Orthography Engineering in Grammatical Framework

Krasimir Angelov
University of Gothenburg
krasimir@chalmers.se

Abstract

Orthography is an integral part of language but in grammar engineering it is often ignored, simplified or just delegated to external tools. We present new extensions to Grammatical Framework, which allow one and the same formalism to describe both orthography, syntax and morphology. These extensions are also easily generalizable to other formalisms.

1 Introduction

Orthography is often assumed to be something simple that is easily delegated to pre- or post-processors. The real grammar engineering starts only after the tokenization.

Unfortunately if this is mostly true for English, it is more complicated in other languages. For instance, most Germanic languages tend to build compound nouns composed of one or more simple nouns. Furthermore German requires that nouns should start with a capital letter unless if the noun is in the second part of a compound. Handling compounds requires a separate compound splitter (Koehn and Knight, 2003) for parsing and proper generator in linearization. The capitalization is usually ignored in parsing but must be recovered for generation (Lita et al., 2003; Chelba and Acero, 2004).

Agglutinative languages tend to build long words by adding more and more suffixes which blurs the borderline between word and morpheme. In that case the words themselves need to be parsed since it is not possible to enumerate all word forms in a finite lexicon. The extreme case is in languages that does not separate words with spaces at all. This is usually solved by using a preprocessor that finds a lattice of possible word segmentations which are later parsed (Chappelier et al., 1999; Hall, 2005).

Finally, in many languages there are lexical units that are phonetically dependent on the context. A typical example is the indefinite article *a/an* in English which is different depending on the next word in the sentence. Similarly the definite article *la/l* in French depends on the next word, except that the correct resolution also requires knowledge of a syntactic context which is available only in the grammar.

Because of all these complications, delegating the orthography to an external tool has many engineering disadvantages. To start with, a new tool has to be developed for every language. The tool moreover should partly encode knowledge that is already in the grammar. For instance for compound splitting the tool should have access to the lexicon of the grammar. When an application is ported from one platform to another then the grammar itself is usually stored in a portable format and only the grammar interpreter needs to be ported. However if external pre- and post-processors are used then they have to be ported as well. Everything is a lot simpler if the orthography is encoded in a portable way as part of the grammar itself.

We present extensions to the Grammatical Framework (GF; Ranta 2011) which allow orthographic conventions to be encoded as an integral part of the grammar. This possess the following challenges.

First of all GF is a reversible formalism. One and the same grammar is used for both parsing and generation. In parsing we want grammars that are robust and permissible as long as this does not produce incorrect analyses. On the other hand in generation we want to produce text of the best possible shape. This means for instance that accepting a German noun that is not capitalized can be desirable but generating a text where the nouns are not capitalized should be avoided.

GF is by design a multilingual formalism. A

Proceedings of the Grammar Engineering Across Frameworks (GEAF) Workshop, 53rd Annual Meeting of the ACL and 7th IJCNLP, pages 33–40,
Beijing, China, July 26-31, 2015.

single grammar typically contains modules for several different languages. The modules are linked together through a Logical Framework[1] (Harper et al., 1993) which serves as a language independent abstract syntax. In this multilingual setting, having different pre- or post-processors for the different languages will defeat the purpose of having a single multilingual grammar.

GF grammars are distributed in a portable format (Angelov et al., 2010) which can be deployed in different environments ranging from web servers and desktop translation systems to mobile devices. The virtual machine for GF (Angelov, 2011) is also developed as a platform independent software. By adding the orthographic extensions to the framework itself, we automatically make more GF applications portable since they will not be dependent on external tools.

The grammarian in GF writes a grammar by using a high-level functional programming language reminiscent of Haskell and ML. However, if we abstract away from the high-level features, the backbone of the framework (Ljunglöf, 2004) is equivalent to a Parallel Multiple Context-Free Grammar (PMCFG; Seki et al. 1991). The latter subsumes other popular formalisms such as TAG (Joshi and Schabes, 1997) and CCG (Steedman, 2000) but contains only some of the possible RCG grammars (Boullier, 1998). The full logical framework embedded in GF in principle makes GF as expressive as HPSG (Pollard and Sag, 1994) and other unification grammars, but we often find this extra level of complexity unnecessary and we stick with the backbone of the framework.

The extensions that we present are mostly framework independent and they can be added even to simple context-free grammars. Because of that and to make it easier to relate the extensions to other formalisms we will use context-free grammars for the rest of the paper. The actual implementation is in the PMCFG engine behind GF.

There are four groups of extensions that we present in four sections:

- `BIND`, `SOFT_BIND` and `SOFT_SPACE` in Section 2
- `CAPIT` and `ALL_CAPIT` in Section 3
- `pre` in Section 4
- `nonExist` in Section 5

All of these extensions were frequently requested by different people in the GF community. They are now available as primitive operations of type string (`Str`) and are exported from the standard module `Prelude`. The only exception is `pre`, which is a complex programming language construction. In a public application the extensions were first extensively used in the offline mobile translator for twelve languages developed in Angelov et al. (2014).

2 Controlling the Spaces

The first problem is to decide whether and when to put spaces between words. Like all formalisms, GF describes a language as a set of strings over a finite set of terminal symbols. This is problematic in agglutinative languages and languages with compound words since their vocabulary is theoretically infinite.

The solution is to redefine what is considered a word in the language. For instance, a Swedish grammar should treat compound words such as *datavetenskap* 'computer science' as two separate words *data* and *vetenskap*, which, following the orthographic convention in Swedish, are written without space in between. The grammar encodes that the two words are bound together by inserting a special token called BIND. This can be exemplified with a rule like:

```
fun CompoundN : N -> N -> N
lin CompoundN n1 n2 =
          n1 ++ BIND ++ n2
```

Where `CompoundN` combines the two nouns referred by the two variables `n1` and `n2` into a single compound noun. Obviously a realistic rule will be more complicated but the example captures the essense. The operator `++` specifies that we combine words together to build a phrase. We put the nouns one after another but we also insert `BIND` to indicate that there is no space in-between. The result from:

```
CompoundN data_N vetenskap_N
```

is the compound *datavetenskap*. In contrast if we had missed the `BIND`, we would generate *data␣vetenskap* which is not the correct Swedish spelling. Both the types of the arguments as well as the type of the result is noun (`N`), which allows for compounds with multiple components with alternative associativities.

[1]The name Grammatical Framework (GF) itself is the analogy of the general Logical Framework in Harper et al.

	0	$1 \ldots \infty$	$0 \ldots \infty$
0	BIND	$*$	SOFT_BIND
1	$*$		SOFT_SPACE

Table 1: Tokens for controlling the spacing. The column shows how many spaces the parser will accept and the row shows how many spaces are put by the generator. Inconsistent combinations are marked with $*$.

The same mechanism makes it possible to model agglutinative languages. There we use a finite lexicon of words but we are free to add suffixes syntactically. The suffixes are attached by using BIND which prevents the insertion of unnecessary spaces. This has been used extensively in Finnish where the lexicon is composed of stems and suffixes while the words are composed syntactically. To a lesser extend the same technique is also applied in Estonian (Listenmaa and Kaljurand, 2014) and Maltese (Camilleri, 2013).

BIND is useful even in English. For example the grammars for all languages including English must parse numerals in order to recognize whether the numerals require singular or plural noun, i.e. "1 apple" but "2 apples". When we have numerals with more than one digit then the grammar must use BIND to glue individual digits together.

A related issue is that in many languages the punctuation signs are glued to the previous word (or the next word for opening parenthesis). Here we could use BIND as well but this means that the parser would reject sentences where the punctuation is separated. Usually we do not want this and for that purpose we also introduced SOFT_BIND. In generation, BIND and SOFT_BIND are identical, but in parsing, the latter allows optional spaces between the surrounding words.

For completeness we also added SOFT_SPACE – a token which in generation mode leaves space between the surrounding words but in parsing allows the space to be omitted. The three spacing tokens are summarized in Table 1. The rows show how many spaces are inserted in generation and the columns how many spaces are accepted in parsing. Two of the combinations are inconsistent since this would generate sentences that are impossible to parse. One combination corresponds to the default case where ++ is used alone and the rest of the combinations are represented with a special token.

It should be obvious by now how BIND can be implemented in the natural language generator. When we see BIND then we just glue the next word to the previous one. The implementation in the parser is not so obvious. We use a parsing algorithm (Angelov, 2009) which is a variant of Earley (1968) but is generalised to PMCFG.

Earley's algorithm maintains items like:

$$[_i^j \text{ A} \rightarrow \alpha \bullet \beta] \tag{1}$$

which encode the fact that the rule $\text{A} \rightarrow \alpha\beta$ has been partly recognized between positions i and j in the input sentence. The difference in a parser which can handle bindings is that positions i and j must be measured in number of characters rather than number of words. Now if the parser encounters an item like:

$$[_i^i \text{ N} \rightarrow \bullet \text{ "data" BIND "vetenskap"}] \tag{2}$$

and the current word is *datavetenskap* then it generates a new item:

$$[_i^{i+4} \text{ N} \rightarrow \text{"data"} \bullet \text{BIND "vetenskap"}] \tag{3}$$

The items in the original Earley algorithm are grouped in sets with exactly one set for every position between two words. In our implementation there could be from zero to two sets for every character position. Typically there is one set for every position that corresponds to a space between two words, i.e. exactly like in the original algorithm. However, item 3 for example will force a new set to be generated for the position between data and vetenskap. This state is marked in a special way since there is no space at that position. The only tokens that make it possible to exit from this state are BIND, SOFT_BIND and SOFT_SPACE. This is exactly what will happen with item 3 which will lead to the item:

$$[_i^{i+4} \text{ N} \rightarrow \text{"data" BIND} \bullet \text{"vetenskap"}] \tag{4}$$

The new item ends at the same position but now it will be put in a new state which is not marked as special and this will make it possible to accept *vetenskap* as a next token.

Exactly the same modification to the Earley algorithm is also applicable to the PMCFG parser which is the basis of the implementation in GF. Note that unlike Chappelier et al. (1999) and Hall (2005), we do not need a lattice to represent the ambiguous word segmentation. The ambiguity is

naturally represented as different alternatives in the parse chart. However, the advantage of the lattice is that a word is segmented only if all parts are possible parts of the lexicon. In contrast the naive implementation of a parser with `BIND` would segment out a prefix even if the rest of the word is not a possible word. This is easily resolved by using limited lookahead in the parser.

3 Controlling the Capitalization

Compounds in German require both gluing words together as well as altering the capitalization. For example from the nouns *Aktion* and *Plan* we build *Aktionsplan* where the second noun is lower-cased. This means that for every noun we need one form where the first letter is capitalized, and another where it is not. The same applies also to verbs since all verbs can be nominalized. For example from *laufen* 'walk' we get *das Laufen* 'walking' which requires capitalization.

Instead of storing each word form twice in the lexicon, it is advantageous to have a way to dynamically control the capitalization. We can achieve this by introducing one more special token which we call `CAPIT`. The effect of `CAPIT` is that it causes the next word in the sentence to be rendered with initial upper case letter. We store the words in the lexicon in lower case, but by inserting `CAPIT` in the right places in the grammar we guarantee the right capitalization.

In a very simplified form, it could be written like this:

```
fun UseN : N -> CN
lin UseN n = CAPIT ++ n
```

Here `UseN` is a function which converts a noun into a common noun. Just like with `CompoundN`, here, we ignore case, agreement and other irrelevant grammatical features. The noun can be a simple noun as well as a compound (composed by using `CompoundN`), and it is always in lower-case. We add `CAPIT` in front of it to alter the capitalization.

Altering the capitalization is the behaviour of `CAPIT` in generation mode. In parsing we would like to have robust processing, even if the input sentence does not have the correct capitalization. By default the GF parser is case sensitive but by adding in the top module of the corresponding language the declaration:

```
flags
  case_sensitive=off;
```

we can instruct the parser to lower-case the input before parsing. If all words in the grammar are lower-cased too, the parsing becomes case insensitive. This demands also that the parser must simply ignore `CAPIT`. In an Earley style parser, this corresponds to adding the rule:

$$\frac{[_i^j \text{ A} \rightarrow \alpha \bullet \text{CAPIT } \beta]}{[_i^j \text{ A} \rightarrow \alpha \text{ CAPIT} \bullet \beta]} \quad (5)$$

i.e. every time when we encounter an item with a dot before `CAPIT` we are simply allowed to move the dot to the next position.

The requirement that all words in the grammar must be lower-cased contradicts the already established convention for defining lexical entries in German. An entry is defined by applying a smart paradigm function (Détrez and Ranta, 2012) to one or more word forms:

```
mkN "Junge" "Jungen" masculine
```

Here the forms are expected to be the lexicographic forms used in the traditional paper dictionaries. Since the tradition is to use the capitalized forms, the same convention was adopted in the GF grammar as well. The conflict was easily resolved by changing the definition of the paradigm function `mkN`. Because it is computed at compile time, it has more freedom and it internally lower-cases the forms listed in the definition.

German was the primary motivation for adding `CAPIT`, but the robustness with respect to the capitalization is another aspect which is also useful in other languages. One prototypical use is the capitalization of the first word in a sentence. Another example is English which has a number of words (compass directions, country adjectives, etc.) that by convention are spelled with upper case. In all those examples if they are encoded with `CAPIT` then parsing succeeds even if the capitalization is wrong.

The requirement that all lexical entries in the grammar must be in lower-case means that we need special treatment for acronyms which are typically written with capital letters. Using `CAPIT` would be enough, if we split those into a sequence of letters glued with `BIND`. For example IT (information technology) can be encoded as:

```
          CAPIT ++ "i" ++
BIND ++ CAPIT ++ "t"
```

The same trick works even for mixed case combinations like the word LaTeX. However, using the

trick requires a tedious and verbose encoding. It is acceptable for rare combinations like LaTeX, but we want a more compact solution for the normal acronyms. For that purpose we also added `ALL_CAPIT`. The behaviour of `ALL_CAPIT` is that it capitalizes all letters in the next orthographic word instead of only the first one. It is important to emphasize that `ALL_CAPIT` applies to the next orthographic word and not to the next grammatical word. This means that if `ALL_CAPIT` is followed by a sequence of words glued with `BIND` or `SOFT_BIND` then the whole sequence will be capitalized.

Finally we should note that the robustness in the parser also introduces additional spurious ambiguities. For instance the acronym IT becomes indistinguishable from the pronoun *it*. There should be a soft preference for the pronoun, if the word is spelled with lower-case, and alternatively in the other direction if the word is upper-cased. This is easy to implement in a parser which computes weights for alternative analyses. In that case additional weight must be added to every analysis that needs Rule 5 and where the next word in the sentence has the wrong capitalization.

4 Phonetic Dependencies

It is quite common to have words in the language whose exact form depends on the next word. Typical examples are the indefinite article a/an in English, the definite articles le/l', la/l' in French, and the prepositions s/sъs and v/vъv in Bulgarian. Keeping track of the form of the next word in the grammar is tedious and error-prone. It is much easier to delay the choice until we know all words in the sentence. At that phase it is easy to check the next word.

We do this by using the `pre` token[2]. Its general syntax is:

```
pre {default;
     form1 / prefixes1;
     ...
     formN / prefixesN}
```

Here we start with the default form which applies if there is no more specific variant. After the default are the specific cases. Every case is a word form followed by slash and a list of prefixes. If the next word after `pre` starts with one of the prefixes in the list then the final sentence will contain the corresponding word form. In case of ambiguity the first matching alternative is selected.

Both the default and each of the specific forms are arbitrary expressions. They could for instance consist of multiple orthographic words, or they could include other special tokens such as `BIND`. The definite article in French is a good example:

```
pre {"la"; "l'" ++ BIND / voyelle}
```

If the next word starts with a vowel then the article *la* should be contracted to *l'* and the contraction should be attached to the next word. We specify this by using the expression `voyelle` which is defined as the list of all vowels in French. *l'* is followed by `BIND` to indicate that there is no space before the next word. The default is *la* without `BIND` since it is spelled as a separate word.

The above gives the false impression that contraction in French is fully implemented. Unfortunately this is not exactly the case. The problem is that the aspirated h in French does not allow contraction while the non-aspirated h demands it. Unfortunately it is not possible to find whether h is aspirated by just looking at the prefix of the word. For this we need to know the whole word. There is a similar problem in English where in order to choose a/an we need to know whether the next word is pronounced with initial vowel. Unfortunately this is not always possible to infer from the orthographic prefix of the word. For that reason the implementation for phonetic dependencies in English and French is only approximate in the current grammars. In contrast, the prepositions s/sъs and v/vъv in Bulgarian are always reliably detected since the orthography and the phonetics in Bulgarian match very well. A better implementation for English and French would require a grammar that models in parallel both the orthography and the phonetics of the language. In that case `pre` should refer to the phonetic form of the next word rather than to the orthographic form.

French has one more feature which illustrates why orthography should not be implemented outside of the grammar. The problem is that the definite article *le* in combination with the preposition *de* is contracted to *du*, i.e.:

Le livre du garçon

instead of:

Le livre de le garçon

At the same time *le* is also the pronoun *it* which

[2]Strictly speaking `pre` is not a new addition to GF, but its operational behaviour was never precisely defined. It also interacts with the orthographic extensions, so it deserves attention in the current paper.

does not allow the same contraction here:

Il m'a dit de le faire

This is easy to implement in the grammar by using the syntactic context, but it is difficult to resolve in a post processor working only with the surface form of the final sentence.

`pre` is also used in the generation of punctuation. The problem is that in many languages non-restrictive relative clauses are separated from the rest of the sentence with commas before and after the clause. However, the closing comma should be used only if there is no other punctuation mark and if the comma is not the last word in the sentence. The right way to encode this is:

```
pre {"";
     "" / punct;
     SOFT_BIND ++ "," / ""}
```

Here the first special case uses the expression `punct` which lists all punctuation symbols. This means that if the word after the comma is punctuation, then we just generate the empty string. The second case uses empty string as a prefix for filtering. By definition, an empty prefix matches only if there is at least one word after `pre`. Obviously the empty string is a valid prefix for every possible word. But, if there is no following word then only the default form is applicable. We used the empty string as the default since this is what we want to generate if we are at the end of the sentence. Finally, the use of `SOFT_BIND` in the expression ensures that comma is glued to the preceding word in the sentence.

In generation mode, we always propagate the `pre` until we know all words in the sentence and then we replace it with the correct alternative. In parsing we decided that we want to accept all alternatives regardless of whether they agree with the constraints. Usually this does not lead to false ambiguities and accepting more alternatives only makes the parser more robust.

5 Degenerate Inflection Tables

Another issue, that is not really related to orthography, but is also easily solved by adding special kinds of tokens in the grammar, is when a word has missing inflection form. The problem is that every lexical entry in GF is not a single word but an inflection table with all possible forms. The grammarian defines the entry by applying a smart paradigm function, which is computed to an inflection table. For example if we define:

```
lin apple_N = mkN "apple"
```

the result will be something similar to:

```
lin apple_N =
      table {Sg => "apple";
             Pl => "apples"}
```

This representation fails for degenerate words that miss one or more forms. For that purpose we introduced the token `nonExist`. It can be used in any place where a word should be put but the actual language does not have the appropriate form.

In generation mode, `nonExist` behaves like an exception. Any attempt to linearise a sentence which contains `nonExist` will fail completely and no output will be generated. Ideally the grammarian should avoid exceptional situations and write grammars that always produce sensible output. At least this should be the goal for expressions generated from the intended start category, i.e. for sentences. Avoiding the exceptions is usually possible by using parameters to control the possible choices in the grammar. `nonExist` is still useful as an explicit marker for impossible combinations in nested categories, i.e. categories that are not used as start categories. If hiding all exceptions in the grammar is not possible then at least by using `nonExist` the user gets a chance to handle the situation by rephrasing the request.

`nonExist` interacts in an interesting way with variants in the grammar. GF provides the bar (`|`) operator for listing alternative linearisations. For example `a | b` is an expression which lists the two expressions `a` and `b` as two different alternatives. The API to the GF runtime provides methods for computing all possible alternatives for one and the same expression. In that case if one of the alternatives includes `nonExist` then it is simply filtered out. Using variants in combination with `nonExist` is yet another way to hide exceptions.

In parsing mode `nonExist` is implemented by simply stating that items like this:

$$[_i^j \, \mathrm{A} \rightarrow \alpha \bullet \texttt{nonExist} \; \beta]$$

should not lead to any further derivations. A hacked-up version of `nonExist` can be implemented by defining it as a special word which is usually not used in the target language. In this way we can hope that the special word will never appear in an actual sentence and this will give us the intended behaviour. However, this is first of all inelegant and second it will become visible from some of the API calls. For example GF

is commonly used for designing controlled languages where the parser is used inside authoring tools to assist the user in writing valid phrases in the language. In that case using the hack will cause the special token to show up in the suggestions from the authoring tool. Instead we implemented `nonExist` as a parser internal operation which is always invisible from the API calls.

6 Conclusion and Future Work

We presented a number of extensions in GF that make it easier to model orthographic phenomena without the need to use external pre- and post-processors. Keeping all the grammatical knowledge in a single framework is a definite engineering advantage that gave us knowledge sharing and portability. All of the extensions were extensively tested in practice as part of the translation system demonstrated in Angelov et al. (2014).

In the design we assumed two conflicting requirements: high-precision in generation and robustness in parsing. This sounds like a logical choice for most applications. It is possible, however, that for some applications, a more strict parser will be desirable, too. For example in an application for checking the correctness of a sentence we may want to be more strict. There are two ways in which this can be achieved. First of all it is always possible to use the robust parser for parsing the original input. After that the analysis can be fed back to the generator which will produce the canonical linearisation. The input can be compared with the canonical linearisation to identify potential problems. Another more involved solution is that we can extend the parser to keep record of all cases where some constraint has been violated. In that way after the parsing is finished, the user could consult the list of cases.

There is one more issue for which we have not made a firm decision yet. We have already mentioned several times the ++ operator which puts two phrases together. The framework has also an operator + which is similar but does not insert a space between the phrases. Its behaviour is similar to the combination `++ BIND ++` except that it is computed at compile time rather than at runtime. Currently its primary use is to compute new inflection forms in the definitions for different morphological operations. In principle, we could extend its use to a runtime operator which will further fuse the boundary between grammatical and orthographic words. Whether or not this is a good design decision is still not clear to us. In any case extending the domain of + would not make the use of `BIND` obsolete since `BIND` can also interact in a non trivial way with `pre` as in the French example. There are also similar examples in Catalan and Maltese.

Acknowledgement

Thanks to Aarne Ranta and the annonymous reviewers for their useful comments on the earlier draft of the paper. Swedish Research Council (Vetenskapsrådet) has supported this work under grant number 2012-4506.

References

Krasimir Angelov, Björn Bringert, and Aarne Ranta. 2010. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, 19:201–228.

Krasimir Angelov, Aarne Ranta, and Björn Bringert. 2014. Speech-enabled hybrid multilingual translation for mobile devices. In *European Chapter of the Association for Computational Linguistics*, Gothenburg.

Krasimir Angelov. 2009. Incremental parsing with parallel multiple context-free grammars. In *European Chapter of the Association for Computational Linguistics*.

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

Pierre Boullier. 1998. A proposal for a natural language processing syntactic backbone. Technical Report 3342, INRIA.

John J. Camilleri. 2013. A Computational Grammar and Lexicon for Maltese. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, September.

J.-C. Chappelier, M. Rajman, R. Arages, and A. Rozenknop. 1999. Lattice parsing for speech recognition. In *In Proceedings of 6me*, pages 95–104.

Ciprian Chelba and Alex Acero, 2004. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, chapter Adaptation of Maximum Entropy Capitalizer: Little Data Can Help a Lo.

Grégoire Détrez and Aarne Ranta. 2012. Smart paradigms and the predictability and complexity of inflectional morphology. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '12,

pages 645–653, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jay Clark Earley. 1968. *An Efficient Context-free Parsing Algorithm*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA. AAI6907901.

Keith B. Hall. 2005. *Best-first Word-lattice Parsing: Techniques for Integrated Syntactic Language Modeling*. Ph.D. thesis, Providence, RI, USA. AAI3174615.

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *J. ACM*, 40:143–184, January.

Aravind Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York.

Philipp Koehn and Kevin Knight. 2003. Empirical methods for compound splitting. In *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1*, EACL '03, pages 187–193, Stroudsburg, PA, USA. Association for Computational Linguistics.

Inari Listenmaa and Kaarel Kaljurand. 2014. Computational estonian grammar in grammatical framework. In *Proceedings of LREC 2014*, pages 13–18.

Lucian Vlad Lita, Abe Ittycheriah, Salim Roukos, and Nanda Kambhatla. 2003. truecasing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 152–159, Stroudsburg, PA, USA. Association for Computational Linguistics.

Peter Ljunglöf. 2004. *Expressivity and Complexity of the Grammatical Framework*. Ph.D. thesis, Department of Computer Science, Gothenburg University and Chalmers University of Technology, November.

Carl Jess Pollard and Ivan A. Sag. 1994. *Head-driven phrase structure grammar*. Studies in contemporary linguistics. Center for the study of language and information Chicago (Ill.) London, Stanford (Calif.).

Aarne Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October.

Mark Steedman. 2000. *The syntactic process*. MIT Press, Cambridge, MA, USA.