linearised correctly), but in this case, the order of the constructors is as follows:

- ReflVP like

  ```
      s = "like" ;
  ncomp = table { I => "myself" ; You => "yourself" ; … } ;
  ```

- VPSlashPrep (ReflVP like) without

  ```
      s = "like" ;
  ncomp = table { I => "myself" ; You => "yourself" ; … } ;
   prep = "without"
  ```

- ComplSlash (VPSlashPrep (ReflVP like) without) we_Pron)

```
      s = "like" ;
  ncomp = table { I => "myself" ; You => "yourself" ; … } ;
    adv = "without us"
```

The desired behaviour is to put the complement into an adverbial slot and keeping the agreement in `ncomp` open to wait for the subject. But the following happened after our initial changes in `ComplSlash`:

```
      s = "like" ;
  ncomp = table { _ => "ourselves without us" } ;
```

To fix this problem, we added another parameter to the category `VPSlash`. All `VPSlashes` constructed by `VPSlashPrep` have now a `missingAdv` set True: this tells that the `VPSlash` is not missing a core argument, so it shouldn't affect the agreement. With the new parameter, `ComplSlash` can now distinguish when to choose the agreement from the `NP` argument and when to leave it open for the subject.

All in all, this was a complex phenomenon, and there were different interpretations in the RGL. For the Scandinavian languages, the behaviour was as we expected, but for English and German, it was similar to Dutch. We fixed the grammar for all three languages (Dutch, English and German), using the same strategy.

5.6.3 Results

How many bugs were fixed and how many were of which kind? We skip the ease of understanding, and just classify the bugs by how easy they were to fix.

Easy to fix

1. Several lexical changes.
2. Several inflection fixes.
3. `youPol_Pron` had agreement of `Sg P2`, changed it to `Sg P3` so that a correct reflexive pronoun is chosen.
4. Choose always stressed forms of personal pronouns.
5. Change agreement in conjunctions
6. Extra prefix in prefix verbs for perfect tense
7. Missing participle in future tense
8. Plural imperatives

9. Two bugs in postmodifier `AP`s: placement and the adjective form. *een getrouwde worm* 'a married worm' is correct, but a heavier `AP`, such as *getrouwd met mij* 'married with me' should become a postmodifier (*een worm getrouwd met mij*), and in that case, the adjective form should be without the e at the end.
10. Superlatives and ordinals
11. `DetQuant` (and `DetQuantOrd`) combining a `Quant` and a `Num`, and when `Num` is an actual digit, both `Quant` and the `Num` contribute with a string, thus becoming *een 1 huis* 'a 1 house'.

Hard to fix

1. Add missing inflected forms for past participles + add missing linearisation for the function `PastPartAP`
2. Preposition contraction (*met dit huis* 'with this house' and *hiermee* 'with this')
3. Negation patterns (*niet* and *geen*)
4. Variety of word order weirdness in verbal complements: affected several functions, fixed in several functions
5. Scope of `ReflVP` with `VPSlash`

Effort Tester 1 has read around 5,000 sentences over the course of several weeks, usually taking just minutes of time at one go. Tester 2 has been used as a backup when Tester 1 was not available and the grammarian wanted quick feedback. Where Tester 1 was instructed to read through a long list of sentences (out of which most are correct), Tester 2 has been given sentences that Tester 1 already flagged as wrong. All in all, Tester 2 has read around 50 sentences.

We found that communication via email was suboptimal, and would be hard to scale up to more testers. In the future, we aim to improve the process management with a more efficient workbench.

5.7 Conclusion and future work

We have presented method for automatically generating minimal and exhaustive sets of test cases for testing grammars. We have found the tool useful in large-scale grammar writing, in a context where grammars need to be *reliable*.

One problem we have encountered is that the test sentences from resource grammars are often nonsensical semantically, and hence a native speaker might intuitively say that a sentence is wrong, even though it is just unnatural. For instance, the function `SelfAdVVP` covers constructions such as "the president herself is at home". However, the function itself is completely general and can take any verb phrase and add the reflexive pronoun, and furthermore it may add any subject NP, or none at all, for context. This means that bizarre combinations often appear in the generated test cases, such as "to always itself be hungry"–there is no meaningful subject present, but the VP requires a reflexive pronoun anyway.

So far the only mode of operation is generating test cases for a single function. As future work, we are planning to add a separate mode for testing the whole grammar from scratch: intentionally create trees that test several functions at once. We have an implementation only for GF grammars so far, but the general method works for any grammar formalism that can be compiled into PMCFG. GF already supports reading context-free grammars, so testing any existing CFG is a matter of some preprocessing.

Chapter 6

Conclusions

This chapter concludes the thesis. Firstly, we summarise the main results of this thesis. For the remainder of this chapter, we discuss insights gained from this work, and possible directions for future research.

6.1 Summary of the thesis

We set out to test and verify natural language grammars. In this thesis, we have described two systems for two different grammar formalisms: Constraint Grammar (CG) and Grammatical Framework (GF).

For CG, previous evaluation methods relied on a gold standard corpus, or manual inspection if there was no corpus available. Our work started with the observation that CG rules resemble logical formulas; REMOVE and IF can be expressed as negations and implications respectively. As noted by Lager (1998), a straight-forward implementation in a logical framework results in a parallel and unordered CG.

Our implementation of the sequential CG is very similar to the encoding by Lager and Nivre (2001). However, we modified the setup in a crucial detail: instead of an actual sentence, we applied the rules to a *symbolic sentence*, where every word contains every possible reading. Then, instead of the original question "Which readings are true after all rule applications?", we found a meaningful SAT-problem, with the question "Which readings were originally true?"

We exploited this newly found property for grammar analysis. Our method tests the internal consistency of a grammar, only relying on a morphological lexicon. The system explores all possibilities to see if a particular rule can apply: if there is no input that would trigger a rule r after rule r', we know that rules r and r' are in conflict. This method was successful in finding such conflicts, when tested with grammars between 60–1200 rules. In addition, the mechanisms that we developed in order to test CG have interesting side effects: namely, they let us model CG as a generative formalism, which opens potential for deriving CG grammars from other grammar formalisms.

When testing GF grammars, earlier we relied on an informant to remember and explain all important phenomena to test, and a grammarian to form all relevant trees that showcase these phenomena. This time, our inspiration was all the work done for automatic test case generation for general software, such as Celentano et al. (1980); Geist et al. (1996); Claessen and Hughes (2011). Software testing deals with notions of coverage and compactness, and deriving test cases from a formal specification; this proved to be a useful approach for grammar testing as well.

In this thesis, we have presented a tool that automates the two steps: identifying the grammatical phenomena to test and formulating them as trees. Now, the grammarian only needs to input a function or category, and the system outputs a minimal and representative set of sentences that showcases all the variation in the given function or category. The informant only needs to read and evaluate the example sentences. (Of course, humans are still needed to write the grammars in the first place, but we consider that a fun task!)

6.2 Insights and future directions

6.2.1 On completeness

Testing can only show the presence of bugs, not the absence. The tools introduced in this thesis are no exception: here we comment on the reliability and completeness of both systems.

CG The quality of a CG grammar can be judged according to purely internal criteria: either the rules are internally consistent or not. It is a separate question (not the research question in this thesis!) whether the CG grammar also does a good job at disambiguating natural language. Such questions are better left for corpus-based methods.

To be more precise, a CG grammar is internally consistent given a certain tagset, rules to combine tags into readings, and ambiguity classes. This means that reliability of our CG analysis tool depends on the consistency of the CG grammar and the morphological lexicon where the tags and the readings come from. If the morphological lexicon contains a word with the wrong tag, or the specifications for how tags can combine into readings are outdated, our tool may do the wrong thing: report a false conflict or fail to detect a real error.

But if the lexicon and tag set are correct, then we can be fairly certain of the analysis. If the grammar is deemed conflict-free, it means that for each rule r in the grammar, the program has found a sequence of words which can pass through all previous rules, and r can remove something from at least one word. In other words, the system has to *generate* a positive example for all rules r it judges conflict-free. This r doesn't even need to do the right thing—it may as well be "*wish* is a verb in the context *the wish*"—but we can still trust that r doesn't conflict with any of the previous rules in the grammar.

Can we trust the program when it finds r being in conflict with some other rule? We use symbolic evaluation to explore all possibilities: if there was a way in which r could apply (e.g. create a context which prevents a previous rule from matching), the SAT-solver would have found it. If we find r impossible to apply, it has to be one of the following causes: (a) its conditions are inconsistent (e.g. "-1 has to be a noun and not be a noun")[1], or (b) another rule r' or set of rules R' before it forces the symbolic sentence to be in a certain way. In the case of (a), the human grammarian would hopefully see the error when it is pointed out. In the case of (b), even if the conflict is not clear by just looking at the grammar, the SAT-solver can help by generating a positive example, i.e. a sequence that triggers r when it goes through all the previous rules except r'.

GF On the GF side, we are trying to answer the harder question: does the grammar do an adequate job at producing some natural language? For that, we need a set of test sentences that cover all phenomena in the grammar. In the CG world, we had a natural restriction: there is only a few hundreds or thousands of different tags in the morphological lexicon, and there are restrictions in how they mix together into readings. So the language of the tags and readings is finite—sounds reasonable that we can do the job thoroughly.

But natural language is infinite. How does that work with a finite set of test cases? First of all, we are testing a PMCFG representation of natural language. This representation may still define an infinite language: for example, the grammar can parse and produce a sentence with arbitrarily many subordinate clauses (*I know that you think that she disagrees that …this example is contrived*), only limited by physical constraints such as memory or having a source of power. But importantly, the grammar still has a finite number of *functions*, and all those functions can only take arguments of finitely many different types.

[1] If we have ambiguity classes in place, the inconsistency can also be more subtle, e.g. "-1 has to be noun and punctuation", assuming that there is no word form in the lexicon that has such an ambiguity.

How about repeating the same function arbitrarily many times? Could it be that the tree for *you think that X* is linearised correctly, but *I know that you think that X* introduces an error? We can answer this definitely, and the answer is no. In the PMCFG format, all variation is encoded as parameters: if the n^{th} repetition of some function would cause different behaviour, then there would be a parameter that counts the applications, and the program would generate separate test cases for one subordinate clause, and n nested subordinate clauses.

But we can still fail to generate a test that reveals the error—the failure mode is, simply, not enough parameters in the grammar. Consider the `AdjCN` function for Spanish again, and say that we don't have the parameter for pre- and postmodifier at all, and `AdjCN` only comes in two variants, one for feminine and one for masculine. Suppose that all adjectives are, incorrectly, postmodifiers. Then we would notice the error only if the single `AP` argument for $\texttt{AdjCN}_{\texttt{fem}}$ and $\texttt{AdjCN}_{\texttt{masc}}$ happened to be a premodifier one.

If the missing parameter handles a common phenomenon, it is more likely that we catch the error even with missing parameters. This was the case with, for example, Dutch preposition contraction (from Section 5.2.1). Originally, none of the prepositions contracted, but this is the most common behaviour—only a handful of prepositions don't contract. Thus we got some test cases which showed the error, and went on to fix the grammar. After the fixes, the system created separate test cases for contracting and non-contracting prepositions.

6.2.2 Future directions

The tools are available, and have had already some usage outside the research group. Out of the two tools, the GF test suite generation shows more potential for wider adoption and becoming a standard practice of GF grammar engineering.

There are many directions to develop the GF test suite tool further. As we mention in Section 5.7, there is no semantic coherence in the generated sentences—this is particularly important when working with non-linguist testers. However, if it turns out that most testing work is done on application grammars, this step may not be as crucial as for testing resource grammars. Furthermore, resource grammarians are often linguists themselves, and thus more capable of distinguishing between semantic and syntactic weirdness.

Another interesting feature is shrinking the test sentences. Especially for resource grammars, the generated examples are quite long: this is due to the preference for maximally unique fields, and "always run with Mary" is a more distinguishing example of a verb phrase than "run". However, if an error is found in a sentence with "always run with Mary", it would be desirable to generate another example with smaller examples, such as "run", "always run" and "run with Mary", to help narrow down which function introduces the error.

As we mention in Section 5.6, communication with the testers via email is not a scalable solution. We would like to use the tool as a part of a more general language workbench, where the tester can read test sentences along with a linearisation in another language they know. The tester can then comment on the sentences, and give a correct linearisation in cases where the grammar outputs an incorrect sentence. Then when the grammar is updated, the grammarian could see immediately whether the produced sentences are the same as the tester specified or not.

In more philosophical terms, we would like the grammar writing community to view testing, in terms of Beizer (2003), as a "mental discipline that helps all IT professionals develop higher-quality software." Our work is by no means the first approach to grammar testing: for instance, Butt et al. (1999) recommend frequent use of test suites, as well as extensive documentation of grammars. We argue that generating test suites automatically from the grammar itself, combined with treebanks for coverage testing, is an even better way to test grammars. It is always a risk that a test suite is deprecated when the grammar is changed; with automatic generation, the test suite is updated along with the grammar.

Bibliography

Itziar Aduriz, José María Arriola, Xabier Artola, Arantza Diaz de Ilarraza, Koldo Gojenola, and Montse Maritxalar. Morphosyntactic disambiguation for basque based on the constraint grammar formalism. In *Proceedings of Recent Advances in NLP (RANLP97)*, 1997.

Itziar Aduriz, Maria Jesús Aranzabe, Jose Maria Arriola, Aitziber Atutxa, Arantza Diaz de Ilarraza, Nerea Ezeiza, Koldo Gojenola, Maite Oronoz, Aitor Soroa, and Ruben Urizar. Methodology and steps towards the construction of EPEC, a corpus of written Basque tagged at morphological and syntactic levels for the automatic processing. In *Corpus Linguistics Around the World*, volume 56 of *Language and Computers*, pages 1–15. Rodopi, Netherlands, 2006.

Hiyan Alshawi. *The Core Language Engine*. MIT Press, Cambridge, Ma, 1992.

Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

Krasimir Angelov. *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University of Technology, 2011.

Boris Beizer. *Software Testing Techniques*. Dreamtech, 2003. ISBN 9788177222609. URL https://books.google.se/books?id=Ixf97h356zcC.

Eckhard Bick. *The parsing system "Palavras". Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework*. PhD thesis, University of Århus, 2000.

Eckhard Bick. A CG & PSG hybrid approach to automatic corpus annotation. In *Proceedings of the Shallow Processing of Large Corpora Workshop (SProLaC 2003)*, pages 1–12, 2003.

Eckhard Bick. A Constraint Grammar Parser for Spanish. In *Proceedings of TIL 2006 – 4th Workshop on Information and Human Language Technology*, 2006. URL http://visl.sdu.dk/~eckhard/pdf/TIL2006.pdf.

Eckhard Bick. Constraint grammar applications. In *Constraint Grammar workshop at the 18th Nordic Conference of Computational Linguistics (NODALIDA 2011)*, page iv, 2011.

Eckhard Bick. ML-Tuned Constraint Grammars. In *Proceedings of the 27th Pacific Asia Conference on Language, Information and Computation (PACLIC 2013)*, pages 440–449, 2013. URL http://visl.sdu.dk/~eckhard/pdf/PACLIC2013_grammar_tuning2.pdf.

Eckhard Bick and Tino Didriksen. CG-3 – Beyond Classical Constraint Grammar. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, 2015. URL http://www.ep.liu.se/ecp/109/007/ecp15109007.pdf.

Eckhard Bick, Kristin Hagen, and Anders Nøklestad. Optimizing the Oslo-Bergen Tagger. In *Constraint Grammar workshop at the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*. Linköping University Electronic Press, 2015.

Armin Biere. *Handbook of Satisfiability*. Frontiers in artificial intelligence and applications. IOS Press, 2009. ISBN 9781586039295.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference (TACAS'99)*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49059-3.

Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In Ranjit Jhala and David Schmidt, editors, *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 70–87, Berlin, Heidelberg, January 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18275-4.

Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, 1995.

Miriam Butt, Tracy Holloway King, María-Eugenia Niño, and Frédérique Segond. *A Grammar Writer's Cookbook*. CSLI Publications Stanford, 1999.

Augusto Celentano, S Crespi Reghizzi, P Della Vigna, Carlo Ghezzi, G Granata, and Florencia Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10 (11):897–918, 1980.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 46(4):53–64, May 2011. ISSN 0362-1340. doi: 10.1145/1988042.1988046. URL http://doi.acm.org/10.1145/1988042.1988046.

Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson. SAT-Solving in Practice, with a Tutorial Example from Supervisory Control. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.

Koen Claessen, Jasmin Fisher, Samin Ishtiaq, Nir Piterman, and Qinsi Wang. Model-Checking Signal Transduction Networks through Decreasing Reachability Sets. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification, (CAV 2013)*, pages 85–100, July 2013.

Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual Symposium on Theory of Computing*, pages 151—158, 1971.

Haskell B. Curry. Some logical aspects of grammatical structure. *Structure of language and its mathematical aspects*, 12:56–68, 1961.

Doug Cutting, Julian Kupiec, Jan Pedersen, and Penelope Sibun. A practical part-of-speech tagger. In *Proceedings of 3rd Conference on Applied Natural Language Processing*, pages 133–140, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. doi: 10.3115/974499.974523. URL http://dx.doi.org/10.3115/974499.974523.

Philippe de Groote. Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter*, pages 148–155, Toulouse, France, 2001.

Tino Didriksen. *Constraint Grammar Manual*. Institute of Language and Communication, University of Southern Denmark, 2014. URL http://beta.visl.sdu.dk/cg3.html.

Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364515. URL http://doi.acm.org/10.1145/2364506.2364515.

Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004.

Martin Eineborg and Nikolaj Lindberg. Induction of constraint grammar-rules using progol. In David Page, editor, *Inductive Logic Programming*, volume 1446 of *Lecture Notes in Computer Science*, pages 116–124. Springer Berlin Heidelberg, 1998.

Ramona Enache, Aarne Ranta, and Krasimir Angelov. An open-source computational grammar for Romanian. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 163–174. Springer, 2010.

Nerea Ezeiza, Itziar Aduriz, Iñaki Alegria, Jose Mari Arriola, and Ruben Urizar. Combining stochastic and rule-based methods for disambiguation in agglutinative languages. In *COLING-ACL'98. Pgs. 380 - 384. Vol 1. Montreal (Canada). August 10-14, 1998*, 1998.

Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

Mikel Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine translation*, 25(2):127–144, 2011.

Claire Gardent and Shashi Narayan. Error mining on dependency trees. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers - Volume 1*, ACL '12, pages 592–600, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=2390524.2390607.

Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtenstein, Shmuel Ur, and Yaron Wolfsthal. Coverage-directed test generation using symbolic techniques. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 143–158, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-49567-3.

Jorge Graña, Gloria Andrade, and Jesús Vilares. Compilation of constraint-based contextual rules for part-of-speech tagging into finite state transducers. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 128–137. Springer, 2002.

Barbara B. Greene and Gerald M. Rubin. Automatic Grammatical Tagging of English. Technical report, Department of Linguistics, Brown University, 1971.

Maurice Gross. The construction of local grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-state language processing*, pages 329—354. Cambridge (MA), USA: MIT, 1997.

Jacky Herz and Mori Rimon. Local syntactic constraints. In *Proceedings of the Second International Workshop on Parsing Technologies*, pages 200–209, 1991.

Hindle, D. Acquiring Disambiguation Rules from Text. In *Proceedings of the 27th Annual Meeting of the ACL*, pages 118–125, 1989.

Mans Hulden. Constraint grammar parsing with left and right sequential finite transducers. In *Proceedings of 9th International Workshop on Finite State Methods and Natural Language Processing*, pages 39–47. Association for Computational Linguistics, 2011.

Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975. doi: 10.1016/S0022-0000(75)80019-5. URL https://doi.org/10.1016/S0022-0000(75)80019-5.

Fred Karlsson. Constraint grammar as a framework for parsing running text. In *Proceedings of 13th International Conference on Computational Linguistics (COLING 1990)*, volume 3, pages 168–173, Stroudsburg, PA, USA, 1990. Association for Computational Linguistics. ISBN 952-90-2028-7. doi: 10.3115/991146.991176. URL http://dx.doi.org/10.3115/991146.991176.

Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. *Constraint Grammar: a language-independent system for parsing unrestricted text*, volume 4. Walter de Gruyter, 1995.

Janna Khegai. *Language engineering in Grammatical Framework (GF)*. PhD thesis, Chalmers University of Technology, 2006.

Paul Kiparsky. Linguistic universals and linguistic change. In Emmon Bach and R. Harms, editors, *Universals in Linguistic Theory*, pages 170–202. Holt, Rinehart, and Winston, 1968.

Wen Kokke and Inari Listenmaa. Exploring the expressivity of constraint grammar. In *Proceedings of the NoDaLiDa 2017 Workshop on Constraint Grammar-Methods, Tools and Applications, 22 May 2017, Gothenburg, Sweden*, pages 15–22. Linköping University Electronic Press, 2017.

Kimmo Koskenniemi. Finite-state parsing and disambiguation. In *Proceedings of 13th International Conference on Computational Linguistics (COLING 1990)*, volume 2, pages 229–232, Stroudsburg, PA, USA, 1990. Association for Computational Linguistics.

Kimmo Koskenniemi. Representations and finite-state components in natural language. *Finite-state language processing*, pages 99–116, 1997.

Torbjörn Lager. Logic for Part of Speech Tagging and Shallow Parsing. In *Proceedings of the 11th Nordic Conference on Computational Linguistics (NODALIDA 1998)*, 1998.

Torbjörn Lager. Transformation-based learning of rules for constraint grammar tagging. In *Proceedings of the 13th Nordic Conference on Computational Linguistics (NODALIDA 2001)*, 2001.

Torbjörn Lager and Joakim Nivre. Part of speech tagging from a logical point of view. In *Logical Aspects of Computational Linguistics, 4th International Conference (LACL 2001)*, pages 212–227, 2001.

Jean-Louis Lassez, V.L. Nguyen, and Elizabeth Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

Inari Listenmaa and Koen Claessen. Constraint Grammar as a SAT problem. In *Constraint Grammar workshop at the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, 2015.

Inari Listenmaa and Koen Claessen. Analysing Constraint Grammars with a SAT-solver. In *Proceedings of the 10th edition of the Language Resources and Evaluation Conference (LREC 2016)*, 2016.

Inari Listenmaa and Koen Claessen. Automatic test suite generation for PMCFG grammars. In *Proceedings of the Fifth Workshop on Natural Language and Computer Science*, 2018.

Inari Listenmaa and Kaarel Kaljurand. Computational Estonian Grammar in Grammatical Framework. In *9th SALTMIL workshop on free/open-source language resources for the machine translation of less-resourced languages*, 2014.

Inari Listenmaa, Jose Maria Arriola, Itziar Aduriz, and Eckhard Bick. Cleaning up the basque grammar: a work in progress. In *Proceedings of the NoDaLiDa 2017 Workshop on Constraint Grammar-Methods, Tools and Applications, 22 May 2017, Gothenburg, Sweden*, pages 10–14. Linköping University Electronic Press, 2017.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden, 2004.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

João Marques-Silva. Boolean Satisfiability Solving: Past, Present & Future. Presentation given at the Microsoft Research International Workshop on Tractability, Cambridge, UK, July 5–6, 2010. URL http://research.microsoft.com/en-us/events/tractability2010/joao-marques-silvatractability2010.pdf.

Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 595–606, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884797. URL http://doi.acm.org/10.1145/2884781.2884797.

John McCarthy. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28, Munich, West Germany, August 1962. North-Holland.

Richard Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.

Reinhard Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

Dávid Márk Nemeskey, Francis Tyers, and Mans Hulden. Why implementation matters: Evaluation of an open-source constraint grammar parser. In *Proceedings of the 25th International Conference on Computational Linguistics (COLING 2014)*, pages 772–780, Dublin, Ireland, August 2014. URL http://www.aclweb.org/anthology/C14-1073.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan T. McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the 10th edition of the Language Resources and Evaluation Conference (LREC 2016)*, 2016.

Kemal Oflazer and Gökhan Tür. Morphological disambiguation by voting constraints. In *Proceedings of the 8th Conference of the European Chapter of the Association for Computational Linguistics (EACL 1997)*, pages 222–229, 1997.

Lluís Padró. A constraint satisfaction alternative for POS tagging. In *Proceedings of NLP+IA/TAL+AI*, 1996.

Michael Pellauer, Markus Forsberg, and Aarne Ranta. BNF Converter: Multilingual front-end generation from labelled BNF grammars. Technical report, Computing Science at Chalmers University of Technology and Gothenburg University, 2004. URL http://bnfc.digitalgrammars.com.

Janne Peltonen. Rajoitekielioppien toteutuksesta äärellistilaisin menetelmin. Master's thesis, University of Helsinki, 2011.

Jussi Piitulainen. Locally tree-shaped sentence automata and resolution of ambiguity. In *Proceedings of the 10th Nordic Conference of Computational Linguistics*, number 26, pages 50–58, 1995.

Tommi Pirinen. Using weighted finite state morphology with VISL CG-3—Some experiments with free open source Finnish resources. In *Constraint Grammar workshop at the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, 2015.

Carl Pollard. Higher-Order Categorial Grammar. In M. Moortgat, editor, *Proceedings of the Conference on Categorial Grammars (CG2004), Montpellier, France*, pages 340–361, 2004.

Aarne Ranta. Grammatical Framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

Aarne Ranta. The GF Resource Grammar Library. *Linguistics in Language Technology*, 2, 2009. URL http://elanguage.net/journals/index.php/lilt/article/viewFile/214/158.

Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, 2011.

Aarne Ranta, Ramona Enache, and Grégoire Détrez. Controlled Language for Everyday Use: The MOLTO Phrasebook. In *Proceedings of the Second International Conference on Controlled Natural Language*, CNL'10, pages 115–136, Berlin, Heidelberg, 2012. Springer-Verlag.

Aarne Ranta, Krasimir Angelov, Prasanth Kolachina, and Inari Listenmaa. Large-scale hybrid interlingual translation in GF: a project description. In *SLTC'14, 5th Swedish Language Technology Conference*, 2014.

M. Rayner, D. Carter, P. Bouillon, V. Digalakis, and M. Wirén. *The Spoken Language Translator*. Cambridge University Press, Cambridge, 2000.

Robert Reynolds and Francis Tyers. A preliminary constraint grammar for Russian. In *Constraint Grammar workshop at the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*. Linköping University Electronic Press, 2015.

Christer Samuelsson, Pasi Tapanainen, and Atro Voutilainen. Inducing constraint grammars. In Laurent Miclet and Colin de la Higuera, editors, *Grammatical Interference: Learning Syntax from Sentences*, volume 1147 of *Lecture Notes in Computer Science*, pages 146–155. Springer Berlin Heidelberg, 1996. URL http://dx.doi.org/10.1007/BFb0033350.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2):191–229, 1991. doi: 10.1016/0304-3975(91)90374-B. URL https://doi.org/10.1016/0304-3975(91)90374-B.

B. Selman and H. Kautz. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359—363, 1992.

Andrei Sfrent. Machine learning of rules for part of speech tagging. Master's thesis, Imperial College London, United Kingdom, 2014.

Mary Sheeran and Gunnar Stålmarck. *Formal Methods in Computer-Aided Design: Second International Conference, FMCAD' 98 Palo Alto, CA, USA, November 4–6, 1998 Proceedings*, chapter A Tutorial on Stålmarck's Proof Procedure for Propositional Logic, pages 82–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49519-2. doi: 10.1007/3-540-49519-3_7. URL http://dx.doi.org/10.1007/3-540-49519-3_7.

Miikka Silfverberg and Krister Lindén. Conflict resolution using weighted rules in HFST-TWOLC. In *Proceedings of 17th Nordic Conference of Computational Linguistics (NODALIDA 2009)*, pages 174–181, 2009.

Mark Steedman. Combinators and grammars. In *Categorial Grammars and Natural Language Structures*, pages 417–442, 1988.

Pasi Tapanainen. *The Constraint Grammar Parser CG-2*, volume 27 of *Publications of the Department of General Linguistics, University of Helsinki*. Yliopistopaino, Helsinki, 1996.

Pasi Tapanainen. *Parsing in two frameworks: Finite-state and Functional dependency grammar*. PhD thesis, University of Helsinki, 1999.

Pasi Tapanainen and Timo Järvinen. A non-projective dependency parser. In *Proceedings of 5th Conference on Applied Natural Language Processing*, pages 64—71, Stroudsburg, PA, USA, 1997. Association for Computational Linguistics.

Gertjan van Noord. Error Mining for Wide-Coverage Grammar Engineering. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL)*, pages 446–453, Barcelona, Spain, 2004.

Krishnamurti Vijay-Shanker, David Weir, and Aravind Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th annual meeting on Association for Computational Linguistics*, pages 104–111. Association for Computational Linguistics, 1987.

Atro Voutilainen. *Designing a parsing grammar*, volume 22 of *Publications of the Department of General Linguistics, University of Helsinki*. Yliopistopaino, 1994.

Atro Voutilainen. Does tagging help parsing? A case study on finite state parsing. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 1998)*, pages 25–36. Association for Computational Linguistics, 1998.

Atro Voutilainen. Hand crafted rules. In H. van Halteren, editor, *Syntactic Wordclass Tagging*, pages 217–246. Kluwer Academic, 2004.

Atro Voutilainen. FinnTreeBank: Creating a research resource and service for language researchers with Constraint Grammar. In *Proceedings of the Constraint Grammar workshop at the 18th Nordic Conference of Computational Linguistics (NODALIDA 2011)*, pages 41–49, 2011.

Anssi Yli-Jyrä. Structural Correspondence between Finite-State Intersection Grammar and Constraint Satisfaction Problem. In *Finite State Methods in Natural Language Processing 2001 (FSMNLP 2001), ESSLLI Workshop*, pages 1–4, 2001.

Anssi Yli-Jyrä. An efficient constraint grammar parser based on inward deterministic automata. In *Constraint Grammar workshop at the 18th Nordic Conference of Computational Linguistics (NODALIDA 2011)*, pages 50–60, 2011.

Anssi Yli-Jyrä. The Power of Constraint Grammars Revisited. In *Proceedings of the Constraint Grammar workshop at the 21th Nordic Conference of Computational Linguistics (NODALIDA 2017)*, 2017.

Appendix: User manual for gftest

`gftest` is a program for automatically generating systematic test cases for GF grammars. The basic use case is to give `gftest` a PGF grammar, a concrete language and a function; then `gftest` generates a representative and minimal set of example sentences for a human to look at. There are examples of actual generated test cases later in this document, as well as the full list of options to give to `gftest`. **It is recommended to compile your PGF with the gf flag** `--optimize-pgf`, otherwise this tool can be very slow. For example, `gf -make --optimize-pgf LangEng.gf`.

A live version of this user manual is found at `https://github.com/GrammaticalFramework/gftest#readme`. The version in this thesis is from October 2018.

Table of Contents

  - Unused fields: `-u`
  - Erased trees: `-r`
  - Debug information: `-d`

- Detailed information about the grammar
  - `--show-cats`
  - `--show-funs`
  - `--show-coercions`
  - `--show-contexts`
  - `--funs-of-arity`

Installation

Clone this repository `git clone https://github.com/GrammaticalFramework/gftest.git`. See installation instructions in the Travis file `https://github.com/GrammaticalFramework/gftest/blob/master/.travis.yml`.

Common use cases

Run `gftest --help` of `gftest -?` to get the list of options.

```
Common flags:
  -g --grammar=FILE        Path to the grammar (PGF) you want to test
  -l --lang="Eng Swe"      Concrete syntax + optional translations
  -f --function=UseN       Test the given function(s)
  -c --category=NP         Test all functions with given goal category
  -t --tree="UseN tree_N"  Test the given tree
  -s --start-cat=Utt       Use the given category as start category
     --show-cats           Show all available categories
     --show-funs           Show all available functions
     --funs-of-arity=2     Show all functions of arity 2
     --show-coercions      Show coercions in the grammar
     --show-contexts=8410  Show contexts for a given concrete type (given as FId)
     --concr-string=the    Show all functions that include given string
  -q --equal-fields        Show fields whose strings are always identical
  -e --empty-fields        Show fields whose strings are always empty
```

```
-u --unused-fields       Show fields that never make it into the top category
-r --erased-trees        Show trees that are erased
-o --old-grammar=ITEM    Path to an earlier version of the grammar
   --only-changed-cats   When comparing against an earlier version of a
                         grammar, only test functions in categories that have
                         changed between versions
-b --treebank=ITEM       Path to a treebank
   --count-trees=3       Number of trees of depth <depth>
-d --debug               Show debug output
-w --write-to-file       Write the results in a file (<GRAMMAR>_<FUN>.org)
-? --help                Display help message
-V --version             Print version information
```

Grammar: `-g`

Give the PGF grammar as an argument with `-g`. If the file is not in the same directory, you need to give the full file path.

It is recommended to compile your PGF with the gf flag `--optimize-pgf`, otherwise this tool can be very slow. For example, `gf -make --optimize-pgf FoodsEng.gf FoodsGer.gf`.

You can give the grammar with or without `.pgf`.

Without a concrete syntax you can't do much, but you can see the available categories and functions with `--show-cats` and `--show-funs`

Examples:

- `gftest -g Foods --show-funs`
- `gftest -g /home/inari/grammars/LangEng.pgf --show-cats`

Language: `-l`

Give a concrete language. It assumes the format `AbsNameConcName`, and you should only give the `ConcName` part.

You can give multiple languages, in which case it will create the test cases based on the first, and show translations in the rest.

Examples:

- `gftest -g Phrasebook -l Swe --show-cats`

- `gftest -g Foods -l "Spa Eng" -f Pizza`

Function(s) to test: `-f`

Given a grammar (`-g`) and a concrete language (`-l`), test a function or several functions.
Examples:

- `gftest -g Lang -l "Dut Eng" -f UseN`
- `gftest -g Phrasebook -l Spa -f "ByTransp ByFoot"`

You can use the wildcard `*`, if you want to match multiple functions. Examples:

- `gftest -g Lang -l Eng -f "*hat*"`

matches `hat_N`, `hate_V2`, `that_Quant`, `that_Subj`, `whatPl_IP` and `whatSg_IP`.

- `gftest -g Lang -l Eng -f "*hat*u*"`

matches `that_Quant` and `that_Subj`.

- `gftest -g Lang -l Eng -f "*"`

matches all functions in the grammar. (As of March 2018, takes 13 minutes for the English resource grammar, and results in ~40k lines. You may not want to do this for big grammars.)

Start category for context: `-s`

Give a start category for contexts. Used in conjunction with `-f`, `-c`, `-t` or `--count-trees`. If not specified, contexts are created for the start category of the grammar.
Example:

- `gftest -g Lang -l "Dut Eng" -f UseN -s Adv`

This creates a hole of `CN` in `Adv`, instead of the default start category.

Category to test: `-c`

Given a grammar (`-g`) and a concrete language (`-l`), test all functions that return a given category.
Examples:

- `gftest -g Phrasebook -l Fre -c Modality`
- `gftest -g Phrasebook -l Fre -c ByTransport -s Action`

Tree to test: `-t`

Given a grammar (`-g`) and a concrete language (`-l`), test a complete tree.

Example:

- `gftest -g Phrasebook -l Dut -t "ByTransp Bus"`

You can combine it with any of the other flags, e.g. put it in a different start category:

- `gftest -g Phrasebook -l Dut -t "ByTransp Bus" -s Action`

This may be useful for the following case. Say you tested `PrepNP`, and the default NP it gave you only uses the word *car*, but you would really want to see it for some other noun—maybe `car_N` itself is buggy, and you want to be sure that `PrepNP` works properly. So then you can call the following:

- `gftest -g TestLang -l Eng -t "PrepNP with_Prep (MassNP (UseN beer_N))"`

Compare against an old version of the grammar: `-o`

Give a grammar, a concrete syntax, and an old version of the same grammar as a separate PGF file. The program generates test sentences for all functions (if no other arguments), linearises with both grammars, and outputs those that differ between the versions. It writes the differences into files.

Example:

```
> gftest -g TestLang -l Eng -o TestLangOld
Created file TestLangEng-ccat-diff.org
Testing functions in…
<categories flashing by>
Created file TestLangEng-lin-diff.org
Created files TestLangEng-(old|new)-funs.org
```

- TestLangEng-ccat-diff.org: All concrete categories that have changed. Shows e.g. if you added or removed a parameter or a field.
- **TestLangEng-lin-diff.org** (usually the most relevant file): All trees that have different linearisations in the following format.

```
* send_V3


** UseCl (TTAnt TPres ASimul) PPos
         (PredVP (UsePron we_Pron)
                 (ReflVP (Slash3V3 _ (UsePron it_Pron))))
TestLangDut> we sturen onszelf ernaar
TestLangDut-OLD> we sturen zichzelf ernaar



** UseCl (TTAnt TPast ASimul) PPos
         (PredVP (UsePron we_Pron)
                 (ReflVP (Slash3V3 _ (UsePron it_Pron))))
TestLangDut> we stuurden onszelf ernaar
TestLangDut-OLD> we stuurden zichzelf ernaar
```

- TestLangEng-old-funs.org and TestLangEng-new-funs.org: groups the functions by their concrete categories. Shows difference if you have e.g. added or removed parameters, and that has created new versions of some functions: say you didn't have gender in nouns, but now you have, then all functions taking nouns have suddenly a gendered version. (This is kind of hard to read, don't worry too much if the output doesn't make any sense.)

Additional arguments to `-o`

The default mode is to test all functions, but you can also give any combination of `-s`, `-f`, `-c`, `--treebank`/`-b` and `--only-changed-cats`.

With `-s`, you can change the start category in which contexts are generated.

With `-f` and `-c`, it tests only the specified functions and categories. With `-b FILEPATH` (`-b`=`--treebank`), it tests only the trees in the file.

With `--only-changed-cats`, it only test functions in those categories that have changed between the two versions.

Examples:

- `gftest -g TestLang -l Eng -o TestLangOld` tests all functions
- `gftest -g TestLang -l Eng -o TestLangOld -s S` tests all functions in start category S

- `gftest -g TestLang -l Eng -o TestLangOld --only-changed-cats` tests only changed categories. If no categories have changed (and no other arguments specified), tests everything.
- `gftest -g TestLang -l Eng -o TestLangOld -f "AdjCN AdvCN" -c Adv -b trees.txt` tests functions, `AdjCN` and `AdvCN`; same for all functions that produce an `Adv`, and all trees in trees.txt.

Information about a particular string: `--concr-string`

Show all functions that introduce the string given as an argument.

Example:

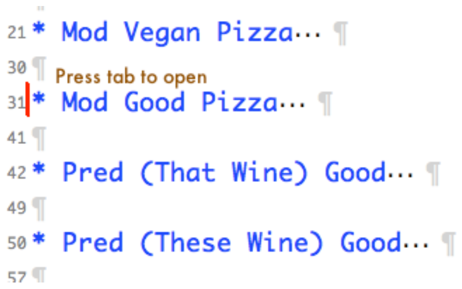
- `gftest -g Lang -l Eng --concr-string it`

which gives the answer ==> `CleftAdv, CleftNP, DefArt, ImpersCl, it_Pron`
(Note that you have the same feature in GF shell, command `morpho_analyse/ma`.)

Write into a file: `-w`

Writes the results into a file of format `<GRAMMAR>_<FUN or CAT>.org`, e.g. TestLangEng-UseN.org. Recommended to open it in emacs org-mode, so you get an overview, and you can maybe ignore some trees if you think they are redundant.

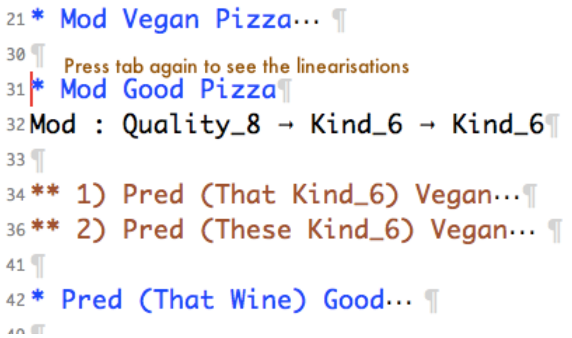
1) When you open the file, you see a list of generated test cases, like this:



```
* Mod Vegan Pizza…
* Mod Good Pizza…
* Pred (That Wine) Good…
* Pred (These Wine) Good…
```

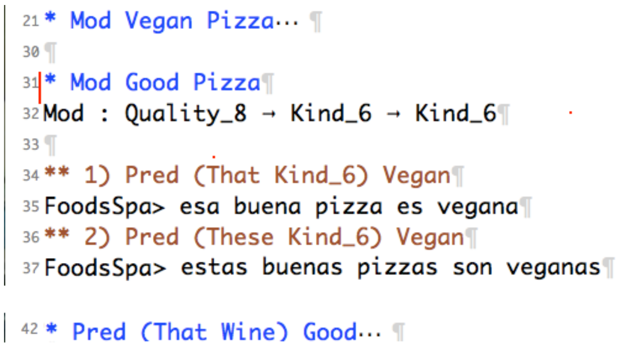
Place cursor to the left and click tab to open it.

2) You get a list of contexts for the test case. Keep the cursor where it was if you want to open everything at the same time. Alternatively, scroll down to one of the contexts and press tab there, if you only want to open one.



```
21 * Mod Vegan Pizza…
30
   Press tab again to see the linearisations
31 * Mod Good Pizza
32 Mod : Quality_8 → Kind_6 → Kind_6
33
34 ** 1) Pred (That Kind_6) Vegan…
36 ** 2) Pred (These Kind_6) Vegan…
41
42 * Pred (That Wine) Good…
```

3) Now you can read the linearisations.



```
21 * Mod Vegan Pizza…
30
31 * Mod Good Pizza
32 Mod : Quality_8 → Kind_6 → Kind_6
33
34 ** 1) Pred (That Kind_6) Vegan
35 FoodsSpa> esa buena pizza es vegana
36 ** 2) Pred (These Kind_6) Vegan
37 FoodsSpa> estas buenas pizzas son veganas

42 * Pred (That Wine) Good…
```

If you want to close the test case, just press tab again, keeping the cursor where it's been all the time (line 31 in the pictures).

Less common use cases

Empty or always identical fields: `-e`, `-q`

Information about the fields: always empty, or always equal to each other. Example of empty fields:

```
> gftest -g Lang -l Dut -e
* Empty fields:
==> Ant: s
==> Pol: s
==> Temp: s
==> Tense: s
==> V: particle, prefix
```

The categories `Ant`, `Pol`, `Temp` and `Tense` are as expected empty; there's no string to be added to the sentences, just a parameter that *chooses* the right forms of the clause. `V` having empty fields `particle` and `prefix` is in this case just an artefact of a small lexicon: we happen to have no intransitive verbs with a particle or prefix in the core 300-word vocabulary. But a grammarian would know that it's still relevant to keep those fields, because in some bigger application such a verb may show up. On the other hand, if some other field is always empty, it might be a hint for the grammarian to remove it altogether.

Example of equal fields:

```
> gftest -g Lang -l Dut -q
* Equal fields:
==> RCl:
s Pres Simul Pos Utr Pl
s Pres Simul Pos Neutr Pl

==> RCl:
s Pres Simul Neg Utr Pl
s Pres Simul Neg Neutr Pl

==> RCl:
s Pres Anter Pos Utr Pl
s Pres Anter Pos Neutr Pl

==> RCl:
s Pres Anter Neg Utr Pl
s Pres Anter Neg Neutr Pl

==> RCl:
s Past Simul Pos Utr Pl
s Past Simul Pos Neutr Pl
…
```

Here we can see that in relative clauses, gender does not seem to play any role in plural. This could be a hint for the grammarian to make a leaner parameter type, e.g. `param RClAgr = SgAgr <everything incl. gender> | PlAgr <no gender here>`.

Unused fields: `-u`

These fields are not empty, but they are never used in the top category. The top category can be specified by `-s`, otherwise it is the default start category of the grammar.

Note that if you give a start category from very low, such as `Adv`, you get a whole lot of categories and fields that naturally have no way of ever making it into an adverb. So this is mostly meaningful to use for the start category.

Erased trees: `-r`

Show trees that are erased in some function, i.e. a function `F : A -> B -> C` has arguments A and B, but doesn't use one of them in the resulting tree of type C. This is usually a bug.

Example:

```
> gftest -g Lang -l "Dut Eng" -r

* Erased trees:

** RelCl (ExistNP something_NP) : RCl
- Tree:  AdvS (PrepNP with_Prep
              (RelNP (UsePron it_Pron)
                     (UseRCl (TTAnt TPres ASimul) PPos
                             (RelCl (ExistNP something_NP)))))
                                    (UseCl (TTAnt TPres ASimul) PPos (ExistNP something_NP))
- Lin:   ermee is er iets
- Trans: with it, such that there is something, there is something

** write_V2 : V2
- Tree:  AdvS (PrepNP with_Prep
              (PPartNP (UsePron it_Pron) write_V2))
                       (UseCl (TTAnt TPres ASimul) PPos (ExistNP something_NP))
- Lin:   ermee is er iets
- Trans: with it written there is something
```

In the first result, an argument of type `RCl` is missing in the tree constructed by `RelNP`, and in the second result, the argument `write_V2` is missing in the tree constructed by `PPartNP`. In both cases, the English linearisation contains all the arguments, but in the Dutch one they are missing. (This bug is already fixed, just showing it here to demonstrate the feature.)

Detailed information about the grammar

Debug information: `-d`

When combined with `-f`, `-c` or `-t`, two things happen:

1) The trees are linearised using `tabularLinearize`, which shows the inflection table of all forms.
2) You can see traces of pruning that happens in testing functions: contexts that are common to several concrete categories are put under a separate test case.

When combined with `--show-cats`, also the concrete categories are shown.

--show-cats

Shows the categories in the grammar. With `--debug`/`-d`, shows also concrete categories.

Example:

```
> gftest -g Foods -l Spa --show-cats -d

* Categories in the grammar:
Comment
    Compiles to concrete category   0
Item
    Compiles to concrete categories 1—4
Kind
    Compiles to concrete categories 5—6
Quality
    Compiles to concrete categories 7—8
Question
    Compiles to concrete category   9
```

--show-funs

Shows the functions in the grammar. (Nothing fancy happens with other flags.)

--show-coercions

First I'll explain what *coercions* are, then why it may be interesting to show them. Let's take a Spanish Foods grammar, and consider the category `Quality`, e.g. `Good` and `Vegan`. `Good` "bueno/buena/buenos/buenas" goes before the noun it modifies, whereas `Vegan` "vegano/vegana/…" goes after, so these will become different *concrete categories* in the PGF: `Quality_before` and `Quality_after`. (In reality, they are something like `Quality_7` and `Quality_8` though.)

Now, this difference is meaningful only when the adjective is modifying the noun: "la buena pizza" vs. "la pizza vegana". But when the adjective is in a predicative position, they both behave the same: "la pizza es buena" and "la pizza es vegana". For this, the grammar creates a *coercion*: both `Quality_before` and `Quality_after` may be treated as `Quality_whatever`. To save some redundant work, this coercion `Quality_whatever` appears in the type of predicative function, whereas the modification function has to be split into two different functions, one taking `Quality_before` and other `Quality_after`.

Now you know what coercions are, this is how it looks like in the program (just mentally replace 7 with `before`, 8 with `after` and 11 with `whatever`.):

```
> gftest -g Foods -l Spa --show-coercions
* Coercions in the grammar:
Quality_7--->_11
Quality_8--->_11
```

--show-contexts

Show contexts for a given concrete category, given as an FId (i.e. Int). The concrete category may be a coercion or a normal category. By combining with `-s`, you can change the start category of the context. (You can get a list of all concrete categories by pairing `--show-cats` with `--debug`: see `--show-cats`.)
Examples:

- First, find out some concrete categories:

  ```
  > gftest -g Foods -l Spa --show-cats -d
  …
  Quality
      Compiles to concrete categories 7–8
  …
  ```

- Then, list the contexts for some of them, say `Quality_7`:

```
> gftest -g Foods -l Spa --show-contexts 7

Pred (That (Mod _ Wine)) Vegan
Pred (That Wine) _
Pred (These (Mod _ Wine)) Vegan
Pred (These Wine) _
Pred (That (Mod _ Pizza)) Vegan
Pred (That Pizza) _
Pred (These (Mod _ Pizza)) Vegan
Pred (These Pizza) _
```

- You can also give it a range of arguments, with starting and ending category in quotes: `gftest -g Foods -l Spa --show-contexts "7 10"`
- Check out from `--show-coercions` how to find coercions, and you can try `--show-contexts` with them:

```
> gftest -g Foods -l Spa --show-contexts 11

Pred (That Wine) _
Pred (These Wine) _
Pred (That Pizza) _
Pred (These Pizza) _
```

--count-trees

Number of trees up to given size. Gives a number how many trees, and a couple of examples from the highest size. Examples:

```
> gftest -g TestLang -l Eng --count-trees 10
There are 675312 trees up to size 10, and 624512 of exactly size 10.
For example:
* AdvS today_Adv (UseCl (TTAnt TPres ASimul) PPos (ExistNP (UsePron i_Pron)))
* UseCl (TTAnt TCond AAnter) PNeg (PredVP (SelfNP (UsePron they_Pron)) UseCopula)
```

This counts the number of trees in the start category. You can also specify a category:

```
> gftest -g TestLang -l Eng --count-trees 4 -s Adv
There are 2409 trees up to size 4, and 2163 of exactly size 4.
For example:
* AdAdv very_AdA (PositAdvAdj young_A)
* PrepNP above_Prep (UsePron they_Pron)
```

--funs-of-arity

Show all functions of given arity (not up to).

Example:

```
> gftest -g Phrasebook --funs-of-arity 3
* Functions in the grammar of arity 3:
ADoVerbPhrasePlace
AModVerbPhrase
HowFarFromBy
QWhereModVerbPhrase
```