Automatic systematic test case generation for producing reliable grammars

Abstract

We present a method for finding errors in formalized natural language grammars, by automatically and systematically generating test cases that are intended to be judged by a human oracle. The method works on a per-construction basis; given a construction from the grammar, it generates a finite but complete set of test sentences (typically tens or hundreds), where that construction is used in all possible ways. Our method is an alternative to using a corpus or a treebank, where no such completeness guarantees can be made. The method is language-independent and is implemented for the grammar formalism PMCFG, but also works for weaker grammar formalisms. We evaluate the method on a number of different grammars for different natural languages, with sizes ranging from toy examples to real-world grammars.

1 Introduction

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.

There are several ways to test grammars that do not involve human labour. The morphology and lexicon can be compared against existing resources, or if there are none, a large corpus of any text should give indications whether the word forms are correct. The same corpus can be used to test the coverage of the grammar: how many sentences are successfully parsed. However, often we want information beyond numbers: do the rules we wrote for relative clauses correctly accept all relative clauses and nothing else? In other words, we are interested in the strong generative capacity (Chomsky, 1963) of the grammar, i.e. combinations of a string and its structural description.

Here is an example of a typical situation we want to improve. Suppose a grammarian implements relative clauses, and then comes up with a test suite of sentences with their analyses. The next grammarian implements relative clauses for another language, and adapts the test set to the new language. Every time someone touches relative clauses in any language, the test suite will be rerun and verified by someone who knows the language, or compared to the original gold standard, if there is one. This scheme can fail for various reasons:

- The original list is not exhaustive: for instance, it tests only “X, who loves me” but not “X, whom I love”.
- The original list is exhaustive in one language, but not in all: for instance, it started in English and only included one noun, but in French it would need at least one masculine and one feminine noun.
- The list is overly long, with redundant test cases, and human testers are not motivated to read through.
- A grammarian makes a change somewhere else in the grammar, and does not realize that it affects relative clauses, and thus does not rerun the appropriate test suite.

We present a method that addresses these problems, by designing a tool that can automatically generate test cases, given a grammar and a syntactic function that we want to test. Our tool improves on the four weak points in the following way:

- The set of test cases consists of the syntactic function, applied to all relevant arguments and the result is placed in all relevant contexts ("relevant" is defined in Section 3).
- The tests cases are automatically generated for each different language. If there is a parameter of gender or noun class in the grammar, then the program is guaranteed to choose an example of each of them, when it matters.
- When some feature doesn't matter, the test cases are pruned: for example, in English we need to test a reflexive construct with three different 3rd person singular subjects, because the object has to agree with the subject: "he sees himself", "she sees herself" and "it sees itself". With a non-reflexive object, it is enough to test with only one of *he, she, it*, or any singular noun or proper name, because the agreement only shows in the verb form, thus generating both "she sees a dog" and "John sees a dog" is redundant.
- The grammar is a collection of grammatical categories, syntactic functions and a lexicon, and everything is interconnected. By changing e.g. the category of prepositions, the changes are propagated to several functions, e.g. for building adjuncts ("in the house") and complements ("believe in something"). This is all part of the grammar, and the chain of effects can be automatically traced.

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 method works for any formalism that is at most as expressive as PMCFG, including formalisms such as Tree-Adjoining Grammar (TAG) (Joshi et al., 1975) and Combinatorial Categorial Grammar (CCG) (Steedman, 1988).

2 Grammatical Framework

Grammatical Framework (GF) (Ranta, 2004) is a framework for building multilingual grammar applications. Its main components are a functional programming language for writing grammars and a resource library (Ranta, 2009), which, as of March 2018, contains the linguistic details of 40 natural languages. The library has had over 50 contributors, and it consists of 1900 program modules and 3 million lines of code. GF is well suited for creating domain-specific systems: examples include mathematics (Caprotti, 2006), legal documents (Camilleri, 2017) and information extraction (Safwat et al., 2015).

A GF grammar consists of an *abstract syntax*, which is a set of grammatical categories and functions between them, and one or more *concrete syntaxes*, which describe how the abstract functions and categories are linearized, i.e. turned into surface strings. The resulting grammar describes a mapping between concrete language strings and their corresponding abstract trees. This mapping is bidirectional—strings can be *parsed* to trees, and trees *linearized* 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 linearizing the obtained tree into a new string.

Figure 1 shows a small example of a GF abstract grammar. The grammar generates noun phrases for a lexicon of 15 words (*a, the, ..., without*) in four lexical categories, and five 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 paper.

```
abstract NounPhrases = {
  flags startcat = NP ;
  cat
    NP ; Adv ;                      -- Non-terminal categories
    CN ; Det ; Adj ; Prep ;         -- Terminal (lexical) categories
  fun
    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, ready : Adj ;
    house, hill : CN ;
    in, on, with, without : Prep ;
}
```

Figure 1: GF grammar for noun phrases

As examples that help illustrate different testing needs for different languages, let us take three language-specific phenomena in the scope of our small grammar: preposition contraction in Dutch, adjective agreement in Estonian and determiner placement in Basque.

2.1 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:

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

2.2 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. The following table shows the different behaviours in attributive position, with *sinine* 'blue' as an example of a regular adjective, and *valmis* 'ready' as an invariable.

(1)	sinises	majas
	blue-SG.INE	house-SG.INE
	'in a blue house'	
	sinistes	majades
	blue-PL.INE	house-PL.INE
	'in blue houses'	

(2)	sinise	majaga
	blue-SG.GEN	house-SG.COM
	'with a blue house'	
	siniste	majadega
	blue-PL.GEN	house-PL.COM
	'with blue houses'	

(3)	valmis	majas
	ready.SG.NOM	house-SG.INE
	'in a finished house'	
	valmis	majades
	ready.SG.NOM	house-PL.INE
	'in finished houses'	

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

2.3 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:

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

2.4 Using our tool

We have seen that, in order to test whether or not we have implemented a linguistic phenomenon correctly, we take a single function as a base, and describe all combinations of arguments that are needed to test the function. If the result of the function is an inflection table rather than a fully specified result, then we need several *contexts* to squeeze out all the different forms. For example, a CN in English is open for number—house is really a table {Sg => *"house"* ; Pl => *"houses"*}, and applying a determiner chooses the right form: DetCN this house linearizes to *this house* and DetCN these house linearizes to *these houses*.

The example grammar, with only 15-word lexicon and 5 syntactic functions, generates over 10,000 trees[1] up to depth 5. However, as we have seen in the examples above, we can test complex morphosyntactic phenomena with just a set of 4, 8 or 12 trees, depending on the complexity of the language.

As mentioned previously, the base of a test case is one syntactic function, but often the same sentence ends up showcasing several functions. In the Estonian example, we start from AdjCN and end up in a context formed by PrepNP—in fact, these 8 trees are exactly the same that we would've chosen to test PrepNP. Thus, it is possible to shrink the test cases effectively, if one wants to test the whole grammar at one go.

Of course, such a test set will not catch e.g. individual misspellings, or more systematic bugs in the morphological paradigms. But there are easier methods to test for such bugs—our goal is to test the more abstract, syntactic phenomena with as few trees as possible.

3 How the tool works

A GF grammar compiles into a low-level format called PGF ("Portable Grammar Format"), which is processed by our tool. For each category in the original grammar, the GF compiler introduces a new category in the PGF for each combination of parameters. For example for English adjectives, we get $\mathtt{A} \Rightarrow \{\mathtt{A}_{\mathtt{pos}}, \mathtt{A}_{\mathtt{comp}}, \mathtt{A}_{\mathtt{superl}}\}$, and for Spanish, $\mathtt{A} \Rightarrow \{\mathtt{A}_{\mathtt{pos}\times\mathtt{sg}\times\mathtt{masc}}, \ldots, \mathtt{A}_{\mathtt{superl}\times\mathtt{pl}\times\mathtt{fem}}\}$.

This compilation step can dramatically increase the number of categories of the grammar, but it also removes the need for dealing with these parameters explicitly when we generate test cases. Instead, each

[1]e.g. DetCN the (AdvCN (PrepNP on (DetCN a (AdjCN small hill)) (AdjCN blue house)) 'the blue house on a small hill'

syntactic function from the original grammar turns into multiple syntactic functions into the PGF – one for each combination of parameters of its arguments.

We now describe the generation of test cases for a given syntactic function. We assume that all test cases are trees with the same start (top-level) category, such as `NP` in our example grammar, or `S` (for sentence) for more general grammars. The requirement is that the start category is linearized as one string only. (In a PMCFG, categories in general can be linearized to vectors of strings, which is perhaps unsuitable for test cases that are presented to a human.)

Enumerate functions As we explained before, each syntactic function turns into multiple versions, one for each combination of parameters of its arguments. We test each of these versions seperately. This enumeration is the main reason we see several test cases in the examples in Section 2.

In order to construct trees that use the syntactic function, we need to supply it with *arguments*, as well as put the resulting tree into a *context* that produces a tree in the correct start category.

Enumerate arguments Some syntactic functions are simply a single lexical item (for example the word *small*); in this case just the tree `small` is our answer. If we choose a function with arguments, such as for example `PrepNP`, then we have to supply it with argument trees. Each argument tree 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 information from its arguments, in the right way. The information that a syntactic function uses is any of the strings that come from linearizing its arguments. In order to be able to see which string in the result comes from which string from which arguments, we want to generate test cases that only contain unique strings (no duplicates).

For example, when we test a preposition together with a noun, we want to pick a noun that actually has different forms (unique strings) for different prepositions, rather than having identical forms, because the human would not be able to see if the `PrepNP` function picked the wrong form.

It is often possible to generate one combination of arguments where all strings in the linearizations are different. However, it is not always possible to 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. In this way, if the syntactic function contains a mistake, there is always one test case that reveals it.

Example: Test cases using `AdjCN` Let us test the function `AdjCN : Adj → CN → CN`, and take a Spanish concrete syntax as an example. Firstly, we need a minimal and representative set of arguments of types `Adj` and `CN`. Consider the nouns first: Spanish has a grammatical gender, so in order to be representative, we need an example of both masculine and feminine. Out of the small lexicon, `house` (*casa*) is feminine, and `hill` (*cerro*) is masculine, so we return those two nouns as the full set of argument trees in `CN`.

Secondly, we consider the adjectives. Most commonly, adjectives follow the noun, e.g. *casa pequeña* 'small house', but some adjectives precede the noun, e.g. *buena casa* 'good house'. Thus in order to cover the full spectrum of adjective placement, we need one premodifier and one postmodifier adjective. We pick the words `good` and `small` as the arguments of type `Adj`.

Now, our full set of test cases are `AdjCN` applied to the cross product of $\left\{\begin{smallmatrix}\texttt{good}\\ \texttt{small}\end{smallmatrix}\right\} \times \left\{\begin{smallmatrix}\texttt{house}\\ \texttt{hill}\end{smallmatrix}\right\}$. Note that `CN` is unspecified for number, because it is still waiting for a determiner (e.g. `this` or `these`) to complete it into an `NP`. Thus all the test cases contain both singular and plural variants, and by linearizing these 4 trees, we get the 8 strings shown in Figure 2.

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 result into a context, so that we can generate a single string from the result (a sentence). We do this for all trees we have generated so far.

The important thing here is that the generated set of contexts shows all the possible different ways the tree can be used. For example, for a test tree with an inflection table of size 4, we would generate 4 different sentences in which each of the 4 inflections is used.

`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

Figure 2: Agreement and placement of adjectives in attributive position

By *context*, we mean a tree in the start category, with a *hole* of type `CN`. A tree of type `CN` can be plugged into the hole to form a tree in the start category. In this grammar, the only category above `CN` is `NP`, and there is only one way that a `CN` can end up in an `NP`: by using the function `DetCN : Det` $\rightarrow$ `CN` $\rightarrow$ `NP`.

Let us return to our running example. So far the relevant features have been grammatical gender an adjective placement—we have 4 trees with all combinations of {masculine, feminine} and {pre,post}. The resulting trees have *variable* number, and we need to generate contexts that specify that number. In the PGF, the function `DetCN` actually comes in two versions: one for singular and one for plural determiners. Both of these use their arguments in different ways (different strings from the hole appear in the result). So, both versions of `DetCN` lead to one context each. The final contexts are `DetCN this _` and `DetCN these _`. Note that we do not have to enumerate all possible first arguments to `DetCN`. We insert the 4 test cases into the holes, and get 8 trees in total: {`DetCN this (AdjCN good house), ..., DetCN these (AdjCN small hill)`}.

So, what we want to compute is, given the result category T of the syntactic function, and the start category S of the grammar, a minimal set of contexts in the start category S with hole of type T, such that any string appearing in the linearization of T also appears somewhere in the linearization of S. We compute this by setting up a system of equations for each category C in the grammar: for each C, we define all the relevant contexts with hole type C in terms of all the relevant contexts with hole type C' for other categories C' that use C. So, the answer for each category is expressed in terms of the answer for other categories. In general, this system of equations is *recursive*, and we use a fixpoint iteration to compute the smallest solution.

Pruning For the previous example, we did not need any pruning: the cross product of all relevant distinctions produced a minimal and representative set of trees. Now assume we have a larger grammar, which also covers adjectives in a predicative position: e.g. *esta casa es pequeña* 'this house is small' and *esta casa es buena* 'this house is good'. The distinction which made us choose `small` and `good` in the first place is now gone: in predicative position, both adjectives behave the same. Thus, in this larger grammar, when we want to test adjectives, it is necessary to include two examples when in attributive position, but only one when in predicative position.

4 General features of PMCFG: unused, equal, erased or empty fields

Aside from concrete language-dependent phenomena, there are more general, engineering questions a grammar writer may ask. For instance, say that our concrete type for a `CN` in Dutch 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. A yes answer to any of these may indicate a bug in the grammar.

In Dutch, nominative and accusative are only different for pronouns, so for this grammar we would indeed find out that case is redundant: all nominative and accusative fields would be identical. As grammarians, we could decide to keep the distinction for further extension of the grammar—maybe we want to add pronouns in the future—or remove it as redundant.

GF has the expressivity of PMCFG, which means that it is possible to erase arguments: say that there is a bug, `AdjCN : Adj` $\rightarrow$ `CN` $\rightarrow$ `CN` never actually adds the adjective to the new `CN`, in which case

`AdjCN blue house` and `house` are linearized identically. Instead of testing every single function, we would like to know if there are any functions in the grammar that behave like this.

The analyses mentioned in this section are implemented in a similar way to the method for enumerating all contexts.

5 Use cases and evaluation

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*. The test cases reveal a bug, where `DetNP your` picks a gender too soon, instead of leaving it open in an inflection table. We implement a fix by adding 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 have not caused new bugs in other 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 fixes, 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 without *that*" and pointing 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 broke.

	Concrete grammar →		**Dutch**		**Basque**		**Estonian**	
↓ Abstract grammar	#funs+lex	#trees	#total	#uniq	#total	#uniq	#total	#uniq
Noun phrases	5+15	>10,000	21	18	33	27	40	36
Phrasebook	130+160	>480,000	2006	1892	2808	2650	1513	1314
Resource grammar	217+446	>500 billion	59,316	51,145	278,092	216,058	60,600	38,517

Table 1: Test cases for all functions in three grammars

Resource grammar function		**Dutch**		**Basque**		**Estonian**	
		#trees in contexts	#combinations of args	#t	#c	#t	#c
`ComparA`	'younger than me'	11	3	36	6	21	3
`RelNP`	'a cat that I saw'	25	9	90	10	123	12
`ReflVP`	'see myself'	1655	23	10838	128	1608	13

Table 2: Test cases for some individual functions in the resource grammar

In order to evaluate our method, we generate test cases for grammars of varying sizes, using the three languages presented earlier: Dutch, Estonian and Basque. These languages come from different language families, and cover a wide range of grammatical complexity. Dutch, an Indo-European language, has fairly simple nominal and verbal morphology, but the rules for handling word order, prefixes and particles in verb phrases are somewhat intricate. Estonian and Basque both have rich morphology, each featuring 14 nominal cases, but Basque verb morphology is much more complex, with agreement in subject, object and indirect object. Contrary to our expectations, we found Dutch and Estonian behaving similarly to each other and Basque significantly worse, both in execution time and examples generated.

Table 1 shows the number of generated trees for in total for all syntactic functions in the three grammars, and Table 2 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.

5.1 Grammars

The first grammar is the toy example introduced earlier in this article: NounPhrases with 5 syntactic functions and 15 words 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 three concrete syntaxes were completed in less than an hour, by an experienced grammarian with knowledge in all three 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[2].

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. However, we should not limit the lexicon too much, because we may miss important distinctions in some languages—to give a hypothetical example, some grammar may have a bug that shows up only in animate nouns which end in a consonant. This subset of the resource grammar produces hundreds of billions of trees up to depth 5.

5.2 Results

Execution time We ran all the experiments on a MacBook Air with 1,7 GHz processor and 8 GB RAM. For the smaller grammars, all languages took just seconds to run. 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 several hours. We ran the experiment in smaller batches over two days, and noticed a lot of variance: functions that handle e.g. noun phrases, adjectives and adverbs ran in a few minutes, but a function involving verb phrases could take an hour just by itself.

Finding bugs We read through the test sentences of the small grammar in all the three languages. For Dutch we had a native speaker; for Estonian an intermediate non-native, and for Basque, a beginner. None of the three grammars had been tested systematically before—(Listenmaa and Kaljurand, 2014) report testing the morphological paradigms extensively against existing resources, and syntactic functions with a treebank of 425 trees.

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

We have been developing the tool by testing it on the Dutch resource grammar, and during 4 months, we have committed 16 bugfixes 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!)

The Basque resource grammar is still a 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. We think that the best course of action is to evaluate the morphological paradigms against existing resources, fix the implementation, and then concentrate on syntax. The Phrasebook was implemented using the resource grammar, so the same problems apply.

We read through the first 500 sentences from Estonian Phrasebook, which took around 20 minutes. We found 3 errors in the inflection of individual words (they did not come from the resource lexicon, which was tested, but were implemented separately in the grammar); one error of type "Spaniard restaurant", and one suggestion for a more idiomatic construction. As expected, Phrasebook sentences were easier to read, and made more sense semantically than sentences from the resource grammar.

5.3 Previous 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. (Bender et al., 2010) describe a system for creating and testing HPSG (Pollard and Sag, 1994) grammars, by using a detailed questionnaire about the features of the given language. This system achieves both generating the grammar rules and testing them simultaneously, whereas our method relies on an existing grammar. On the other hand, our system is more general to any kinds of grammars, including application grammars where the distinctions are not syntactic but semantic.

6 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 `AdvQVP` covers constructions such as "you did *what*?". However, the function itself is completely general and can take any verb phrase and any question adverb, thus bizarre combinations like "you saw the dog why" may appear in the generated test cases. For future work, we plan to use an external treebank to guide the algorithm to pick trees that also make sense semantically.

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.

References

[Bender et al.2010] Emily M Bender, Scott Drellishak, Antske Fokkens, Michael Wayne Goodman, Daniel P Mills, Laurie Poulson, and Safiyyah Saleem. 2010. Grammar prototyping and testing with the lingo grammar matrix customization system. In *Proceedings of the ACL 2010 system demonstrations*, pages 1–6. Association for Computational Linguistics.

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

[Camilleri2017] John J. Camilleri. 2017. *Contracts and Computation–Formal modelling and analysis for normative natural language*. Ph.d. thesis, University of Gothenburg, Gothenburg, Sweden, October.

[Caprotti2006] Olga Caprotti. 2006. Webalt! deliver mathematics everywhere. In *Society for Information Technology & Teacher Education International Conference*, pages 2164–2168. Association for the Advancement of Computing in Education (AACE).

[Chomsky1963] Noam Chomsky. 1963. Formal properties of grammars. *Handbook of mathematical psychology*, pages 323–418.

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

[Joshi et al.1975] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. 1975. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

[Khegai2006] Janna Khegai. 2006. *Language engineering in Grammatical Framework (GF).* Ph.D. thesis, Chalmers University of Technology.

[Listenmaa and Kaljurand2014] Inari Listenmaa and Kaarel Kaljurand. 2014. Computational Estonian Grammar in Grammatical Framework. In *9th SALTMIL workshop on free/open-source language resources for the machine translation of less-resourced languages*.

[Marcus et al.1993] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

[Nivre et al.2016] 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. 2016. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the 10th edition of the Language Resources and Evaluation Conference (LREC 2016).*

[Pollard and Sag1994] Carl Pollard and Ivan Sag. 1994. *Head-driven phrase structure grammar*. University of Chicago Press.

[Ranta et al.2012] Aarne Ranta, Ramona Enache, and Grégoire Détrez. 2012. 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. Springer-Verlag.

[Ranta2004] Aarne Ranta. 2004. Grammatical Framework. *Journal of Functional Programming*, 14(2):145–189.

[Ranta2009] Aarne Ranta. 2009. The GF Resource Grammar Library. *Linguistics in Language Technology*, 2.

[Safwat et al.2015] Hazem Safwat, Normunds Gruzitis, Ramona Enache, and Brian Davis. 2015. Embedded controlled languages to facilitate information extraction from eGov policies. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services (iiWAS).*

[Seki et al.1991] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2):191–229.

[Steedman1988] Mark Steedman. 1988. Combinators and grammars. In *Categorial Grammars and Natural Language Structures*, pages 417–442.