Rule-Based Language Technology
2023
NEALT Monograph Series, 2

Grammatical Framework: from Interlingual Semantics to Morphophonemic Processes[1]

Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
412 96 Göteborg
Sweden
`aarne.ranta@cse.gu.se`

Inari Listenmaa
Digital Grammars AB
Gothenburg, Sweden
`inari@digitalgrammars.com`

Abstract

Grammatical Framework (GF) is a programming language and a grammar formalism. It is designed to cover all levels of rule-based language processing, from semantics via syntax to the smallest details of morphology. Its most well-known applications are interlingual machine translation and multilingual text generation, but it is being increasingly used for creating general-purpose language-processing components, both for the major languages of Europe and the World and for languages with little or no earlier resources. In this paper, we will focus on the practical use of GF for rule-based description of linguistic structures, ranging from semantics to morphophonemics.

Key Words: grammar formalism, multilinguality

1 Introduction

Grammatical Framework (GF, Ranta, 2011a) is a programming language and a grammar formalism. It was designed to cover all levels of rule-based language processing,

[1]**Ref**: Ranta, Aarne and Inari Listenmaa 2023. Grammatical Framework: from Interlingual Semantics to Morphophonemic Processes. In: Arvi Hurskainen, Kimmo Koskenniemi, and Tommi Pirinen (eds.), Rule-Based Language Technology. NEALT Monograph Series, 2:9–48. `https://dspace.ut.ee/handle/10062/89595`

from logical semantics via syntax to the smallest details of morphology. The most distinct feature of GF is its **multilinguality**: it provides constructs that enable common representations and code sharing for sets of languages.

The most characteristic applications of GF are **interlingual translation** and **multilingual text generation**, both supported by shared semantic structures. But GF has also been used for creating **linguistic resources**, originally to support translation and generation, but over the time developed into large-scale libraries usable for many different language processing tasks. These resources cover over 50 languages and are constantly developed further by a world-wide community and freely usable for both academic and commercial purposes.

In this chapter, we will focus on the practical use of GF for the rule-based description of linguistic structures, ranging from semantics to morphophonemics. In recent years, GF has also been used as a component of hybrid systems involving statistics and machine learning. In this chapter, however, will not cover so much of these aspects, but refer to a survey in Ranta et al. (2020).

This chapter aims to be self-contained and only assume general knowledge of programming languages and linguistic concepts. But it is neither a complete description nor a tutorial; for these, we refer to the GF book (Ranta, 2011a) and the GF homepage[1]. The purpose here is to present a variety of ideas about where GF might be useful for the working computational linguist.

The paper is structured as follows: Section 2 outlines the history of GF. Section 3 explains the notions of abstract and concrete syntax, which are fundamental for GF. Section 4 adds morphological features and agreement. Section 5 explains the type system and static type checking of GF. Section 6 is a lengthy case study of morphological paradigms. Section 7 is about the GF Resource Grammar Library. Section 8 contains a case study of the complex adposition rules in Somali. Section 9 gives examples of semantics in GF. Section 10 concludes.

2 A short history of GF

GF was first released in 1998 at Xerox Research Centre Europe (XRCE) to support **Multilingual Document Authoring** (MDA), with the slogan "write a document in one language and see it develop in other languages at the same time" (Dymetman et al., 2000). The technical basis of MDA was an **interlingual semantic representation**, which had automatic **linearizations** to different languages. This idea had been previously demonstrated in the WYSIWYM ("What You See Is What You Mean", Power and Scott, 1998). What GF was expected to generalize this by using the full power of **constructive type theory** (Martin-Löf, 1984; Ranta, 1994). GF should also contain a **declarative notation** to enable the rapid development of new WYSIWYM-like systems: most of the system components, such as the parser, generator, and user in-

[1] `http://grammaticalframework.org`

terface, should be reusable in different systems by plugging in a declaratively defined **multilingual grammar**.

Fig. 1 gives an example of an interlingual representation and some languages related to it, together with word alignments that show how one and the same representation can relate to different surface strings. The example is from a commercial project carried out in 2017–18, targeting the legal terminology of the General Data Projection Regulation (GDPR) of the European Union[2].

The technical ideas of GF were inherited from two different directions. One was **Logical Frameworks** (LF, Harper et al., 1993), which were designed to implement logical calculi (such as intuitionistic, classical, modal, higher-order) in a declarative way. A part of GF is thus inherited from ALF (Another Logical Framework, Magnusson, 1994): this is the part used for defining the semantic representations. Another direction was **grammar formalisms**, of which there were several in active use at the time: HPSG (Pollard and Sag, 1994), LFG (Bresnan, 1982), TAG (Joshi, 1985), DCG (Pereira and Shieber, 1987), and CCG (Steedman, 2000). Most of those formalism supported a separation between **constituency** and **morphological features**, which also became a backbone of GF, as the separation made it possible to use common representations for languages with different systems of features. The use of features was to be described in GF itself, rather than in a separate morphological component, and this part of GF was greatly inspired by the Xerox Finite State Tool (XFST, Beesley and Karttunen, 2003).

Unlike the mainstream grammar formalisms of the 1990s, processing in GF was not based on **unification** but on generation-oriented **linearization**. This set-up also encouraged the development of **abstraction mechanisms** familiar from **functional programming languages**, such as higher-order functions, algebraic datatypes, type-driven overloading, and parameterized modules. The second version of GF (Ranta, 2004) adopted a notation inspired by Haskell[3], although using C-like braces and semicolons instead of layout syntax. Also a module system was developed (Ranta, 2007), inspired by object-oriented languages with multiple inheritance, as well as the parameterized modules of ML (Milner et al., 1990).

Parallel to the development of the GF programming language, the theoretical properties of GF were investigated, establishing an equivalence with **Parallel Multiple Context Free Grammars** (PMCFG, Seki et al., 1991) and hence polynomial, mildly context-sensitive parsing complexity (Ljunglöf, 2004). Generation had linear complexity by design. New algorithms enabled GF parsing to scale up from fragments of language to full-scale text and also permitted the statistical ranking of parse trees (Angelov and Ljunglöf, 2014).

After the first times at Xerox, GF was used mainly by computer scientists and mathematicians, rather than linguists. Its ambition was to make grammar writing accessible

[2]The official documents analysed were from `https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679`. The syntax tree is based on the GF grammar of Attempto Controlled English Fuchs et al. (2008) in Angelov and Ranta (2009). A demo of the GDPR project is available in `https://gdprlexicon.com/`.

[3]`https://haskell.org`

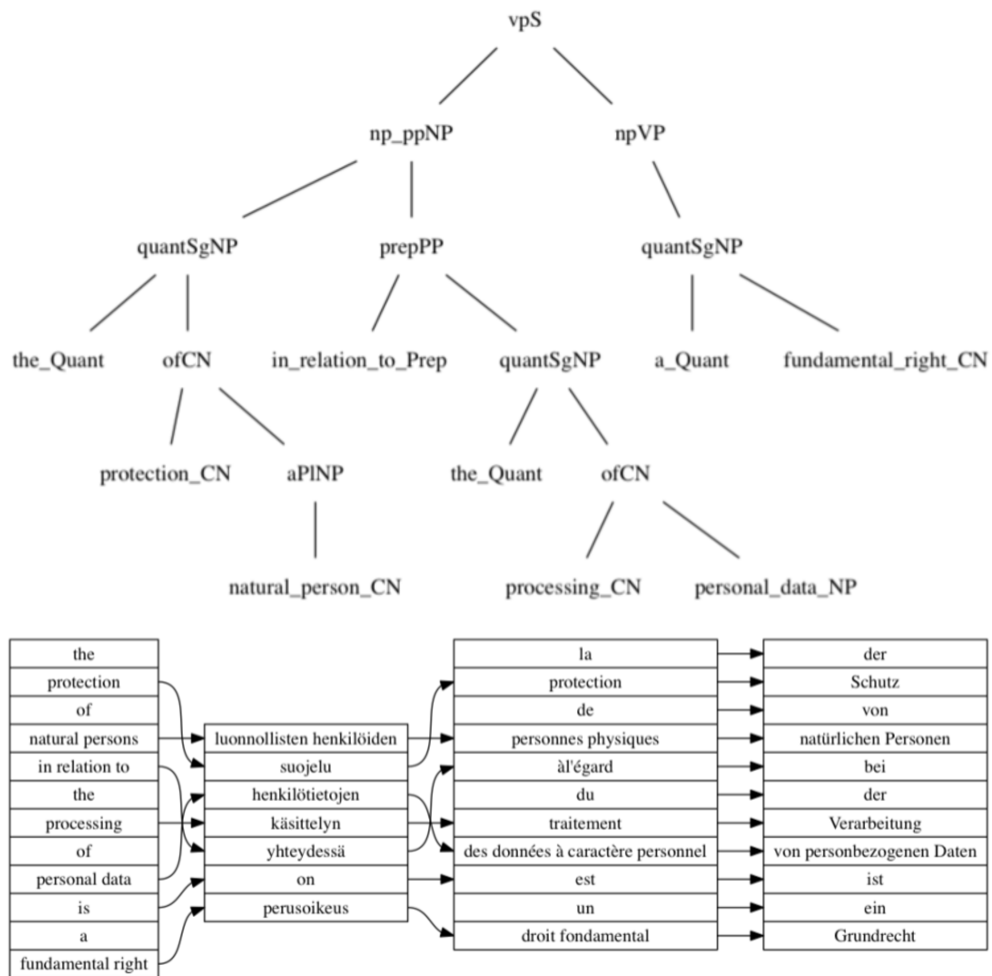

Figure 1: An abstract syntax tree in GF for the first sentence of the GDPR legislation of the European Union, with word-aligned renderings in English, Finnish, French, and German.

to programmers with non-linguist backgrounds, both by a familiar-looking programming language and by the general set-up, where grammar writing was seen as something similar to **compiler construction**. This was also manifested by the **BNF Converter** (BNFC)[4], which is a compiler tool that emerged as a spin-off from GF and can be combined with it when, for instance, developing natural language interfaces to formal notations.

As grammars are complex software, programming language support is useful when developing them. But it does not yet solve the problem of **linguistic knowledge**: programmers that implement for instance natural language interfaces to software modelling languages (Burke and Johannisson, 2005) may need to master a lot of morphology and syntax even for limited fragments of language. This need led to the most substantial set of grammars developed in GF, the **Resource Grammar Library** (RGL, Ranta, 2009), which, by the time of writing, has been implemented for over 50 languages. About half of these languages are equipped with a large lexicon based on the **WordNet** (Fellbaum, 1998; Angelov, 2020)[5].

The functional programming abstractions of GF enable the use of the RGL via a high-level API (Application Programming Interface) which, among other things, hides all information about morphological features and agreement. The idea is similar to software libraries for e.g. graphics: their user can specify forms such as circles and rectangles without caring about how they are rendered in pixels.

The RGL was a response to practical needs, but also inspired by theoretical linguistic considerations: it was interesting to see how GF could combine low-level linguistic rules in its high-level multilingual setting, and what this would say about the relations between different languages. In this respect, the RGL is similar to the LiNGO Matrix written in HPSG (Bender et al., 2010) and the ParGram in LFG (Butt et al., 2002). But it goes beyond these sets of grammars by actually using a common tree structure — **abstract syntax** — for all of the languages.

A later approach based on common structures is Universal Dependencies (UD, Nivre et al., 2016), which uses a common set of dependency relations for different languages. Soon after the release of UD, it turned out that these relations are quite compatible with the RGL of GF (Kolachina and Ranta, 2016). The similarities have been used in various ways in systems combining UD and GF: a common feature is that, while UD is more powerful and robust in parsing, GF is more accurate in generation (Ranta, 2017).

While the RGL and many applications of GF are academic or otherwise open-source software built by a community, later years have also seen commercial applications, which are typically focused on **Natural Language Generation** (NLG). They build on the original vision of MDA at Xerox: generate several languages from a single abstract source. An ultimate challenge for multilingual NLG is **Abstract Wikipedia**, whose mission is to generate encyclopaedic articles in the over 300 languages of Wikipedia (Vrandečić, 2021). GF was initially seen as a proof of concept that such a system is

[4] `http://bnfc.digitalgrammars.com/`
[5] For an up-to-date view, see `http://cloud.grammaticalframework.org/wordnet/`

theoretically possible. But an actual collaboration between the Wikimedia and GF communities has started, trying to show actual results and see how far one can go (Ranta, 2022). In a nutshell, the challenge is to scale up GF applications by two orders of magnitude: ten times more languages, ten times richer content than before.

3 Abstract and concrete syntax

Let us start with the “hello world” example of syntax, the classical rule building a sentence (S) from an noun phrase (NP) and a verb phrase (VP). In GF, one can write this rule in the context-free BNF (Backus-Naur) notation,

```
S ::= NP VP
```

But this notation does not support multilinguality. First, it assumes a certain word order, either SVO (subject-verb-object) or SOV, depending on what the rule for VP is. Secondly, it does not specify the agreement between the subject and the verb, which can work in different ways in different languages.

The GF solution is to split the grammar into two parts. The first part is an **abstract syntax**, which defines **construction functions** that build **abstract syntax trees**. For the NP-VP rule, we define the function `PredVP` (for **predication**), introduced by the keyword `fun` and declaring the **type** of the function,

```
fun PredVP : NP -> VP -> S
```

Following the notation of typed functional programming languages such as Haskell, function types are denoted by an arrow notation, where the last category (here, S) is the **value type** of the function and the previous ones are the **argument types** (here, NP and VP).

The second part is a **concrete syntax**, which defines **linearization functions** that convert abstract syntax trees to **strings** and other surface representations. Linearization functions are introduced by the keyword `lin`, followed by the function name and variables matching each of its arguments. For `PredVP`, the SVO (or SOV) rule without agreement is

```
lin PredVP np vp = np ++ vp
```

where the operator `++` is used for **concatenation**.

Now, the above linearization of `PredVP` works for SVO and SOV languages, if we ignore agreement for the moment. The difference between these types comes from the **complementation function** that combines transitive verbs (V2) with their complements,

```
fun ComplV2 : V2 -> NP -> VP
```

The linearization rules for SVO and SOV are

```
lin ComplV2 v2 np = v2 ++ np
lin ComplV2 v2 np = np ++ v2
```

respectively. These rules belong to two different concrete syntaxes of the same abstract syntax, such as English using SVO and Latin using SOV. But we will see in a moment that they can also encode variations within one and the same language.

Now, how can we describe the VSO order, which is found e.g. in Arabic? The problem is that the verb phrase contains both the verb and the object. When the subject is added, the verb phrase must be broken into two parts. This cannot be done with just string concatenation (or a context-free grammar), which would mean that the Arabic grammar needs to have a constituent structure different from the English grammar. In GF, however, the target of linearization is not restricted to strings. It can also be a **record**, a data structure consisting of several **fields**. Thus the linearization of a verb phrase can produce a record with separate fields for the verb and the object:

```
lin ComplV2 v2 np = {verb = v2 ; obj = np}
```

When the subject is added, these fields can be obtained as **projections** from the record:

```
lin PredVP np vp = vp.verb ++ np ++ vp.obj
```

producing the VSO word order. Projections are denoted by dot notation (`vp.verb`, `vp.obj`).

3.1 How to test it: modules, files, and the GF shell

GF grammars can be tested in the **GF shell**, which can be downloaded from the GF web page; ready-made binaries are available for the main operating systems. In order to use the grammars, they must be stored in files, each of which contains a **module**. The above rules need to be completed with some other ones to enable well-formed module. To start with, a minimal module for an abstract syntax is

```
abstract Test = {
  cat
    S ; NP ; VP ; V2 ;
  fun
    PredVP : NP -> VP -> S ;
    ComplV2 : V2 -> NP -> VP ;
    John, Mary : NP ;
    Love : V2 ;
  }
```

The functions `PredVP` and `ComplVP` are familiar, but the rest are new:

- The first line is the **module header**, which declares `Test` as the name of an abstract syntax module. The module must be stored in a file named `Test.gf` so that GF can recognize it.
- The first two lines in the **module body** (in braces) are **category rules** that declare a set of categories. The keyword `cat` is common for these categories; it could also be repeated for each of them, but this is not necessary. Every rule is terminated by a semicolon.
- The remaining lines are function rules, with the common keyword `fun`. Semicolons are needed also here, although we did not show them earlier.
- The last three rules (two declaring an NP, one a V2) are added to enable actual sentences to be formed. They are examples of **lexical rules**, whose abstract syntax functions take no arguments.

An abstract syntax, once defined, can have one or more concrete syntaxes declared to be *of* that module. Thus we have

```
concrete TestEng of Test = {
  lincat
    S, NP, VP, V2 = Str ;
  lin
    PredVP np vp = np ++ vp ;
    ComplV2 v2 np = v2 + np ;
    John = "John" ;
    Mary = "Mary" ;
    Love = "loves" ;
  }
```

What we have here is

- for each `cat`, a `lincat` definition giving the **linearization type** of the category, here uniformly Str (“string”).

Once we have saved this module in a file named `TestEng.gf`, we can start the GF shell with the command `gf` and **import** the module with the `import` command:

```
$ gf
-- welcome message appears
> import TestEng.gf
```

The command also imports all other modules that are needed, in this case, `Test.gf`. After this, one can start using the grammar for linearization, parsing, random generation, and their pipes (with the notation | as in Unix):

```
> linearize PredVP Mary (ComplV2 Love John)
Mary loves John

> parse "Mary loves Mary"
PredVP Mary (ComplV2 Love Mary)

> generate_random
PredVP Mary (ComplV2 John)

> generate_random | linearize
John loves John
```

A good exercise after this is to write another concrete syntax, with different words and possibly different word order. It can be imported in the same GF shell session, after which one can test **translation** by piping parsing to linearization, for example,

```
> parse -lang=Eng "John loves Mary" | linearize -lang=Fre
Jean aime Marie
```

4 Features and agreement

Verb phrases as shown above are an example of **discontinuous constituents**, which in GF can be implemented as records. Another use of records is to store **inherent features**, such as the genders of nouns and other agreement features of noun phrases. For example, the record

```
{s = "my children" ; agr = {n = Pl ; p = P3}}
```

represents a noun phrase with the string *my children* and the agreement feature — itself a record — with number plural (Pl) and person third (P3). Records of this kind are similar to **feature structures** in **unification grammars** (Shieber, 1986), but the way they are used in GF is a bit different, because of its perspective on generation rather than parsing.

In NP-VP predication, we will need to make sure that the agreement features of the verb phrase match those of the noun phrase. However, the features are attached in a different way to the VP. They are not inherent like in the NP, as they are **variable features**. This means that one and the same VP can appear in different **forms** (first person singular, third person plural, etc), unlike NPs, which cannot (with the exception of personal pronouns) produce different forme for different agreement features.

Variable features are not represented as records, but as **tables**, which is GF's formalization of **inflection tables**. For example, the verb phrase *be tired* could be represented as the following table:

```
table {
  {n = Sg ; p = P1} => "am tired" ;
  {n = Sg ; p = P3} => "is tired" ;
  _  => "are tired"
  }
```

While a record consists of fields, a table consists of **branches**, where the left-hand sides are **patterns** matching different values, here agreement features. Tables can be made compact by using **wild-card patterns** _, which in the above example matches all the features not matched by previous patterns — that is, the second person singular and all plural features. The pattern matching syntax is adapted from functional programming (e.g. Haskell and ML) and has some more constructs that will be explained later.

What happens in agreement is, in its simplest form, that the inherent features of one constituent are passed as variable features to another. Thus, assuming as above that NP has agreement as an inherent feature and VP as a variable feature, we can write the linearization rule for predication as

```
lin PredVP np vp = np.s ++ vp ! np.agr
```

The exclamation mark (`!`) denotes **selection** from a table. Thus `vp ! np.agr` selects the form of the `vp` corresponding to the agreement features of the subject (`np.agr`).

This generalizes to all other word orders, for example, VSO:

```
lin PredVP np vp = vp.verb ! np.agr ++ np.s ++ vp.obj
lin ComplV2 v2 np = {verb = v2 ; obj = np.s}
```

where we assume, for simplicity, that a V2 has the same agreement features as a VP, and that the object has no variable features. (This assumption is too simple when we widen the grammar to cover more structures and introduce for instance reflexive pronouns as objects.)

Features are not only morphological, but they can affect syntax as well. An example is German word order, which in fact displays all of the three word order patterns discussed above: SVO in main clauses (*ich liebe dich*, “I love you”), SOV in subordninate clauses (*(dass) ich dich liebe*), and VSO in inverted clauses such as questions and after adverbials (*liebe ich dich*). Despite this variation, we can maintain the same abstract syntax as done above, by making the word order (actually, constituent order) into a variable feature of S in German. The predication function is then linearized

```
lin PredVP np vp = table {
  SVO => np.s ++ vp.verb ! np.agr ++ vp.obj ;
  SOV => np.s ++ vp.obj ++ vp.verb ! np.agr ;
  VSO => vp.verb ! np.agr ++ np.s ++ vp.obj
  }
```

5 The type system

The interplay of variable and inherent features and discontinuous constituents is a complex system, which also explains much of the coherence in a language: why does a NP have inherent agreement features? Well, that's because the VP has them as variable features and needs to get them from somewhere in order to decide what strings to produce. Different languages have different sets of features, which therefore belong to the concrete syntax, at the same time as a common abstract syntax is maintained.

How features attach to different categories depends on the language and belongs hence to the concrete syntax. For example, in French, unlike in English, nouns have inherent genders, which are passed to adjectives that have variable genders. French has just two genders unlike German, which has three, and Slavic languages that can be seen as having four, as the masculine gender if further divided into animate and inanimate forms, which affect agreement in the same way as the "main" genders.

As we saw in Section 3.1, a concrete syntax must contain a `lin` rule for each `fun` rule in an abstract syntax to define linearization functions, as well as a `lincat` rule for each `cat` rule, to define the **linearization types** of categories. The simplest linearization type is `Str`, **strings** (more accurately, **token lists**). A hierarchy of more and more complex types can be built with **record types** `{a : A ; b : B}` and **table types** `A => B`, where `A` and `B` are in both cases types.

Here is an example, specifying the type system for the German rules as described above:

```
lincat S = Order => Str
lincat NP = {s : Str ; a : {n : Number ; p : Person}}
lincat VP = {verb : {n : Number ; p : Person} => Str ; obj : Str}
```

But where do the types Order, Number, and Person come from? They are defined by using yet another form of rules, `param` for **parameter types**:

```
param Order = SVO | SOV | VSO
param Number = Sg | Pl
param Person = P1 | P2 | P3
```

(The way these features are used is shown by the `PredVP` rule at the end of the previous section.)

The syntax for parameter types comes from the **algebraic datatypes** of functional programming (e.g. ML and Haskell). In this first example they are simply **enumeration types**, but will be later generalized to hierarchic types with more structure. Parameter types belong to concrete syntax modules, not to abstract syntax.

Linearization types are a good way to document a language by giving its type signature. But they also play an important computational role: they are used for the **static type checking** of grammars. If an abstract syntax function has certain argument and value types, its linearization function must operate on the corresponding linearization types. Formally, given any function in abstract syntax

$$f \ : \ C_1 \rightarrow \cdots \rightarrow C_n \rightarrow C$$

its linearization function is

$$f^* \ : \ C_1^* \rightarrow \cdots \rightarrow C_n^* \rightarrow C^*$$

where both the linearization types and the linearization function are marked with an asterisk. The linearization of a tree is computed

$$(f a_1 \ldots a_n)^* \ = \ f^* a_1^* \ldots a_n^*$$

Thus linearization is **compositional**, in the sense that the linearization of a tree is a function of the linearizations of its subtrees. In mathematical terms, an abstract syntax is a **free algebra** and its concrete syntaxes are **homomorphisms** from it to strings and other objects defined by linearization types.

Static type checking means, as in other typed programming languages, that the **compiler** checks that all rules obey the specified types. This means typically in practice that the programmer has to fix a number of **type errors** before she can run the program. But it also means that the program will not fail at runtime: the type checking of linearization rules with respect to the linarization types guarantees that all combinations of features that are required actually exist for all phrases of any given category. This can be an uncertain assumption in languages like Python that do not have static type checking.

In addition to controlling correctness, static typing is used to guide the compiler to produce optimal **machine code**. In the case of GF, this means that grammars are converted into binary code representing low-level PMCFG; the file format is called **Portable Grammar Format**, PGF (Angelov et al., 2009; Angelov, 2011). PGF is portable in the sense that PGF grammars can be used independently of the full GF system from different programming languages, such as C, Java, and Python.

Another use for static typing is **overloading resolution**, which means that different functions can be given the same name as long as they have different types. This is an important technique when building libraries (both in GF and in e.g. Java and C++), and we will provide examples of it later, in Section 7.2.

6 Morphology and lexicon

The examples shown above are from the syntax of natural languages, in the traditional sense where syntax trees model phrases that are combined into larger phrases. The leaves of the trees are **lexical items**. In GF, lexical items are represented by abstract syntax functions (`fun`) with no arguments. Such functions are often, but not always, linearized to single tokens, **words**. In the multilingual setting, however, this need not always be the case. On the contrary, an abstract syntax lexical item is often meant to represent a **word sense**, which some languages express by single words, some others by **multiword expressions**, or, as a limiting case, with no words at all.

6.1 Multiword items

An example of word senses represented by multiwords is the English word *capital* in the sense of the capital city of a country. In Croatian, the concept needs two words, an adjective and a noun: *glavni grad* ("main city"). When the term is inflected for number and case, both parts agree: *glavni grad*, *glavnog grada*, *glavnom gradu*, and so on. Such terms are defined by combining syntactic and morphological functions.

6.2 Empty words

An example of word senses represented by empty strings are pronouns in **pro-drop languages**. They are linearizations of noun phrases such as `we_NP`, which in English is always represented by a word (*we* in the subject case, *us* in other uses). In Italian, which is a pro-drop language, an unstressed subject "we" is dropped, whereas the other cases (stressed *noi* and clitic object *ci*) are ordinary words. Even in the pro-drop case, it is important that the agreement features are preserved. A simple way to implement this is to use the record

```
lin we_NP = {
  s = table {
    Unstressed => "" ;
    Stressed => "noi" ;
    Clitic => "ci"
    } ;
  a = {n = Sg ; p = P1}
  }
```

When this noun phrase is used as the subject, using the familiar rule

```
lin PredVP np vp = np.s ! Unstressed ++ vp ! np.a
```

the subject is not visible as a string, but its agreement features are passed to the verb phrase, resulting in e.g. *siamo stanchi* for "(we) are tired".

Multiwords and empty words play an essential part in understanding how GF works in an interlingual setting. But the *largest* part of the description of a language is its **morphological lexicon**, which describes the words of the language together with their parts of speech (i.e. GF categories), forms produced by their variable features, and their inherent features. Even if meant to be used in interlingual applications, it is beneficial to build such lexica in a monolingual way. Thus for instance a Serbo-Croatian lexicon should not have an item for the noun "capital", but just two separate items, the adjective "glavni" and the noun "grad". They can later be put together with the adjectival modification function, if the concept "capital" needs to be expressed:

```
lin capital_CN = AdjModCN glavni_A grad_N
```

6.3 From inflection tables to type systems

So why not look at in detail how adjectives are formed in Serbo-Croatian. If you consult a traditional grammar or a morphological lexicon such as Wiktionary, you will find a huge table, such as this one[6]:

deklinacija pridjeva *glavni* [sakrij ^]

jednina		muški rod	ženski rod	srednji rod
nominativ		glavni	glavna	glavno
genitiv		glavnog(a)	glavne	glavnog(a)
dativ		glavnom(u/e)	glavnoj	glavnom(u/e)
akuzativ	neživo	glavni	glavnu	glavno
	živo	glavnog(a)		
vokativ		glavni	glavna	glavno
lokativ		glavnom(e/u)	glavnoj	glavnom(e/u)
instrumental		glavnim	glavnom	glavnim
množina		muški rod	ženski rod	srednji rod
nominativ		glavni	glavne	glavna
genitiv		glavnih	glavnih	glavnih
dativ		glavnim(a)	glavnim(a)	glavnim(a)
akuzativ		glavne	glavne	glavna
vokativ		glavni	glavne	glavna
lokativ		glavnim(a)	glavnim(a)	glavnim(a)
instrumental		glavnim(a)	glavnim(a)	glavnim(a)

This table defines the 56 combinations of 2 numbers, 4 genders, and 7 cases. It shares, however, the animate and inanimate forms for all cases except the accusative singular masculine. This is a valid generalization, which shrinks the number of forms

[6]`https://sh.wiktionary.org/wiki/glavni`. We will in what follows refer to the language as Serbo-Croatian, but show the examples in the Latin alphabet of Croatian rather than the Cyrillic alphabet of Serbian. Our main source, Vrabec (2022), includes Bosnian and Montenegrin in the same family of languages.

into 43.

However, a closer inspection tells that there are only 12 different strings in the table. This observation is confirmed by grammar books, which tell us that, for instance, the genitive singular feminine (*glavne*) is also used for nominative, accusative, and vocative plural feminines as well as accusative plural masculines (Vrabec, 2022). (We ignore the alternative endings given in parentheses, which, according to Vrabec (2022), belong to an older literary style.)

What we now will do is build a type system and an inflection function that describe the details accurately. We start with the traditional parameter types for Serbo-Croatian:

```
param
  Number = Sg | Pl ;
  Animacy = Anim | Inanim ;
  Gender = Masc Animacy | Fem | Neutr ;
  Case = Nom | Gen | Dat | Acc | Voc | Loc | Ins ;
```

This definition shows that Gender as a hierarchic datatype: its first constructor `Masc` takes an Animacy as its argument and thereby produces two values instead of one.

An **adjectival phrase** (AP) is natural to describe in the traditional way, as depending on the three parameters Number, Gender, and Case:

```
lincat AP = {s : Gender => Number => Case => Str}
```

When defined in this way, adjectival phrases are easy to be combined with nouns and other elements that they agree with. Notice that we wrap the table in a record, which is a good practice for linearization types, because we may later notice that we also need some inherent features or that the phrase needs to be made discontinuous.

But for **adjectives** as lexical units, tables with 56 forms would be very redundant, as there are ever only 12 different forms. In a full description, adjectives can also have tables of the similar size for comparative and superlative forms, thereby 168 forms — of which at most 36 are unique.

The obvious solution is to introduce a different type for adjectives of the category `A`, that is, lexical adjectives. For this type, we define a new parameter type with just 12 distinct values, and a table type over it. We give the table type a name by using yet another — the last to be shown — form of rule, **auxiliary operations** with the keyword `oper`:

```
param AForm =
    msnom | fsnom | nsnom | msgen | fsgen | msdat
  | fsdat | fsacc | msins | fsins | mpnom | mpgen

oper AdjForms : Type = AForm => Str
```

`oper` rules can define names for types or functions or anything in the concrete syntax. They are different from `fun` functions in the abstract syntax, which can only define new forms of abstract syntax trees and have `cat` categories as argument and function types.

The use of `oper` definitions is a key technique for structuring and modularizing concrete syntax code, in the same way as functions are used in ordinary programming languages. Notice that `oper` definitions also include type signatures, which brings them under static type checking and are also good devices for documentation. In the definition of `AdjForms`, the type is `Type`, since `AdjForms` defines a new type.

The second `oper` to show is one that converts the non-redundant adjective forms into fully explicit tables:

```
oper adjFormsAdjective : AdjForms -> AP =
 \afs -> {s = \\g,n,c => case <n,c,g> of {
    <Sg, Nom|Voc, Masc _>
      | <Sg, Acc, Masc Inanim>   => afs ! msnom ;
    <Sg, Nom|Voc, Fem>
      | <Pl, Nom|Acc|Voc, Neutr> => afs ! fsnom ;
    <Sg, Nom|Acc|Voc, Neutr>     => afs ! nsnom ;
    <Sg, Gen, Masc _ | Neutr>
      | <Sg, Acc, Masc Anim>     => afs ! msgen ;
    <Sg, Gen, Fem>
      | <Pl, Nom|Acc|Voc, Fem>
      | <Pl, Acc, Masc _>        => afs ! fsgen ;
    <Sg, Dat|Loc, Masc _|Neutr>  => afs ! msdat ;
    <Sg, Dat|Loc, Fem>           => afs ! fsdat ;
    <Sg, Acc, Fem>               => afs ! fsacc ;
    <Sg, Ins, Masc _|Neutr>
      | <Pl,Dat|Loc|Ins, _>      => afs ! msins ;
    <Sg, Ins, Fem>               => afs ! fsins ;
    <Pl, Nom|Voc, Masc _>        => afs ! mpnom ;
    <Pl, Gen,_>                  => afs ! mpgen
    }
  }
```

This definition shows a few more programming constructs, which are borrowed from functional programming languages. The first is **lambda abstractions**, expressions of form `\x -> Exp`, which binds the variable `x` so that it can be used in the expression `Exp`. Lambda abstractions are availale both for general functions (the single arrow `->`) and for tables (the double arrow `=>`); in the latter case the expression `\\x -> Exp` is a shorthand for `table {x => Exp}`, where all branches behave uniformly for `x`.

The second new construct is **case expressions**, of form `case Exp of { Branch* }` where the branches have the same syntax as in tables. This is actually just syntactic sugar for `table { Branch* } ! Exp` but more intuitive to use in programming. The

expression that is matched here is a **tuple**, denoted by angle brackets (< >). Thus it defines a table by cases on triples of number, case, and gender. The order of parameters is different from the desired order gender, number, case, which we considered to make programming more convenient.

In the branches of the case expression, **tuple patterns** are used together with wildcards (_, explained earlier) and **disjunctive patterns** (|). For example, the pattern in the first branch,

```
<Sg, Nom|Voc, Masc _> | <Sg, Acc, Masc Inanim>
```

matches 5 different values: nominative and vocative masculine animate and inanimate, plus accusative inanimate, in the singular number. The case expression is built in such a way that each unique form has a unique branch. The type checker of the GF compiler makes sure that all the 56 different combinations are covered; the reader can be sure that the author had to fix it a few times until he got everything right.

The distinction between the lexical category A and the syntactic category AP is an example of the importance to distinguish between features relevant for morphology (the ones that produce different forms) and features relevant for syntax (the ones used in agreement). An even simpler example is English verbs (V) and verb phrases (VP). The English VP may need access to the full agreement features including gender, number, and person, in particular when they contain reflexive pronouns (*myself*, *yourself*, *himself*, *herself*, etc). Moreover, we need verbs in different compound tenses such as perfect (*have seen*) and future (*will see*). But the English *verb* only needs five different forms: *see, sees, saw, seen, seeing.* The copula *be* needs a few more, but still far from the full array of agreement features and tenses. Hence the English grammar should only specify five forms for the category V (assuming a different category for *be*) and deal with agreement features and compound tenses in the syntactic rules that use verbs in verb phrases.

6.4 Inflection paradigms

Now that we have designed the required types and conversions between them, let us move the the most the inflection paradigms themselves. We start with one that just glues endings to a stem. The definition uses the word *velik* (“large”) as a variable name, as common in grammar books (even though *velik* is not regular in its comparison forms).

```
oper velikA : Str -> AdjForms = \velik -> table {
    msnom => velik ;
    fsnom => velik + "a" ;
    nsnom => velik + "o" ;
    msgen => velik + "og" ;
    fsgen => velik + "e" ;
    msdat => velik + "om" ;
    fsdat => velik + "oj" ;
```

```
    fsacc => velik + "u" ;
    msins => velik + "im" ;
    fsins => velik + "om" ;
    mpnom => velik + "i" ;
    mpgen => velik + "ih"
    }
```

Morphological functions like `velikA` make heavy use of the single-plus operator `+`, which glues strings into tokens. This is in contrast to `++`, typically used in syntax rules, which concatenates lists of tokens and always introduces a token boundary.

The purely concatenative stem+suffix paradigm does not cover all cases that are still considered regular. Most of these have to do with how the stem is derived from the `msnom` form (the masculine nominative singular):

- `msnom` ending with *stan*: like *bolestan* - *bolesn* ("ill");
- `msnom` ending with *an* or *ar*: like *dobar* - *dobr* ("good"; **fleeting *a***);
- `msnom` ending with *ak*: like *nizak* - *nisk* ("low" ; fleeting *a* with **voicing assimilation**);
- `msnom` ending with *ao* or *eo*: like *topao* - *topla* ("warm").

In addition, the adjectives whose stem ends with a **soft consonant** (one of *c, č, ć, j, lj, nj, š, ž, št*)[7], get an *e* instead of *o* in the endings for `nsnom`, `msgen`, and `msdat`.

All of these variations could be dealt with by introducing more paradigms. The **table update** operator `**` gives a concise way to do so, by using `velikA` as a basis and listing exceptions to it. A concise — although a bit unintuitive — way to do this is by treating the `msnom` form as the exception. Thus for instance the paradigm matching adjectives like *dobar* could be defined

```
dobarA : Str -> AdjForms = \dobar ->
  let
      dobr = init (init dobar) + last dobar
  in velikA dobr ** {msnom => dobar}
```

The functions `init` and `last` are defined in the standard library of GF and return the initial part (dropping the last character) and the last character of a string, respectively. This definition also shows a **local definition**, also known as a **let expression**, which defines `dobr` as a constant that can be used inside the rest of the expression.

[7] We had to omit the soft *đ* letter since it was not supported by LaTeX.

6.5 Morphophonemic changes

Multiplying the number of paradigms is not the most satisfactory way to implement predictable variations. A more general way is to define **morphophonemic variations**, patterns that can be used in inflection paradigms in systematic ways. We will here use two such operations: a pattern for soft consonants, and the unvoicing of the last consonant of a string:

```
softConsonant : pattern Str
  = #("c"|"č"|"ć"|"j"|"lj"|"nj"|"š"|"ž"|"št") ;
unvoicing : Str -> Str = \s -> case s of {
  x + "b"  => x + "p" ;
  x + "d"  => x + "t" ;
  x + "z"  => x + "s" ;
  x + "dž" => x + "č" ;
  x + "ž"  => x + "š" ;
  _ => s
  } ;
```

The new construct in these definitions is **pattern matching over strings**. The first definition uses a disjunctive pattern matching some consonants and consonant combinations. The second one uses a case expression with **regular expression patterns** — here simply variables followed by constant strings, but also disjunctive patterns such as are available. With these devices, we can generalize the regular adjective paradigm to

```
velikA : Str -> AdjForms = \velik ->
   let
     velk : Str = case velik of {
       vel + "stan" => vel + "sn" ;
       vel + "ao" => vel + "l" ;
       vel + "ak" => unvoicing vel + "k" ;
       vel + "a" + ("n"|"r") => vel + last velik ;
       vel + "i" => vel ;
       _ => velik
       } ;
     oe : Str = case velik of {
       _ + #(softConsonant) => "e" ;
       _ => "o"
       }
   in {
     msnom => velik ;
     fsnom => velk + "a" ;
     nsnom => velk + oe ;
     msgen => velk + oe + "g" ;
```

```
fsgen => velk + "e" ;
msdat => velk + oe + "m" ;
fsdat => velk + "oj" ;
fsacc => velk + "u" ;
msins => velk + "im" ;
fsins => velk + "om" ;
mpnom => velk + "i" ;
mpgen => velk + "ih" }
```

In the Serbo-Croatian grammar, morphophonemic changes also include palatalization, used e.g. in noun declensions. Factoring out such changes as separate operations is analogous to the **transducer composition** in finite-state morphology (Beesley and Karttunen, 2003), denoted by the operator `.o.` in Xerox Finita State Tool (XFST). Defining names for these operations is analogous to **macros** in XFST. The main difference is that the GF operations have types, which implies that the validity of compositions is checked at compile time; another difference is that GF uses function composition instead of transducer composition.

6.6 Smart paradigms and the morphological lexicon

Traditional grammars and dictionaries often follow the **word and paradigm** model (Hockett, 1954), according to which the inflection of a word is specified by assigning a **paradigm** to it. Paradigms are usually presented as inflection tables of concrete words, such as the noun *rosa* in Latin, or the adjective *velik* in Serbo-Croatian. But their intended use is as functions, where the stem of the concrete word is seen as a variable that can be replaced by other stems to produce their inflections.

The word and paradigm model is a natural choice in GF, where words are primarily seen as abstract syntax objects whose linearizations include complete inflection tables. Pure stem+suffix (or prefix+stem+suffix) paradigms are easy to define as we did in the first version of the `velikA` operation above. The second version, however, departs from this by introducing elements of the **item and process model** (Hockett, 1954): inside the paradigms, operations such as voicing modify the simple concatenation of stems with suffixes. This kind of paradigms are in GF called **smart paradigms** (Détrez and Ranta, 2012).

The advantage of smart paradigms is that we need a smaller set of them. This makes it easier for lexicon builders to assign paradigms to words, since they do not need to choose among so many alternatives. The limiting case is automated selection of paradigms. In this method, it is enough to know a word and its part of speech (if the latter cannot be inferred as well) to produce the full inflection table. Thus it is possible to derive a morphological lexicon from a plain list of words that have part of speech tags.

A generalization of smart paradigms with single words as arguments is functions that take two or more words. This method formalizes a common practice in traditional dictionaries. For instance, Latin nouns are typically specified by giving two **principal**

forms: the nominative and genitive singular, together with the gender, which are almost always sufficient to determine all of the ten forms of the noun. An even more familiar example is thematic forms of verbs such as English *go-went-gone*. The remaining forms *goes* and *going* can be derived by using smart paradigms. If only one form is given, all the other forms are derived by the smart paradigm, which implements predictable variations such as *cry-cries-cried* and *echo-echoes-echoed*.

Smart paradigms lead to an intersting question: how predictable are the word forms in different languages. An evaluation for English, French, and Finnish, showed that, typically, just above one form on average was enough, and that one form was enough for close to 90% of words (Détrez and Ranta, 2012). This applied even to Finnish, where nouns and verbs have hundreds or thousands of forms (Koskenniemi, 1983) and inflections involve complex changes in stems and suffixes. For a collection of 10,355 Finnish verbs, for example, 96% were inferred correctly from just one form (the first infinitive) and 99% when a second form (third person singular past tense indicative) was given; the average was 1.09 forms per word.

6.7 Non-concatenative morphology

We saw in Sec. 3 how the record data structure can be used for discontinuous constituents in syntax. They are useful in morphology as well, in particular, in the **non-concatenative morphology** of Semitic languages. In the RGL, they are used in Amharic, Arabic, Hebrew, and Maltese.

A slightly simplified code for Arabic defines two concepts, **roots** and **patterns**. A root consists of three consonants, called **radicals**.

```
Root = {F,C,L : Str}
```

A pattern consists of four strings, specifying what vowels (and possibly consonants as well) are to be inserted between the radicals:

```
Pattern = {F,FC,CL,L : Str}
```

Putting these together into a word is done by interleaving:

```
word r p = p.F + r.F + p.FC + r.C + p.CL + r.L + p.L
```

For example, the verb form *naktubu* ("we write") is a result of combining the root *ktb* with the pattern *naFCuLu*.

This simple model takes care of a majority of words, but some refinements are needed for roots not consisting of three consonants as well as for "weak" root consonants, which may undergo changes when patterns are applied to them.

7 The Resource Grammar Library

We have shown how GF can model the syntax and morphology of languages and take care of variations such as agreement, word order, discontinuity, inflection, and morphophonemic changes, while maintaining a common abstract syntax that encodes pure constituency independently of language. GF has proven to be expressive enough to describe these phenomena in all of the languages studied so far, with a possible counterexample in the **hyperbaton** phenomenon of Classical Latin (Hublet, 2022). However, doing this in full detail is still hard work and needs expertise in both programming and linguistics.

Hence a need arose, at an early stage of GF, to make sure that the work only needs to be done once per language and can then be reused by other programmers. The result was a **Resource Grammar Library** (RGL, (Ranta, 2009)), which at the time of writing contains grammars for 54 languages[8]. 35 of these grammars are "complete" in the sense that they fully implement a common abstract syntax. Of the rest, many have a substantial morphology and a part of syntax that is sufficient for some intended purpose, such as natural language generation. 24 languages have large-scale dictionaries implementing parts of an abstract syntax based on WordNet senses (Fellbaum, 1998; Angelov, 2020).

In addition to several Germanic, Romance, and Slavic languages, the RGL contains also Fenno-Ugric (Listenmaa and Kaljurand, 2014), Indo-Iranian (Virk, 2013), Semitic (Dada and Ranta, 2006; Camilleri, 2013), Bantu (Ng'ang'a, 2012; Pretorius et al., 2017; Bamutura et al., 2020; Kituku et al., 2021), and East-Asian (Zimina, 2012; Ranta et al., 2015) languages. Figure 2 shows a world map with the languages placed in their approximate places.

The library has been built by the community. GitHub lists 45 contributors, but since much of the library was built before it was on GitHub, the correct figure is at least over 60. Building a complete RGL implementation for one language has typically been a work of three to six person months and often the topic of a Master's thesis. Some languages have been implemented as parts of PhD theses (Virk, 2013; Erdenebadrakh, 2015; Kituku et al., 2019), in connection with larger projects about languages or language families. Many have been implemented by senior academics as well; one author of this paper has contributed to 14 languages. But the normally required skill level is that of a Masters student who has made an introductory course in GF and is native or fluent in the target language. Many projects have started with a two-week GF summer school, which have been organized seven times and gathered 30 to 50 participants each[9].

7.1 The core abstract syntax

The syntax of the RGL is based on a **core abstract syntax**, which has

[8] `http://www.grammaticalframework.org/lib/doc/synopsis/` gives a synopsis and `https://github.com/GrammaticalFramework/gf-rgl` the full code.

[9] `http://school.grammaticalframework.org/`

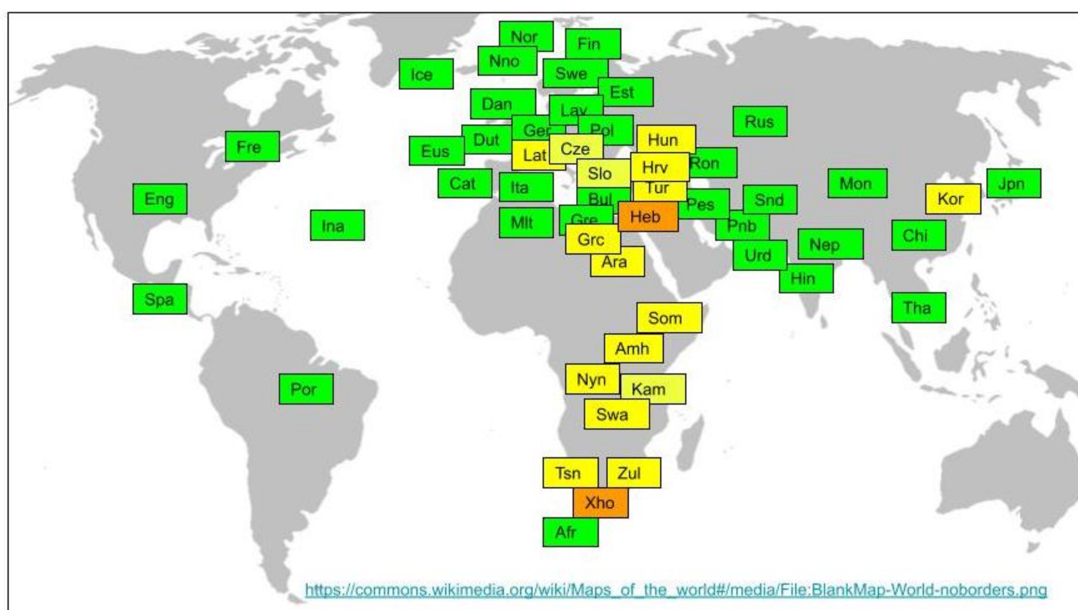

Figure 2: Resource Grammar Languages as of October 2022. Green colour means complete (or close to complete) implementation of morphological rules and the core abstract syntax, yellow means a significant subset, and orange a smaller subset. The underlying world map is from Wikimedia Commons.

- 87 categories
- 217 combination functions (ones that take arguments)
- 140 lexical items (structural words)

These numbers vary a little, as later versions of the abstract syntax have added functions that are not implemented by all earlier languages.

The categories and functions are mostly "familiar" ones such as sentences, noun phrases, verb phrases, adjectival phrases, adverbials, determiners, interrogatives, relatives, conjunctions, subjunctions, and numerals. They are rooted in the Western tradition of grammar, but they are common in grammar books of all types of languages. Since GF is an open framework, it is possible to write grammars based on very different concepts that are more "native" for different languages. But this has rarely been done, and the value of a common abstract syntax is that it makes it much easier to use the library: one can easily use the grammar of a language that one does not even know, for instance when writing a text generation system.

The core abstract syntax is, in a sense, a **universal grammar**, but its claims are more modest than this term suggests. We do not claim that it is *the* grammar of *all* languages, but just that it encodes parts of many languages. The RGL project can be seen as an experiment in how much abstract syntax structure languages can share.

The functions and their linguistic foundations, are explained in (Ranta, 2009) (which is still mostly up to date) and in the on-line synopsis[10]. Let us here just look at the "core of the core", known as the **mini resource grammar**. It contains the following combination functions:

```
PredVPS : NP -> VP -> S       -- John walks
ComplV2 : V2  -> NP -> VP     -- love it
UseComp : Comp -> VP          -- be small
CompAP  : AP  -> Comp         -- small
AdvVP   : VP -> Adv -> VP     -- sleep here
DetCN   : Det -> CN -> NP     -- the house
AdjCN   : AP -> CN -> CN      -- big house
PrepNP  : Prep -> NP -> Adv   -- in the house
```

These functions operate on phrasal categories, which are linked to lexical categories in expected ways, such as

```
PositA  : A -> AP
```

which selects the positive forms of an adjective; the comparison forms are not included in the mini resource.

[10] `http://www.grammaticalframework.org/lib/doc/synopsis/`

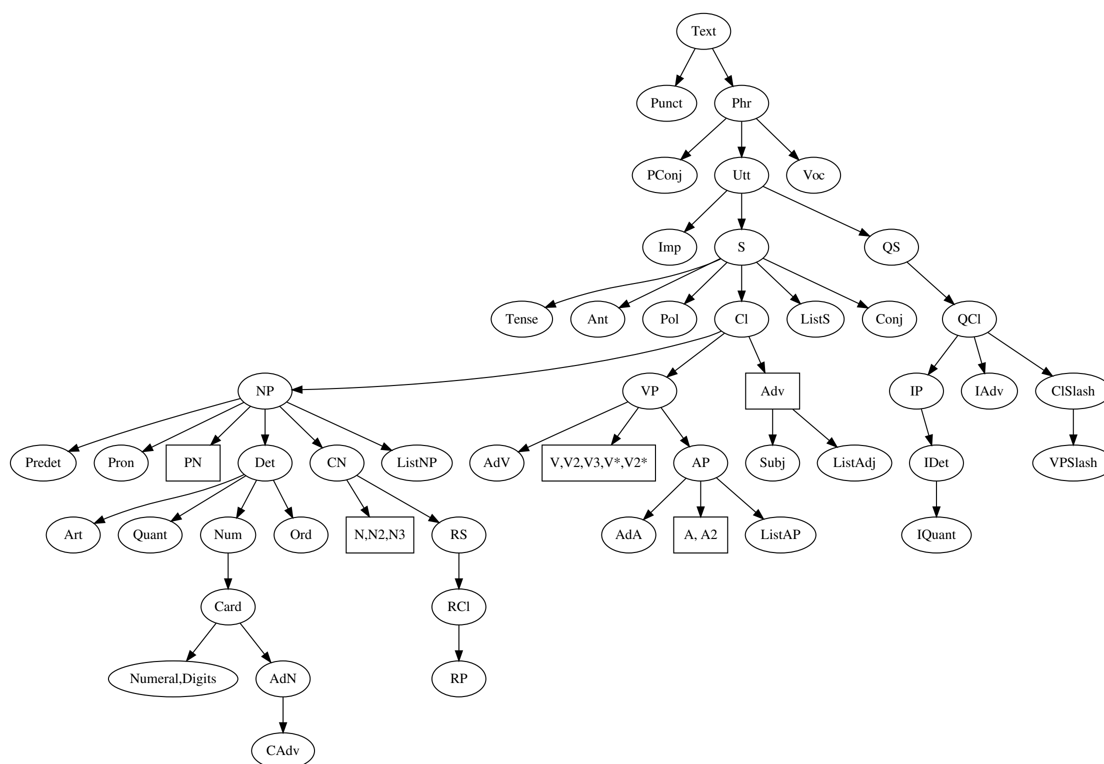

Figure 3: Categories of the core abstract syntax of the GF Resource Grammar Library. The rectangular boxes mark contain lexical categories. The asterisks in V and V2 refer to sets of further subcategories such as V2V, which takes both an NP and a VP complement.

The RGL implementation project of a new language often starts with the mini RGL, which was originally developed to be used in the GF summer schools. It gives a representative idea of what parameters are needed in the language and is a good basis of extension to the complete abstract syntax, where constructs such as questions, relative clauses, and coordination are added, and the morphology needs to be extended with verb tenses and adjective comparison forms, as well as numerals and more kinds of pronouns. A category of **clauses**, Cl, is inserted between S and VP, to represent NP-VP predications from which sentences (S) can be built in different tenses an polarities.

An overview of core abstract syntax categories is shown in Fig. 3. The hierarchy is not strictly unidirectional, because for instance sentences (S) can occur as constituents of noun phrases (NP), as in *the fact that the Earth is round*.

7.2 The API

The core abstract syntax is a compact and non-redundant coverage of syntactic structures. Its functions usually build binary combinations, which results in deep trees. When

parsing the string *I have four children*, the resulting tree is, with comments added at each combination step,

```
UseCl                            -- use clause as sentence
  (TTAnt TPres ASimul)           -- present tense simultaneous
    PPos                         -- positive polarity
    (PredVP                      -- predication with VP
      (UsePron i_Pron)           -- subject "I"
      (ComplSlash                -- complement to VP/NP
        (SlashV2a have_V2)       -- VP/NP from the V2 "have"
        (DetCN                   -- add determiner to noun
          (DetQuant              -- with a determiner phrase
          IndefArt               -- that is indefinite
          (NumCard               -- with a cardinal number
            (NumNumeral          -- numeral in words
              (num (pot2as3      -- built in several steps
                (pot1as2         -- ...
                  (pot0as1       -- ...
                    (pot0        -- ...
                      n4))))))))  -- from the digit "four"
          (UseN child_N))))      -- using noun "child"
```

This tree may be difficult to understand because of the many steps of a theoretical nature, such as the formation of VP from the **slash category** VP/NP. But at least it need not be constructed by hand, if returned by the parser. However, when the library is used in an applications, writing trees like this manually is laborious and error-prone.

For an easier use as a library, the RGL has been equipped with an **Application Programming Interface** (API), which is a layer of abstraction over the core abstract syntax. The API is presented as a set of overloaded functions, whose names have the form `mk`C where C is the value type. Every function of the core abstract syntax has a corresponding overloaded function, which already helps the programmer by eliminating the need to remember the names of the functions. But an even greater help is the creation of **shortcuts**, which enable bypassing some steps in the core abstract tree construction. The above example has the following shortcut:

```
mkS (mkCl i_NP have_V2 (mkNP (mkCard "4") child_N))
```

One of the API functions that it uses is `mkCl`, "make clause", building predication clauses. The instance of `mkCl` that is used is

```
mkCl : NP -> V2 -> NP -> Cl
```

Its definition in terms of core abstract syntax is

```
mkCl subj verb obj =
  PredVP subj (ComplSlash (SlashV2a verb) obj)
```

The other functions have similar definitions, as can be seen by comparing this expression with the full core abstract syntax tree.

The overloaded `mkCl` function has several instances corresponding to different kinds of predicates. They are documented by English example sentences; here are a few examples of the 21 instances, at the same time showing a subset of the **verb subcategories** included in the RGL:

```
mkCl : NP -> V -> Cl               -- she sleeps
mkCl : NP -> V2 -> NP -> Cl        -- she loves him
mkCl : NP -> V3 -> NP -> NP -> Cl  -- she sends it to him
mkCl : NP -> VV -> VP -> Cl        -- she wants to sleep
mkCl : NP -> VS -> S  -> Cl        -- she says that she sleeps
mkCl : NP -> V2V -> NP -> VP -> Cl -- she begs him to sleep
mkCl : NP -> A  -> Cl              -- she is old
mkCl : NP -> N -> Cl               -- she is a woman
mkCl : NP -> Adv -> Cl             -- she is here
```

The API documentation[11] is generated automatically from the GF code. The English examples are shown there in all RGL languages, by linearizing the underlying trees.

While the syntax API is shared among the RGL languages, the morphology is documented separately for each language. The reason is that languages have their own parameter and paradigm systems, which require different sets of functions. A common feature is that most of the languages have overloaded groups of smart paradigms following the same naming convention, e.g. `mkN` to build nouns. German, for instance, has

```
mkN : (snom : Str) -> N
mkN : (snom : Str) -> Gender -> N
mkN : (snom,plnom : Str) -> Gender -> N
mkN : (snom,sacc,sdat,sgen,pnom,pdat : Str) -> Gender -> N
mkN : Str -> N -> N
mkN : N -> N -> N
```

In German, inferring the morphological properties from only one string is less often possible than in many other languages. Adding a gender helps a little, but the most useful instance of `mkN` is the third one, which takes two forms and the gender; this is also the information often provided by dictionaries. The worst case needs six arguments and is seldom needed. The last two are used for building compounds.

[11] `http://www.grammaticalframework.org/lib/doc/synopsis/\#Cl`

7.3 Variations in RGL languages

An obvious challenge in the RGL is to maintain a common abstract syntax in all of the different types of languages. Success in this task is not automatic, because the in-built requirement of compositionality (Section 5) could actually exclude some kinds of concrete syntaxes. The existing implementations of a representative number of languages from different families (Fig. 2) gives some hope that one can cover many more.

The only counterexample to compositional linearization of the common abstract syntax so far is in the Japanese grammar Zimina (2012), where the coordination of relative clauses (*who is sleeping and whose mother is sleeping*) does not apply. This means that coordinated relative clauses have to be paraphrased, i.e., converted to other syntactic structures that have the same meaning.

Paraphrasing is also typically needed in applications where the RGL is used for translation. This is because a syntactic structure that is natural in one language can be very clumsy in another one, even if it were grammatically possible. An example is the passive constructions with an agent used for topicalizing the object in English. Even if a language has passives with agents, their usability may be more restricted. Finnish is an example. The English sentence *the eclipse was seen by millions of people* is more fluently translated by just fronting the object in an active sentence where "millions of people" is the subject: *pimennyksen näkivät miljoonat ihmiset.*

Paraphrases are easy to define in a layer of **semantic grammars**, where the abstract syntax is more abstract than the RGL. A semantic grammar can for instance have a function

```
fun TopObjectS : NP -> V2 -> NP -> S
```

which is linearized either

```
fun TopObjectS subj verb obj = mkS (mkCl obj (PassAgVP v2 subj))
```

in languages like English or

```
fun TopObjectS subj verb obj = FrontObjS (mkCl subj v2 obj)
```

in languages like Finnish. The functions `PassAgVP` and `FrontObjS` do not actually belong to the core abstract syntax at all, because they do not exist in all languages. Thus the RGL does not assume that *all* syntactic structures are present in all languages. To deal with the ones that can only be found in some languages, the RGL has, in addition to the core abstract syntax, separate modules that extend it.

In the following section, we explain in more detail a case study that is among the most complex ones in the RGL.

8 Case study: contraction of adpositions, pronouns and negation particle in Somali

As a case study, we show the RGL implementation of a complex morphosyntactic phenomenon in Somali: the contraction of adpositions, a subset of pronouns and the negation particle, as well as their placement in the verbal group.

The following pair (Saeed, 1999, p. 110, ex. 106) shows a single adposition *u* 'for', introducing the prepositional phrase 'for Ali', which in Somali is discontinuous.

(1) *Cali shaah u samee*
Ali tea for do:IMP
'Make tea for Ali'

Discontinuity in itself is not a hard problem in GF: thanks to the record syntax, linearization rules can add strings to a record, and postpone building the phrase until more features are known. However, what makes Somali adpositions a challenging exercise for GF is their obligatory contractions with other parts of speech.

Next, Saeed (1999, p. 39, ex. 48) shows two adpositions merging into one.

(2) *Faarax sidan ugu samee*
Faarax si-tan u+u samee
Farah way-this in+for do:IMP
'Do it (in) this way for Farah'

Again the English translation features two prepositional phrases as continuous constituents: 'in this way' and 'for Farah'. In Somali, the noun phrases *sidan* 'this way' and *Faarax* 'Farah' appear in the beginning of the sentence, and the two adpositions *u* and *u* merge into one orthographic word[12], *ugu*, before the verb. Unlike in English where forms like *don't* and *do not* coexist, the Somali contractions are obligatory.

Nilsson (2022, p. 127, ex. 12.2.6a) shows a more complex contraction, with the object pronoun *i* 'me', adposition *ka* and negation *ma* merging into a single word *igama*:

(3) *Igama dul boodi kartid*
i+ka+ma dul boodi kartid
me+from+NEG above jump:INF can:SG2:NEG
'You can't jump over me'

Note that the adposition *ka* on its own has meanings such as 'from' or 'about'. In this expression, it is part of the compound adposition *ka dul* 'over', where the *ka* component merges with the object pronoun and the negation in the beginning of the sentence, and *dul* appears before the infinitive verb.

[12] Also one phonological word: according to (Saeed, 1999, p. 39) *ugu* has only a single high tone, not two tones.

In the remainder of this section, we describe the contractions to the extent needed for this example. For a more complete description, we direct the reader to Saeed (1999) and Nilsson (2022).

Morphology From the systematic listing in Saeed (1999, pp. 38–41), we count at least 80 distinct combinations, many of which feature nontrivial assimilations and metathesis, and are therefore best stored as full forms, not combined from smaller pieces.

- 4 for adposition (not a *combination* per se, but we include these forms in the count for the sake of implementation, explained later);
- 4 for impersonal subject pronoun + object pronoun;
- 6 for adposition + adposition;
- 24 for adposition + any pronoun; and
- 42 for adposition + adposition + any pronoun.

Neither Saeed (1999) nor Nilsson (2022) cover all of the possible combinations, so it is unclear whether they do not contract, or are just unlikely to occur together in a single verbal group. For instance, there is no mention of combining the first item on the list, an impersonal subject pronoun and an object pronoun (4 distinct forms) with one or more adpositions—we assume for the reason that such sentences were not attested in a corpus.

However, later examples in Saeed (1999) reveal forms that are missing from the systematic listing on pp. 38–41, such as impersonal subject pronoun + reflexive pronoun on p. 178, ex. 11 featuring orthographical changes, even if phonologically the process is rather concatenative. So in addition to the explicitly listed 80 forms, we have included in the GF implementation also forms that were found elsewhere in Saeed (1999) (mostly combinations with reflexive pronouns), bringing the total number to 88 unique forms. This is still not a complete list, but based on the source material, it should cover the bulk of the combinations appearing spontaneously.

Negation suffix "ma" can also appear with these 88 combinations, but its addition is purely concatenative, so we have chosen to attach the negation in a separate step, using *run-time gluing* operation in GF.

GF parameters These 88 forms are incorporated into an inflection table in GF. We build the inflection table of three parameter types, which are shown all at once in Figure 4, and explained later one by one.

First, we introduce a parameter for single adpositions, called `Adposition`. This parameter `Adposition` is present in lexical categories, such as Prep, Adv, V2, and others that may introduce a nominal argument with an adposition. (The lack of adposition, `NoAdp`, is included as an explicit value, corresponding to a direct object.)

```
param
  Adposition = NoAdp | U | Ku | Ka | La ;

  AdpCombination =
    Single Adposition     -- 0-1 adpositions (0 = NoAdp)
  | ImpersSubj Adposition -- impersonal subject + 0-1 adpositions
  | Ugu | Uga | Ula       -- two adpositions (6 distinct forms)
  | Kaga | Kula | Kala ;

  AdpObjAgr =            -- NB. separate from verbal agreement
    Sg1Obj
  | Sg2Obj
  | Pl1Obj Inclusion
  | Pl2Obj
  | ReflexiveObj
  | ZeroObj ;    -- i.e. the AdpCombination value on its own

oper
   allContractions : AdpObjAgr => AdpCombination => Str = table {
     Sg1Obj => table {
        Single NoAdp     => "i" ;   -- just the object pronoun
        Single Ka        => "iga" ;
        ...
        ImpersSubj NoAdp => "lay" ;
        ...
        Kala          => "igala"    -- object pronoun + Ka + La
     } ;
     ...
     ZeroObj => table {
        Single NoAdp => "" ;   -- 3rd person direct object = 0
        Single Ka    => "ka" ; -- just the adposition
        ...
        Kala         => "kala"
     }
   } ;
```

Figure 4: Three parameter types needed to represent the contractions, and a fragment of the full inflection table that covers the contractions.

The second parameter, called `AdpCombination`, lists the combinations of adpositions and impersonal subject pronouns.[13] They are represented as one parameter, because both of these components can combine with *object pronouns*: it was a natural way to cluster the parts of speech: object pronouns vs. everything else.The combinations of two adpositions exhibit syncretism: there are only 6 distinct forms, because some of the forms cover many combinations. `AdpCombination` is present in phrasal categories where the full information of all objects and obliques and negation is still open. Referring to Figure 3, `Adposition` appears in V2*, V3, N2, A2, Prep, Adv, IAdv, and `AdpCombination` in VP, VPSlash and ClSlash.

The third parameter, `AdpObjAgr` lists all possible object pronouns that combine with the `AdpCombination` values, to produce the 88 distinct forms. As the last value, we include a zero morpheme, used for third person objects, to add no extra segment into the final contraction. For example combining an `AdpCombination` with a zero object just returns the combination itself: `ZeroObj` + `Ugu` returns the string *ugu*, whereas `Sg1Obj` + `Ugu` returns the string *iigu*.

`AdpObjAgr` appears in the categories where a nominal object or oblique argument can be added: VP, VPSlash, ClSlash and Adv. Note also that `AdpObjAgr` parameter is separate from the parameter that affects the form of the finite verb of the clause. The category `NP` records the full agreement `Agr`, and `AdpObjAgr` is computed from `Agr` whenever a NP becomes the object of a VP or an Adv.

The full inflection table has the type `ObjAgr => AdpCombination => Str` and it produces all the combinations, including also the empty string (for `ZeroObj` and `Single NoAdp`), which means a third person direct object.

Combination of parameters Figure 5 shows part of the rules on how to combine two adpositions into a combination parameter. Due to the syncretism of the contractions, this design streamlines the process: suppose that a VP that inherits the adposition `Ku` from a V2 and the adposition `Ka` from an Adv. Instead of storing the pair `(Ku,Ka)`, the `combine` function merges them into a single `AdpCombination` value `Kaga`.

Updating the AdpCombination value As explained previously, lexical categories have an `Adposition` field, which is elevated to `AdpCombination` in phrasal categories. Below are some examples of how the values are updated at the application of different functions.

- `UseV : V -> VP` elevates an intransitive verb into a VP, and inserts a `Single NoAdp` as the VP's `AdpCombination`. This value can still be updated, if an adverbial is added to the VP.

[13] As previously mentioned, due to gaps in data, we do not currently allow impersonal subject pronoun to combine with two adpositions. We do allow the combination impersonal subject + one adposition + object pronoun, but some of the forms have an empty linearisation, because we could not find a form.

```
oper
  combine : Adposition -> Adposition -> AdpCombination
  combine adp1 adp2 = case <adp1,adp2> of {
    <U, U|Ku> => Ugu ;
    <U, Ka>   => Uga ;
    <U, La>   => Ula ;
    <Ku|Ka,
     Ku|Ka>   => Kaga ; -- 4 combinations, same form
    <Ku, La>  => Kula ;
    <Ka, La>  => Kala ;
    <NoAdp, p> => Single p
  }
```

Figure 5: Part of the rules to combine two `Adposition` values into an `AdpCombination`.

- `PassV2 : V2 -> VP` makes a V2 into a passive VP, for which the corresponding construction in Somali is to use an impersonal subject: "one saw me" for "I was seen". The function `PassV2` elevates the V2s inherent `Adposition` into an `AdpCombination` with the constructor `ImpersSubj`.
- Any other `V2* -> VP` function uses the constructor `Single` to wrap the verb's `Adposition` into an `AdpCombination`
- `AdvVP : VP -> Adv -> VP` adds an adverbial into a VP. This process updates the VP's `AdpCombination` with the `Adposition` of the adverb, with rules similar to Figure 5, but this time taking an `Adposition` and an `AdpCombination`.

Parameters' journey in the syntax tree Figure 6 shows a GF parse tree for Example 3, where the concrete strings are shown in black Times New Roman, and the internal parameters in colorful Courier font with dotted lines from the lexical categories. The string *igama* is only linearized when all three contributing parameters are known: negation from `Pol`, adposition from `Prep` and object pronoun from `Pron`.

All these arguments propagate their inherent parameter up to the S level, at which time the inflection table `allContractions` from Figure 4 is consulted. The VP didn't contain any other adposition, so the full value of the `AdpCombination` became `Single Ka`. The Adv's object NP was build out of `i_Pron`, which contributed with the `AdpObjAgr` value `Sg1Obj`. Finally, the negation morpheme *ma* was glued onto the `iga` that came from the inflection table, and the full contraction was linearized into a single token.

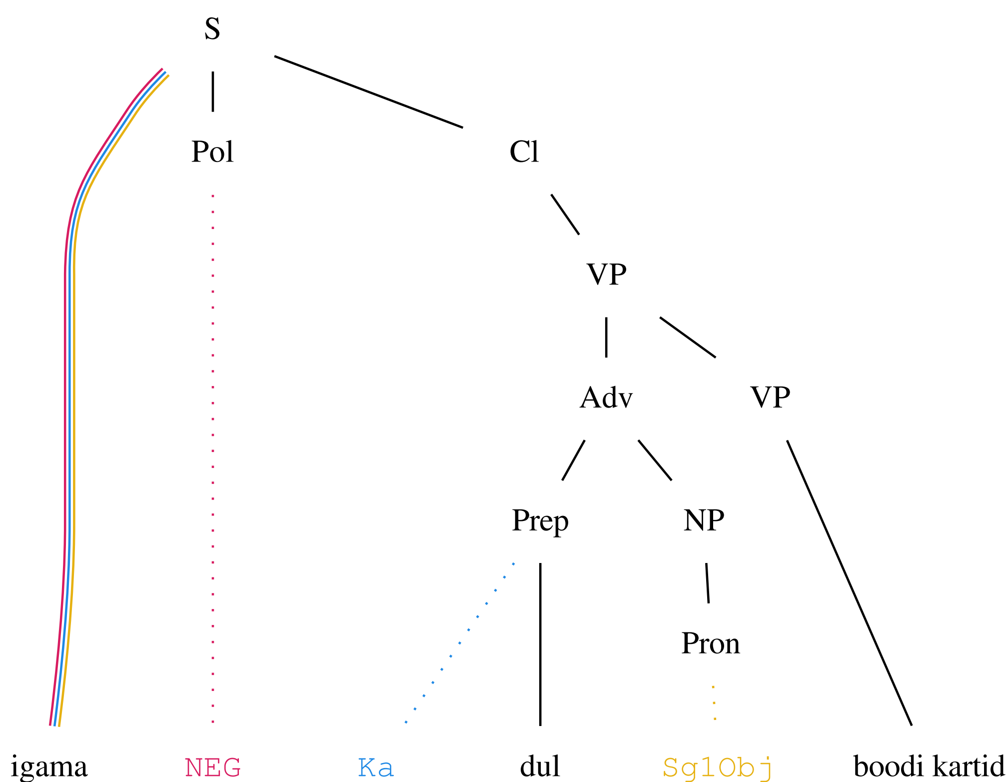

Figure 6: Modified parse tree for Example 3 *igama dul boodi kartid* 'you cannot jump over me', with internal parameters added in the tree alongside the strings.

9 Semantics

GF was originally designed to support interlingual semantic representations in abstract syntax. Thus it supports traditional **logical semantics** in the style of Montague (1974), which defines mappings from abstract syntax to logical formulas. GF actually grew out of Montague-style semantics in constructive type theory (Martin-Löf, 1984; Ranta, 1994). But as a general framework, GF also supports classical and other logics.

In logical semantics, every category of an abstract syntax is given a **semantic type**. The following types cover the usual Montague-style semantics:

- for `S` (sentences), the type `prop` (propositions; truth values in classical logic),
- for `PN` (proper names), the type `ind` (individuals),
- for `VP` (verb phrases), the type `ind -> prop` (functions from `ind` to `prop`),
- for `V2` (two-place verbs), the type `ind -> ind -> prop`,

- for `NP` (noun phrases), the type `(ind -> prop) -> prop` (of quantifiers).

The semantic types are used in **interpretation functions**, which operate on the abstract syntax in a compositional way, just like linearization functions operate on linearization types (cf. Section 5). Interpretation functions can be defined in GF as follows (with names `i`C for each category C):

```
-- for Pred : NP -> VP -> S
iS (Pred np vp) = iNP np (iVP vp)

-- for Compl : V2 -> NP -> VP
iVP (Compl v2 np) = \x -> iNP np (\y -> iV2 v2 x y)

-- for UsePN : PN -> NP
iNP (UsePN pn) = \p -> p (iPN pn)
```

The last rule expresses one of the most famous ideas of Montague: it "raises" proper names to quantifiers.[14]

The most comprehensive Montague-style logical semantics in GF is Bernardy and Chatzikyriakidis (2017), which defines semantics for the entire abstract syntax of the GF Resource Grammar Library and applies it to the FraCaS test suite (Kamp et al., 1994). In addition, there are several specialized systems, addressing for instance software specifications (Burke and Johannisson, 2005) and legal language (Angelov et al., 2013; Ranta et al., 2022). Since semantics is defined on the abstract syntax, it is usable for all languages for which a concrete syntax is defined. Thus Bernardy and Chatzikyriakidis (2017) makes Montague semantics available for all RGL languages.

A second kind of semantic representations is **ontologies**, both general ones such as SUMO (Pease, 2011) which was equipped with a multilingual GF grammar in Angelov and Enache (2010), and more specific **domain ontologies**, such as ones found on the semantic web (Dannélls et al., 2012). Domain ontologies are probably the dominating type of applications of GF. They are encoded as abstract syntaxes and used as interlinguas for translation, as well as starting points of NLG. The Abstract Wikipedia project (Vrandečić, 2021; Ranta, 2022) is one of the latest examples. The ontology is represented as abstract syntax functions such as

```
fun AttrFact : Attr -> Obj -> Val -> Fact
```

for Wikidata facts (Vrandečić and Krötzsch, 2014) of the form "the *Attr* of *Obj* is *Val*". The linearization can be defined, with the RGL API, as

```
lin AttrFact attr obj val =
  mkCl (mkNP the_Det (mkCN attr obj)) val
```

[14]In most applications, interpretation functions are defined outside GF, typically in Haskell (Ranta, 2011b). This makes it easier to link them with other software, such as databases or theorem provers.

Such rules apply to all languages that implement a sufficient part of the API. This makes it possible to write NLG rules for large sets of languages simultaneously. One key task in the Abstract Wikipedia project is to extend the RGL to 300 languages.

Lexical semantics could be seen as a third kind of semantics, but also as a resource for both logical and ontology-based semantics. GF has been used to formalize and generalize both FrameNet (Fillmore et al., 2003; Gruzitis and Dannélls, 2017) and WordNet (Fellbaum, 1998; Angelov, 2020), by representing lexical senses as abstract syntax functions, equipped with linearizations to different languages.

10 Conclusion

GF is a "full-stack" grammar formalism that covers all aspects of written language from semantics to morphophonemics. Its Resource Grammar Library (RGL) implements the main syntax rules of over 40 languages with a shared abstract syntax, and its WordNet-based interlingual lexicon includes 24 languages. The syntax and lexicon together make GF usable as a wide-coverage rule-based translation system — in theory. In practice, the first attempts to achieve this did not reach competitive quality (Kolachina and Ranta, 2015), and it is unclear if open-domain automatic translation is a good use of resources when developing GF. Most work in GF has instead focused on other things: domain-specific translation, natural language interfaces to formal systems, text generation, linguistic resources, and computational infrastructure.

Many advantages of GF in natural language processing are shared with other rule-based approaches: they enable knowledge-based systems that are completely controllable and explainable, and they do not require huge amounts of data and can therefore be applied to languages without such data. The distinguishing features of GF are its interlinguality and its extensive use of abstractions from functional programming. The GF software builds on modern ideas from programming language technology and is mature and extensive. It is readily usable in different operating systems, mobile phones, web applications, and embeddings of GF in other programming languages.

In combination with the linguistic knowledge contained in the RGL, GF software is aimed to help non-linguist programmers to build systems that have high-quality natural language processing as a component. For linguists, GF provides a platform for powerful abstractions and generalizations, including cross-lingual ones.

References

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

Angelov, K. (2020). A parallel WordNet for English, Swedish and Bulgarian. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 3008–3015, Marseille, France. European Language Resources Association.

Angelov, K., Bringert, B., and Ranta, A. (2009). PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, 19:201–228.

Angelov, K., Camilleri, J., and Schneider, G. (2013). A framework for conflict analysis of normative texts written in controlled natural language. *The Journal of Logic and Algebraic Programming*, 82:216–240.

Angelov, K. and Enache, R. (2010). Typeful Ontologies with Direct Multilingual Verbalization. In Fuchs, N. and Rosner, M., editors, *CNL 2010, Controlled Natural Language*.

Angelov, K. and Ljunglöf, P. (2014). Fast Statistical Parsing with Parallel Multiple Context-Free Grammars. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 368–376, Gothenburg, Sweden. Association for Computational Linguistics.

Angelov, K. and Ranta, A. (2009). Implementing Controlled Languages in GF. In *CNL-2009, Controlled Natural Language Workshop, Marettimo, Sicily, 2009*.

Bamutura, D., Ljunglöf, P., and Nabende, P. (2020). Towards Computational Resource Grammars for Runyankore and Rukiga. In *Language Resources and Evaluation (LREC) 2020*, pages 2846–2854, Marseille, France.

Beesley, K. and Karttunen, L. (2003). *Finite State Morphology*. CSLI Publications.

Bender, E. M., Drellishak, S., Fokkens, A., Goodman, M. W., Mills, D. P., Poulson, L., and Saleem, S. (2010). Grammar Prototyping and Testing with the LinGO Grammar Matrix Customization System. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden, System Demonstrations*, pages 1–6.

Bernardy, J.-P. and Chatzikyriakidis, S. (2017). A Type-Theoretical system for the FraCaS test suite: Grammatical Framework meets Coq. In *IWCS 2017*.

Bresnan, J., editor (1982). *The Mental Representation of Grammatical Relations*. MIT Press.

Burke, D. A. and Johannisson, K. (2005). Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In P. Blache and E. Stabler and J. Busquets and R. Moot, editor, *Logical Aspects of Computational Linguistics (LACL 2005)*, volume 3492 of *LNCS/LNAI*, pages 51–66. Springer. `http://www.springerlink.com/content/?k=LNCS+3492`.

Butt, M., Dyvik, H., King, T. H., Masuichi, H., and Rohrer, C. (2002). The Parallel Grammar Project. In *COLING 2002, Workshop on Grammar Engineering and Evaluation*, pages 1–7.

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

Dada, A. E. and Ranta, A. (2006). Implementing an Open Source Arabic Resource Grammar in GF. In *20th Arabic Linguistics Symposium. Western Michigan University March 3-5 2006*.

Dannélls, D., Damova, M., Enache, R., and Chechev, M. (2012). Multilingual online generation from semantic web ontologies. In *Proceedings of the 21st international*

conference on World Wide Web, pages 239–242. ACM.

Détrez, G. and Ranta, A. (2012). Smart paradigms and the predictability and complexity of inflectional morphology. In *EACL 2012*.

Dymetman, M., Lux, V., and Ranta, A. (2000). XML and multilingual document authoring: Convergent trends. In *Proc. Computational Linguistics COLING, Saarbrücken, Germany*, pages 243–249.

Erdenebadrakh, N. (2015). *Implementierung der Grammatik des modernen Mongolischen in der funktionalen Programmiersprache "Grammatical Framework"*. PhD thesis, CIS, LMU Munich.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. MIT Press.

Fillmore, C. J., Johnson, C. R., and Petruck, M. R. (2003). Background to FrameNet. *International Journal of Lexicography*, 16(3):235–250.

Fuchs, N. E., Kaljurand, K., and Kuhn, T. (2008). Attempto Controlled English for Knowledge Representation. In Baroglio, C., Bonatti, P. A., Małuszyński, J., Marchiori, M., Polleres, A., and Schaffert, S., editors, *Reasoning Web, Fourth International Summer School 2008*, number 5224 in LNCS, pages 104–124. Springer.

Gruzitis, N. and Dannélls, D. (2017). A multilingual FrameNet-based grammar and lexicon for Controlled Natural Language. *Language Resources and Evaluation*, 51(1):37–66.

Harper, R., Honsell, F., and Plotkin, G. (1993). A Framework for Defining Logics. *JACM*, 40(1):143–184.

Hockett, C. F. (1954). Two models of grammatical description. *Word*, 10:210–233.

Hublet, F. (2022). IDL-PMCFG: a grammar formalism for describing free word order languages. *Journal of Logic, Language, and Information*, 31:327–388.

Joshi, A. (1985). Tree-adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions. In Dowty, D., Karttunen, L., and Zwicky, A., editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press.

Kamp, H., Crouch, R., van Genabith, J., Cooper, R., Poesio, M., van Eijck, J., Jaspars, J., Pinkal, M., Vestre, E., and Pulman, S. (1994). Specification of linguistic coverage. FRACAS Deliverable D2.

Kituku, B., Nganga, W., and Muchemi, L. (2019). Towards Kikamba Computational Grammar. Journal of Data Analysis and Information Processing. *Journal of Data Analysis and Information Processing*, 7:250–275.

Kituku, B., Nganga, W., and Muchemi, L. (2021). Leveraging on Cross Linguistic Similarities to Reduce Grammar Development Effort for the Under-Resourced Languages: a Case of Kenyan Bantu Languages. In *2021 International Conference on Information and Communication Technology for Development for Africa (ICT4DA)*, pages 83–88.

Kolachina, P. and Ranta, A. (2015). GF Wide-coverage English-Finnish MT system for WMT 2015.

Kolachina, P. and Ranta, A. (2016). From Abstract Syntax to Universal Dependencies. *Linguistic Issues in Language Technology*, 13:1–57.

Koskenniemi, K. (1983). *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* PhD thesis, University of Helsinki.

Listenmaa, I. and Kaljurand, K. (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, LREC 2014, Reykjavík*.

Ljunglöf, P. (2004). *The Expressivity and Complexity of Grammatical Framework*. PhD thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University.

Magnusson, L. (1994). *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Milner, R., Tofte, M., and Harper, R. (1990). *Definition of Standard ML*. MIT Press.

Montague, R. (1974). *Formal Philosophy*. Yale University Press, New Haven. Collected papers edited by Richmond Thomason.

Ng'ang'a, W. (2012). Building swahili resource grammars for the grammatical framework. In Santos, D., Lindén, K., and Ng'ang'a, W., editors, *Shall We Play the Festschrift Game? Essays on the Occasion of Lauri Carlson's 60th Birthday*, pages 215–226. Springer, Berlin, Heidelberg.

Nilsson, M. (2022). Beginner's Somali Grammar. `http://morgannilsson.se/BeginnersSomaliGrammar.pdf`. Online; accessed 5 October 2022.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C. D., McDonald, R., Petrov, S., Pyysalo, S. m., Silveira, N., Tsarfaty, R., and Zeman, D. (2016). Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 1659–1666, Portorož, Slovenia. European Language Resources Association (ELRA).

Pease, A. (2011). *Ontology: A Practical Guide*. Articulate Software Press.

Pereira, F. and Shieber, S. (1987). *Prolog and Natural-Language Analysis*. CSLI, Stanford.

Pollard, C. and Sag, I. (1994). *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

Power, R. and Scott, D. (1998). Multilingual authoring using feedback texts. In *COLING-ACL*.

Pretorius, L., Marais, L., and Berg, A. (2017). A GF miniature resource grammar for Tswana: modelling the proper verb. *Language Resources and Evaluation*, 51(1):159–189.

Ranta, A. (1994). *Type Theoretical Grammar*. Oxford University Press.

Ranta, A. (2004). Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189.

Ranta, A. (2007). Modular Grammar Engineering in GF. *Research on Language and Computation*, 5:133–158.

Ranta, A. (2009). The GF Resource Grammar Library. *Linguistics in Language Technology*, 2.

Ranta, A. (2011a). *Grammatical Framework: Programming with Multilingual Grammars.* CSLI Publications, Stanford.

Ranta, A. (2011b). Translating between language and logic: What is easy and what is difficult. In Bjørner, N. and Sofronie-Stokkermans, V., editors, *Automated Deduction – CADE-23*, pages 5–25, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ranta, A. (2017). Explainable machine translation with interlingual trees as certificates. In Dobnik, S. and Lappin, S., editors, *Proceedings of the Conference on Logic and Machine Learning in Natural Language (LaML 2017)*, pages 63–78, Gothenburg.

Ranta, A. (2022). Multilingual text generation for abstract wikipedia in grammatical framework: Prospects and challenges. In Loukanova, R., Lumsdaine, P. L., and Muskens, R., editors, *Logic and Algorithms in Computational Linguistics 2021 (LACompLing2021)*, Cham. Springer Nature Switzerland. to appear.

Ranta, A., Angelov, K., Gruzitis, N., and Kolachina, P. (2020). Abstract Syntax as Interlingua: Scaling Up the Grammatical Framework from Controlled Languages to Robust Pipelines. *Computational Linguistics*, 46(2):425–486. `https://doi.org/10.1162/coli_a_00378`.

Ranta, A., Listenmaa, I., Soh, J., and Wong, M. W. (2022). An end-to-end pipeline from law text to logical formulas. In *Legal Knowledge and Information Systems, JURIX 2022*, Frontiers in Artificial Intelligence and Applications, pages 237–242, Saarbrücken, Germany.

Ranta, A., Tian, Y., and Qiao, H. (2015). Chinese in the Grammatical Framework: Grammar, Translation, and Other Applications. In *Proceedings of the Eighth SIGHAN Workshop on Chinese Language Processing, ACL*, pages 100–109, Beijing, China.

Saeed, J. (1999). *Somali.* John Benjamins Publishing Company.

Seki, H., Matsumura, T., Fujii, M., and Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Shieber, S. (1986). *An Introduction to Unification-Based Approaches to Grammars.* University of Chicago Press.

Steedman, M. (2000). *The Syntactic Process.* The MIT Press.

Virk, S. (2013). *Computational linguistics resources for Indo-Iranian languages.* PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology and Gothenburg University.

Vrabec, Ž. (2022). *Bosnian, Croatian, Montenegrin and Serbian: An Essential Grammar.* Routledge, London and New York.

Vrandečić, D. (2021). Building a Multilingual Wikipedia. *Communications of the ACM*, 64(4):38–41. `https://cacm.acm.org/magazines/2021/4/251343-building-a-multilingual-wikipedia/fulltext`.

Vrandečić, D. and Krötzsch, M. (2014). Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85.

Zimina, E. (2012). Fitting a Round Peg in a Square Hole: Japanese Resource Grammar in GF. In *JapTAL*, volume 7164, pages 156–167, Kanazawa. LNCS/LNAI.