LiLT volume 13, issue 3 October 2016

From Abstract Syntax to Universal Dependencies

Prasanth Kolachina and Aarne Ranta, University of Gothenburg

Abstract.

Abstract syntax is a semantic tree representation that lies between parse trees and logical forms. It abstracts away from word order and lexical items, but contains enough information to generate both surface strings and logical forms. Abstract syntax is commonly used in compilers as an intermediate between source and target languages. Grammatical Framework (GF) is a grammar formalism that generalizes the idea to natural languages, to capture cross-lingual generalizations and perform interlingual translation. As one of the main results, the GF Resource Grammar Library (GF-RGL) has implemented a shared abstract syntax for over 30 languages. Each language has its own set of concrete syntax rules (morphology and syntax), by which it can be generated from the abstract syntax and parsed into it.

This paper presents a conversion method from abstract syntax trees to dependency trees. The method is applied for converting GF-RGL trees to Universal Dependencies (UD), which uses a common set of labels for different languages. The correspondence between GF-RGL and UD turns out to be good, and the relatively few discrepancies give rise to interesting questions about universality. The conversion also has potential for practical applications: (1) it makes the GF parser usable as a rule-based dependency parser; (2) it enables bootstrapping UD treebanks from GF treebanks; (3) it defines formal criteria to assess the informal annotation schemes of UD; (4) it gives a method to check the consistency of manually annotated UD trees with respect to the annotation schemes; (5) it makes information from UD treebanks available

for the construction and ranking of GF trees, which can improve GF applications such as machine translation. The conversion is tested and evaluated by bootstrapping two small treebanks for 31 languages, as well as comparing a GF version of the English Penn treebank with the UD version.

1 Introduction

Computational syntax can work on different levels of abstraction. The lowest level normally used when processing written text is strings of tokens ("words"). But it is often useful to work with more abstract structures: part-of-speech (POS) tagged lemma sequences, phrase structure trees, dependency trees, or some kind of logical forms.

Raising the level of abstraction often gives new ways to relate different languages to each other. Thus tagged lemmas enable the separation of surface strings from word senses, which can be useful, for instance, in factored machine translation. Logical forms ideally ignore all language-related features, and express just the pure propositional meaning. But what about syntax trees? Traditional phrase structure trees preserve surface words and constituent order and are hence language-dependent. Dependency trees are often a bit more abstract, treating the word order as irrelevant and lemmatizing the words. Sharing the POS tags and dependency labels between languages increases this potential to abstract over languages.

Universal Dependencies (UD, de Marneffe et al. (2014)) is a recent approach to dependency parsing that tries to maximize the sharing of structures between languages. UD has a set of dependency labels and POS tags that are designed to fit many languages, and a series of annotation manuals that guide treebank builders to use the labels and tags in a uniform way. The expected gain is that efforts can be shared among languages. For instance, searching for semantic roles in sentences can be defined uniformly for different languages, and parsers for new languages can be bootstrapped by using treebanks for other languages.

As suggested by Nivre (2015), UD can be seen as a modern approach to **universal grammar**. The originally mediaeval idea of a universal grammar has many times been rejected by linguists, often with good reasons. But much of it can be saved if we think of it as an abstraction: on a proper level of abstraction, languages have much in common, so why not try to find out what is common? Universality should be seen as a working hypothesis rather than an *a priori* truth.

In UD, the working hypothesis is that languages have a common set

of parts of speech (nouns, verbs, etc) as well as grammatical functions (subject, modifier, etc). Some languages don't have all of these features, and individual languages may have features that are not universal. The annotation manuals for treebank builders have recommendations that maximize the use of common features, but give room to diversity. This approach has proved successful, and as a result, UD has presented treebanks for over 30 languages.

An older but nonetheless computational approach to universal grammar is Curry's notion of **tectogrammatical** structure (Curry, 1961). The tectogrammatical representations are function applications[1]. They are trees that describe **pure constituency**: what the constituents are and how they are put together, but ignoring what word strings are ultimately used and what their linear order is. To give an example, subject-verb-object predication could be presented by a tectogrammatical function `Pred`,

```
Pred : TV -> NP -> NP -> S
```

that takes a transitive verb and two noun phrases as its arguments and produces a sentence. The constituent order (SVO, SOV, etc) is specified separately for each language in their **phenogrammatical** rules. These rules may look as follows:

```
Pred verb subj obj = subj ++ verb ++ obj
Pred verb subj obj = subj ++ obj ++ verb
```

for SVO and SOV, respectively (with `++` marking concatenation).

Curry's tectogrammar inspired the Prague school of dependency parsing (Böhmová et al., 2003). The grammars, however, are different, since the Prague school is based on the roles of words (similar to what is traditionally called "grammatical functions"), whereas Curry's tectogrammatical functions are functions that combine words. Grammatical Framework (GF, Ranta (2004, 2011)) is a grammar formalism that is more directly based on Curry's architecture. GF grammars are similar to grammars used in compiler construction, where tectogrammar is called **abstract syntax** and phenogrammar is called **concrete syntax** or **linearization** (McCarthy, 1962, Appel, 1998). [2]

The most comprehensive multilingual grammar in GF is the **Resource Grammar Library**, GF-RGL (Ranta, 2009b), which by the time of writing has concrete syntaxes for over 30 languages, ranging

---

[1] Also known as lambda terms or LISP terms or, as in Curry's original work, terms of combinatory logic.

[2] As noted by Dowty (1979), also Montague grammar (Montague, 1974) can be seen as having Curry's architecture, although Montague only used it for English.

from European through Finno-Ugric and Semitic to East Asian languages.[3] When the UD approach appeared, it became immediately interesting to see how it relates to GF-RGL. The formal correspondence was obvious: once we have a GF abstract syntax tree, we can easily derive a dependency tree. For instance, a rule of the form

```
Pred verb subj obj
```

gives rise to a dependency tree where the first argument produces the head, the second argument produces a dependent with label `subj` and the third argument a dependent with label `obj`. As we will show more formally in Section 2.2, a simple recursive function can convert abstract trees to dependency trees in this way. However, there are details that remain to be worked out:

- How to convert GF-RGL to an independently given dependency scheme, such as UD?
- Is GF-RGL complete, in the sense of covering all UD structures?
- Can GF-RGL give any new insights for developing UD further?

The purpose of this paper is to answer these questions. While doing so, we will often discuss the differences in analyses between GF-RGL and UD, and in many cases argue for the GF-RGL decisions. But in a bigger picture, we have been surprised to see how much the approaches have in common. That so similar structures of "universal grammar" have been found in two independent ways can be seen as confirming evidence for both of them.

Our work is also expected to have practical uses: bootstrapping UD treebanks from GF; using UD treebanks to help GF parsing (in particular, statistical disambiguation); assessing UD annotation schemes and treebanks from a formal perspective. Our conversion moreover makes the GF-RGL parser usable as a rule-based UD parser, although a rather slow one. Perhaps more interestingly, generation from UD trees becomes possible (including translations to other languages), because GF grammars are reversible (and generation is fast, as opposed to parsing).

The structure of the paper is as follows: Section 2 prepares the discussion with a concise introduction to GF and a mathematical definition of the correspondence between abstract syntax trees and dependency trees. Section 3 gives an overview of GF-RGL and UD; parts of it can be skipped by the reader who already knows the approaches, and many of the details are given in an Appendix. Section 4 goes through the great majority of structures, where GF-RGL and UD are similar enough to

[3] The current status of GF-RGL can be seen in http://www.grammaticalframework.org/lib/doc/synopsis.html which also gives access to the source code.

allow a simple, local (i.e. compositional) and language-independent conversion of trees. Section 5 covers the remaining structures, where non-local or language-dependent conversions are needed. Section 6 presents an evaluation with three different treebanks. Section 7 concludes.

2 Grammars and trees

2.1 Abstract and concrete syntax

A GF grammar consists of an abstract syntax and a set of concrete syntaxes. Figure 1 shows a set of abstract syntax rules, which is a small fragment of the GF Resource Grammar Library, but representative in the sense that it covers some of the most fundamental syntactic structures. The rule set includes a lexicon that is large enough to cover our running example, the English sentence

the black cat sees us

and its French equivalent

le chat noir nous voit (word to word: "the cat black us sees")

An abstract syntax has two kinds of rules:

- `cat` rules defining **categories**, here `S`, `NP`, `VP`, etc.
- `fun` rules defining **functions**, here `PredVP`, `ComplTV`, etc.

All rules in Figure 1 are equipped with comments (starting `--`) that explain the categories and functions.

Categories are the basic building blocks of **types**, which have the form

$$C_1 \rightarrow \ldots \rightarrow C_n \rightarrow C$$

where $n \geq 0$ and $C_1, \ldots, C_n, C$ are categories. Each such type is a **function type**, where $C_1, \ldots, C_n$ are the **argument types** and C is the **value type**. The limiting case $n = 0$ gives types of **constant functions**, which typically correspond to **lexical items**, such as `we_Pron` in Figure 1. Types with $n > 1$ typically correspond to **syntactic combinations**, such as `PredVP`, combines an NP with a VP to an S. Types with $n = 1$ are typically **coercions**, such as `UsePron`, which lifts a pronoun into an NP.

A concrete syntax has two kinds of rules, parallel to `cat` and `fun` rules:

- for each category, a `lincat` rule defining its **linearization type**
- for each function, a `lin` rule defining its **linearization**, which is a function that combines the linearizations of the arguments into an object of the linearization type of the value type

To define a concrete syntax for Figure 1, we can start by uniformly

```
cat
  S ;     -- sentence
  NP ;    -- noun phrase
  VP ;    -- verb phrase
  TV ;    -- transitive verb
  AP ;    -- adjectival phrase
  CN ;    -- common noun
  Det ;   -- determiner
  Pron ;  -- personal pronoun

fun
  PredVP  : NP -> VP -> S ;   -- predication: (the cat)(sees us)
  ComplTV : TV -> NP -> VP ;  -- complementation: (see)(us)
  DetCN   : Det -> CN -> NP ; -- determination: (the)(cat)
  AdjCN   : AP -> CN -> CN ;  -- adjectival modification: (black)(cat)
  UsePron : Pron -> NP ;      -- use pronoun as noun phrase: (us)

  we_Pron : Pron ;  -- we/us
  see_TV  : TV ;    -- see/sees
  the_Det : Det ;   -- the
  black_AP: AP ;    -- black
  cat_CN  : CN ;    -- cat/cats
```

FIGURE 1 An abstract syntax for a fragment of GF-RGL.

using `Str` as linearization type:

```
lincat S, NP, VP, TV, AP, CN, Det = Str
```

All linearizations are then defined as strings and their concatenations (denoted by `++`). Thus a tree formed by the function `PredVP` is in both English and French linearized by

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

concatenating the linearization of the `NP` argument with the linearizations of the `VP` argument. But usually the rules are different. Thus lexical items have rules such as

```
lin black_AP = "black"  -- English
lin black_AP = "noir"   -- French
```

More interestingly, linearization rules can also vary the word order:

```
lin ComplTV tv np = tv ++ np  -- English
lin ComplTV tv np = np ++ tv  -- French

lin AdjCN ap cn   = ap ++ cn  -- English
lin AdjCN ap cn   = cn ++ ap  -- French
```

The concrete syntax rules shown above use only strings and their concatenation. Such rules have an expressive power similar to **synchronous context-free grammars** (Aho and Ullman, 1969), the main difference being that synchronous grammars don't make the abstract syntax explicit. Synchronous context-free grammars are sufficient for changing the lexical items and their order, which is what we need in our running example. However, the above rules are not correct, because they don't deal with morphology (case and agreement) nor with variable word order (clitic vs. non-clitic objects in French). To deal with natural languages in full scale while sharing the abstract syntax, we need a bit more expressive power. We will return to this in Section 2.3.

2.2 Trees and their conversions

Figure 2 summarizes the different kinds of trees that we will speak about, by showing different representations of one and the same example.

- **Abstract syntax trees** (a) are trees where the nodes and leaves are abstract syntax functions.
- **Parse trees** (b,c), also known as **concrete syntax trees** or **phrase structure trees**, are trees where the nodes are categories and the leaves are words (strings).
- **Dependency trees** (d,e) are trees where the nodes are words

and the edges are marked by **dependency labels**; the order of words is significant.

- **Abstract dependency trees** (f) are trees where the nodes are constant functions and the edges are marked by dependency labels; the order of nodes is not significant.

The tree visualizations are generated by GF software.

The abstract syntax tree (Figure 2 (a)) is a non-redundant representation from which all the others can be derived. Parse trees are derived as follows: given an abstract syntax tree T,

1. Linearize it to a word sequence S.
2. Link each word in S to its smallest spanning subtree in T.
3. Replace each function in the nodes of T by its value category.

The **smallest spanning subtree** of a word is the subtree whose top node is the function whose linearization generates that word.

To convert an abstract tree to a dependency tree, we specify, for each abstract syntax function, its **dependency configuration**: which of the arguments is the head, and how the other arguments are labelled. The default dependency configuration used in GF says that the head is the first argument, and the other arguments have labels `dep1`, `dep2`, and so on. This default can be overridden by an explicit configuration. In Figure 2, we assume the following configurations to produce the standard UD labels:

```
PredVP  nsubj head
ComplTV head  dobj
DetCN   det   head
AdjCN   amod  head
```

A similar configuration can be used for mapping GF categories to UD part of speech tags. The default is the category symbol itself.

Given an abstract syntax tree T of a word sequence S, a dependency tree is derived as follows:

1. For each word w in S, find the function f_w forming its smallest spanning subtree in T.
2. Link each word w in S with either
   (a) the head argument of f_w, if w is not the head
   (b) the head of whole f_w, if w is itself the head

Given a concrete syntax and a dependency configuration, an abstract syntax tree is thus a non-redundant and faithful representation for all information about a sentence. In contrast to this, parse trees and dependency trees are **lossy representations**. For dependency trees, this is easy to see: many functions can have the same dependency configuration, and if we only see the labels attached to the arguments, we

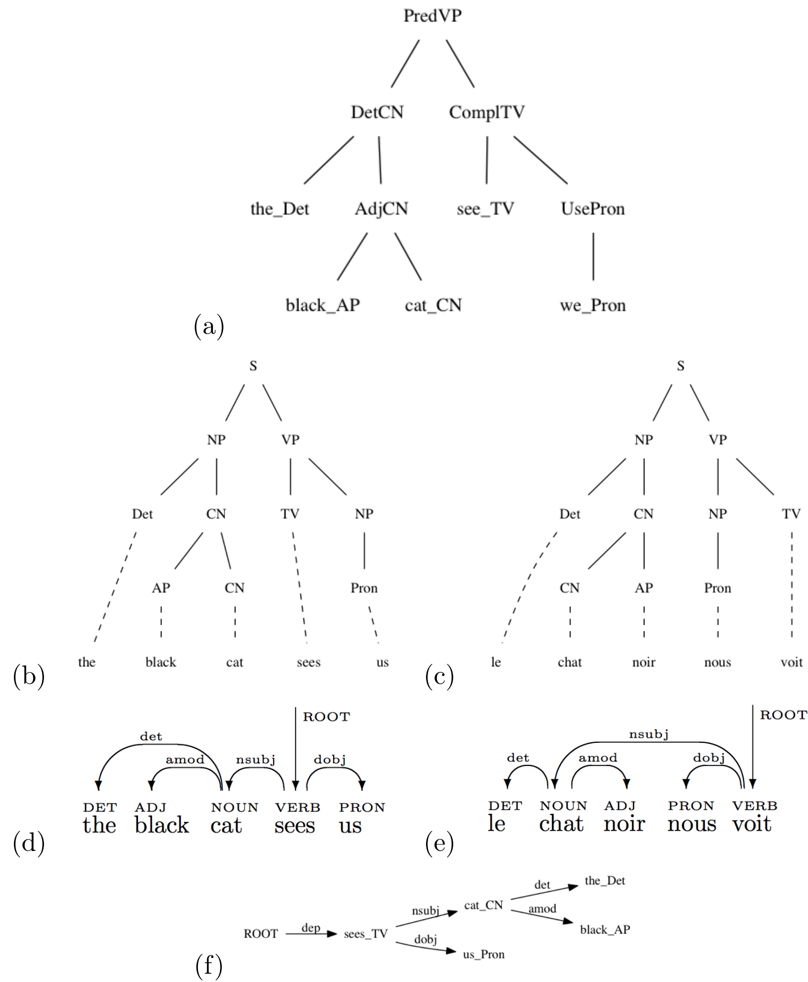

FIGURE 2 Trees for the sentence *the black cat sees us* and its French translation: (a) abstract syntax tree; (b,c) parse trees; (d,e) dependency trees; (f) abstract dependency tree with unordered word senses.

cannot know what the dominating function is. For parse trees, it can likewise happen that a context-free grammar rule encodes different ways of putting together its constituents. For instance, the flat predication rule

$$S \rightarrow \text{NP TV NP}$$

can, in a free word order language, match both SVO and OVS sequences.

For the reason of missing information, dependency trees and parse trees cannot in general be derived from each other. This is why the existing algorithms use uncertain heuristics such as head percolation (Collins, 1996). Using abstract syntax trees as the master representation solves this problem.

An alternative way to derive dependency trees is to use parse trees decorated with abstract syntax functions and dependency labels. This gives a natural way to explain the conversions and will therefore be used later in this paper. In a decorated parse tree, we mark the dependency labels at each branching point of the tree, as shown in Figure 3, but omit the "head" labels. To find the labelled arc for each word,

1. Follow edges up from the word until a label is reached: this is the label of the word.
2. From the dominating node, follow the (unique) path of unlabelled edges down to another word: this is the head of the word. A head path in a tree is called a **spine**.
3. If no label is encountered on the way upwards, the word is itself the head of the sentence.

Figure 3 shows the path corresponding to the labelled arc of the word *cat*.

2.3 Abstracting from morphological variation

If we swap the subject and the object in the example sentence of Figure 2 and linearize with the concatenation rules of concrete syntax in Section 2.1, we get *us sees the black cat* in English and *nous le chat noir voit* in French. Both sentences have a subject-verb agreement error. The English sentence also has a wrong case of the pronoun, and the French sentence has a wrong word order, since the object can appear before the verb only if it is a clitic pronoun. To solve these problems, we need to introduce morphological variation in the grammar. Since morphological variation is language-dependent, we introduce it in concrete syntax and not in the abstract syntax. This forces us to go beyond the context-free linearization rules of Section 2.1.

Let us consider verb inflection first, restricted to present tense indicative forms for simplicity. In English, we need two forms: the third

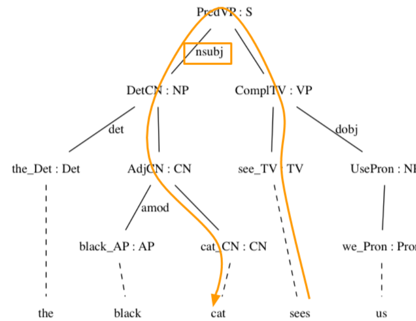

FIGURE 3 Dependency tree derivation from a decorated parse tree. The word *cat* has *sees* as its head and *nsubj* as its label. The path from the top to the word *sees* is a spine.

person singular and "other". We define, in the English concrete syntax, a **parameter type** of verb forms, which has two elements:

```
param VForm = SgP3 | Other
```

The category TV (and also VP) has as its linearization type a **table** (similar to an inflection table), which produces a string as a function of a verb form:

```
lincat VP, TV = VForm => Str
```

Thus the verb *see* is linearized as follows:

```
lin TV = table {SgP3 => "sees" ; Other => "see"}
```

For noun phrases, we need the parameter of case, which is nominative or accusative.

```
param Case = Nom | Acc
```

But we also need to account for the subject-verb agreement: the fact that a noun phrase can determine the form of a verb. This we can do by using a **record type** as the linearization type of NP:

```
lincat NP = {s : Case => Str ; a : VForm} ;
```

A record of this type has two fields: the field `s`, which is a case-dependent string, and the field `a` (agreement feature), which is a verb form. An example is the linearization of `we_Pron`:

```
lin we_Pron =
  {s = table {Nom => "we" ; Acc => "us"} ; a = Other}
```

Putting everything together, we obtain the linearization rule for predication:

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

This rule uses the `s` field of the NP (by the projection operator `.`), from which it takes the `Nom` form (by the selection operator `!`). The result is concatenated to the verb form selected for the value for the `a` field of the NP.

In French, we need different parameter types: for instance, verbs have six forms and not just two. We also need some parameters not present in English, such as the gender of adjectives, nouns, and determiners. The most interesting parameter for the example at hand is, however, a boolean that states if an NP is a clitic, to determine its position when used as object:

```
lincat NP =
  {s : Case => Str ; a : VForm ; isClitic : Bool}
```

Now we can write a complementation rule that inspects the cliticity feature of the object to decide if the verb or the object comes first:

```
lin Compl tv np = table {vf =>
  case np.isClitic of {
    True  => np.s ! Acc ++ tv ! vf ;
    False => tv ! vf ++ np.s ! Acc
    }
  }
```

The generalization of linearization from strings to tables and records leads us from context-free grammars to **multiple context free grammars** (MCFG) (Seki et al., 1991). As shown in Ljunglöf (2004), GF is actually equivalent to PMCFG (Parallel MCFG), which is MCFG with reduplication. [4]

An MCFG is a grammar over tuples of strings rather than just strings. In addition to inflection and word order variations, MCFG enables **discontinuous constituents**. Thus for instance in VSO languages (such as classical Arabic) verb phrases are records with separate fields for the verb and the complement:

```
lincat VP = {verb : Str ; compl : Str}
```

(ignoring all morphological parameters). The VSO order is realized in the predication rule, which puts the subject noun phrase between the verb and the complement:

[4] Reduplication is used only in few places in the RGL: Chinese yes/no questions and some semantic constructions such as the intensification of adjectives in Swedish.

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

2.4 Abstracting from syncategorematic words

Designing a GF grammar involves finding a level of abstraction that makes sense for all languages to be addressed. For instance, morphological distinctions present in one language but not the others should be ignored in the abstract syntax. Some of these questions can be subtle. The **copula** of adjectival predication is an example of a common kind of questions encountered in the RGL-UD mapping task.

Let us extend the abstract syntax of Figure 1 with a rule converting adjectival phrases to verb phrases:

```
fun CompAP : AP -> VP
```

The English linearization rule (ignoring morphology, which is irrelevant for this question) introduces the copula *is* prefixed to the AP:

```
lin CompAP ap = "is" ++ AP
```

The copula has thus no abstract syntax of its own. The cross-linguistic justification of this treatment is that many languages (Arabic, Russian) don't need copulas, and that they should therefore not be introduced in the abstract syntax. Words with no abstract syntax category attached are called **syncategorematic**.

The conversion defined in Section 2.2 generates no dependency labels for syncategorematic words. Thus the tree assigned to *the cat is black* in this grammar has no label for the word *is*. A default dummy label "dep" can then be used, as in Figure 4 (a). This tree moreover uses VP as the part of speech tag, since this is the category of the smallest spanning subtree of the copula.

However, the UD annotation manual for English says that the copula should have the label "cop" with the word *black* as its head (Figure 4 (b)) and AUX as its POS tag. The general principle in dependency parsing is that all words have labels connecting them to a head (except for the head of the whole sentence). According to this principle, *there are no syncategorematic words*.

The principle that all words are categorized makes sense in the dependency parsing context, where the trees are trees of *words* and not of phrases, let alone abstract functions. But from the "universal" point of view, this results in trees that may be unnecessarily different in different languages, because syncategorematic words are not universal. They can also be argued to be irrelevant for the semantic structure. Thus dependency parsers are sometimes supplemented by "flattening" or "collapsing" functions that remove semantically irrelevant words (de Marneffe and Manning, 2008, Ruppert et al., 2015). The "collapsing" process

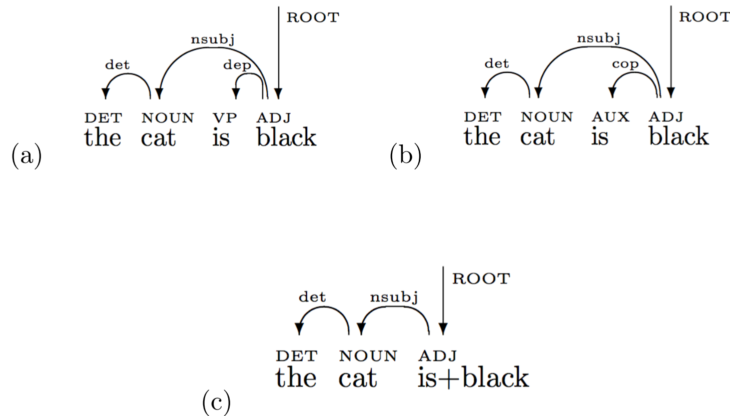

FIGURE 4 Dependency trees for *the cat is black* (a) with a syncategorematic copula (b) with a categorized copula. Also shown in (c) is a collapsed variant of (a)

achieves two characteristics: a collapsed tree no longer contains all the words in a sentence and the dependency labels and heads are semantically coherent (but may be syntactically heterogeneous).

When dependency trees are derived from abstract syntax, the situation is the opposite. What we get first, by straightforward dependency configurations, is a kind of "collapsed" trees (Figure 4 (c)). The semantically irrelevant words are not provided with a dependency label of their own using dependency configurations defined on abstract syntax. To obtain these labels, we must extend the dependency configurations with rules defined on concrete syntax. Such rules are language-dependent, not universal.

One could question whether not just stop at the collapsed trees, since they are the truly universal ones. However, since we are interested in converting GF trees to complete UD trees, we have chosen to extend our conversion algorithm with a "decollapsing" phase, by extending the abstract (language-independent) dependency configurations with language-dependent ones (Section 5). The resulting dependency tree at the end of this "decollapsing" phase is a connected tree where all words have labels connecting them to a head, as done in the UD scheme. The extended conversion algorithm is presented in Section 5.7.

An alternative to these language-dependent configurations is to rewrite the grammar. Rewriting the grammars facilitates constructing complete UD trees using dependency configurations defined only on the

abstract syntax. For the example at hand, this is simple: just introduce a category `Cop` of copulas, with one element `be_Cop`, and change the `CompAP` rule:

```
cat Cop
fun be_Cop : Cop
fun CompAP : Cop -> AP -> VP
```

In languages with zero copulas, the linearization of `be_Cop` is the empty string.

The line that we follow in this paper, however, is to keep GF-RGL as it is and instead make the dependency configurations more elaborate. This choice has several advantages:

- We can automatically get collapsed trees. These can be post-processed later to add labels for syncategorematic words, but still be useful if the end goal is only to extract relations between the content words in the sentence.
- We don't optimize for the particular dependency annotation scheme of UD and can easily change the configurations.
- Since GF-RGL was from the start designed to be multilingual, we get evidence to assess the universality of the current UD.

3 An overview of GF-RGL and UD

This section provides readers with an overview of GF-RGL and the UD annotation project. Readers familiar with GF can skip Section 3.1 and readers familiar with UD can skip Section 3.2.

3.1 Overview of RGL

The first applications of GF were small grammars built for translation systems on specific application domains, such as geographic facts (Dymetman et al., 2000), mathematics (Hallgren and Ranta, 2000), and simple health-care dialogues (Khegai, 2006). The abstract syntax in these applications encoded the semantic structures that were to be preserved in translation. But the idea soon emerged to generalize the abstract syntax idea from domain semantics to domain-independent syntactic structures, such as NP-VP predication. This led to the development of the GF Resource Grammar Library (RGL), whose first version was inspired by the syntactic structures used in CLE (Core Language Engine) (Rayner et al., 2000). The RGL was first intended to be a library that would help domain grammarians by giving them reusable functions for surface syntax and morphology (Ranta, 2009a). But it was soon also seen as a linguistic experiment, to find out how comprehensive a common abstract syntax could be and how many languages it

could apply to. This experiment had no commitment to any theory of linguistic universals, but the idea was just to try and see how far one can get. Nevertheless, the sharing of tree structures was more far reaching than parallel grammar development in systems like CLE, LinGO Matrix (Bender and Flickinger, 2005), and ParGram (Butt et al., 2002).

The first experiments on a handful of not too similar languages (English, Finnish, French, Russian) were encouraging. Gradual modifications led to a stable abstract syntax, which was reported in Ranta (2009b) and was at the time implemented for 14 languages. In late 2015, the same abstract syntax has concrete syntaxes for 30 languages, and 5 to 10 more are under construction. More than 50 persons have contributed to the GF-RGL. Its source code and documentation are available on the GF web page.[5]

The GF Resource Grammar Library has an abstract syntax with 86 categories and 356 functions. Of these functions, 140 are functions that don't take arguments (mostly structural words such as determiners), 85 are one-argument coercions, and 131 are syntactic combinations with more than one argument (the only class that needs dependency configurations). This abstract syntax is the **core RGL**, which is specified as the minimum to be implemented for a language to count as a "complete" resource grammar. This notion of completeness is of course purely formal, and does not mean that the whole language is analysable by the grammar. But it does mean that the standard GF applications, such as controlled language grammars (Ranta et al., 2012, Dannélls et al., 2012, Kaljurand and Kuhn, 2013) are directly portable to that language.

In addition to the core RGL, the library has **language-specific extensions**, which need not be shared by all languages. These extensions are grouped hierarchically, so that for instance Romance languages have a set of common extensions, on top of which Catalan, French, Italian, and Spanish have their own extensions. These modules typically make a 10% addition to the core RGL code. The extension modules enable grammarians to experiment with individual languages without bothering about the universality of all constructs. There is after all no reason why all grammatical constructs should be universal: it is good enough if a core subset is.

Another, recent addition to the core RGL is a set of categories and functions that enable wide coverage parsing and translation (Angelov et al., 2014). To maintain the interlingual translation architecture, these extensions are shared by all languages (15 at the time of writing). But

[5] http://www.grammaticalframework.org/lib/doc/synopsis.html

they are generally less precise than the original RGL functions. In particular, there is a category of **chunks**, which is used as a robust back-up to syntactically complete parsing.

The present paper focuses on the core RGL, showing how the main syntactic structures of GF are mapped into UD. This is sufficient for two small treebanks, which are covered for 31 languages. Dealing with a larger treebank, covering 15 languages, also includes the robust back-up rules. But this part is less stable and more ad hoc, and we will report the result for the two parts separately.

Figure 18 in the Appendix shows the hierarchy of categories in the core RGL. The same picture appears in (Ranta, 2011), and Ranta (2009b) provides a systematic linguistic discussion of the categories and functions. In this paper, we will present the RGL from another perspective, showing how the functions are decorated by UD labels to convert them to dependency trees. Table 11 in the Appendix lists the RGL categories with explanations, examples, and corresponding UD POS tags.

Let us walk through an example, which shows many of the main functions in use. Figure 5 shows an abstract syntax tree for the sentence *my two brothers and I would not have bought that red car* and its RGL equivalents in 29 other languages, artificially constructed to show as many structures as possible. The tree is decorated with UD dependency labels.

The topmost category in Figure 5 is Utt, utterances, which is built from S, sentence; an Utt could also be built from a question or an imperative, as shown in Figure 18. The sentence in turn is built from a clause (Cl) by adding temporal and polarity features: conditional anterior negative. The core RGL has 16 tense-polarity combinations for clauses. The clause is built from a noun phrase (NP) and a verb phrase (VP).

The subject noun phrase is a coordination, built from a list of noun phrases (ListNP) with a conjunction (Conj). The list is a recursive structure, with the base case taking two elements. Longer lists are (in English and many other languages) linearized by using commas, for instance, *her mother, my brother and I.*

The first conjunct of the subject is built from a determiner (Det) and a common noun (CN). The determiner is further analysed into the head quantifier (Quant) and a number (Num). The quantifier is a pronoun (Pron) used possessively, and the number is a cardinal numeral. The second conjunct is just a pronoun.

The verb phrase is built from a slash verb phrase (VPSlash), by providing an object noun phrase; VPSlash is similar to a "slash category"

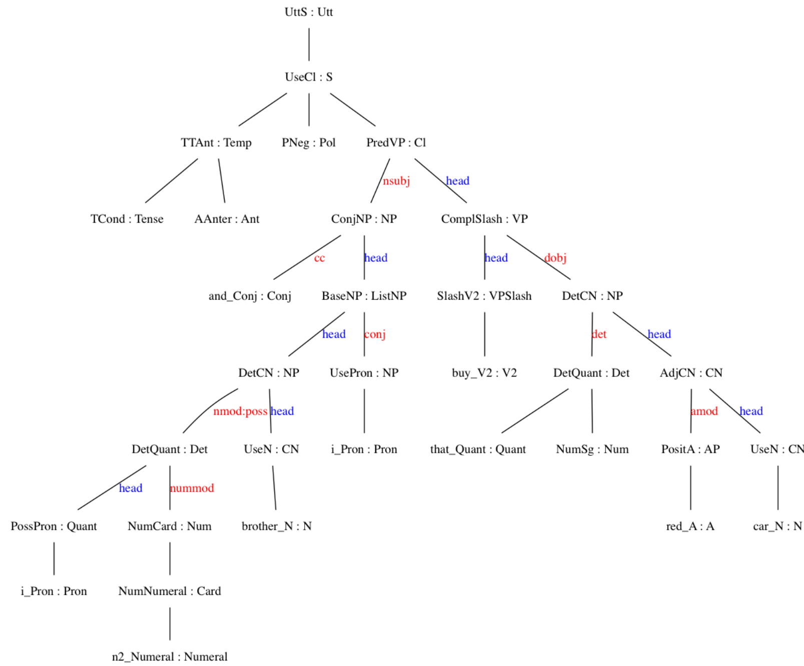

FIGURE 5 Dependency-decorated abstract syntax tree for *my two brothers and I would not have bought that red car.*

(VP/NP in the notation of Gazdar et al. (1985)). The VPSlash is here just a two-place verb (V2). V2 is a generalization of TV and covers both transitive and prepositional verbs, as well as verbs taking different complement cases in different languages. A verb being transitive is not a multilingual invariant and therefore not maintained in the abstract syntax. The complement NP is built from a Det and a CN that has an AP modifier. The determiner has a dummy number (NumSg) indicating that the quantifier `that_Quant` is used in the singular form.

As we can see from the tree, most of the dependency labels are straightforward to define, since the words ultimately appear as categorized lexical items. But some of them need special attention, in particular the tense and polarity features, which in English are realized syncategorematically as auxiliary verbs (discussed in Section 5.4). Also coordination is worth discussion (Section 4.5), as it is a perennial question in dependency parsing.

3.2 Overview of UD

The UD annotation scheme defines a taxonomy of 40 relations as the core or *universal* dependency label set (shown in Table 12 in the Appendix). This taxonomy is further refined (or reduced) to address language-specific extensions, not necessarily shared by all languages. The scheme also defines universal sets of part-of-speech tags and morphological features (shown in Table 13 in the Appendix). The part-of-speech tagset includes 17 tags, with extensions to previously proposed Universal Part-of-Speech tagset (Petrov et al., 2012). The project is an effort to **consistently** annotate multilingual corpora with these morphological features, part-of-speech tags and universal labels in the dependency parse tree. Figure 6 shows all three layers of UD annotation for a French sentence *Toutefois les filles adorent les desserts au chocolat* (trans. "However girls love chocolate desserts"; example and figure quoted verbatim from Nivre et al. (2016)).

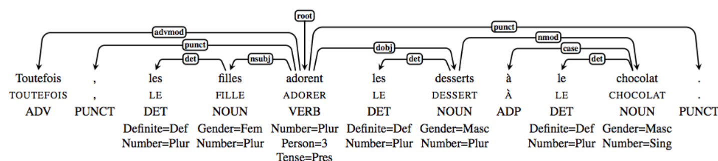

FIGURE 6 UD annotation for a French sentence (lemmas are capitalized). Figure quoted verbatim from Nivre et al. (2016).

Core arguments of clauses are marked as either a subject (`nsubj` in the case of nominal subjects and `csubj` for clausal subjects) or direct/indirect objects depending on their grammatical function. An example is shown in Figure 7. In the case of direct objects, distinction between nominal arguments (`dobj`), finite clausal arguments (`ccomp`) and open clausal arguments (`xcomp`) is made. Furthermore, in the case of passive constructions, separate labels are used to mark subjects (`nsubjpass` and `csubjpass`) to indicate transformation of voice. We will look at these constructions separately in Section 5.

Non-core dependents of clausal predicates like prepositional phrases attached to the verb are annotated as nominal dependents using `nmod` label or adverbial modifiers (`advmod` for adverbs and `advcl` for clausal dependents). Other labels used for non-core dependents in a clause are `neg` to mark negation of predicates (and also noun phrases), `vocative` to mark vocative noun phrases and `expl` for expletives. Expletives are nominals that appear with labels for the core arguments in a clause,

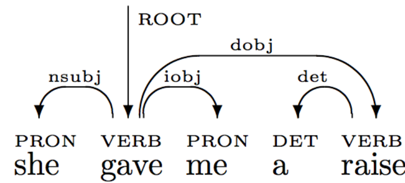

FIGURE 7 UD dependency labels for arguments in clauses

but have no semantic significance by themselves (for example, there is an expletive in *there is a cat in the house*). More frequently found labels in annotated corpora from this class are `aux` for auxiliary verbs, `auxpass` for auxiliary verbs in passive voice constructions and `cop` for copula verbs. The mapping for auxiliary verbs and copulas is discussed in Section 5.

An example of language-specific extensions defined in the UD annotation scheme are labels used to annotate these non-core nominal dependents. In English, the `nmod` label is refined to indicate temporal adverbs `nmod:tmod`, possessive noun modifiers `nmod:poss` and noun phrases used as adverbials `nmod:npmod`. In case of Swedish, the `nmod` is further refined to indicate agents used in passive voice constructions `nmod:agent` and the same possessive noun modifiers as in English. The annotation scheme allows fine-grained extensions to a label, but the choice of annotating the extension is left to the annotators of the language. The guidelines for a specific language provides details on when to annotate these extensions in the language. The mapping we discuss in this paper will ignore these fine-grained extensions, since the goal is to map the core RGL to the core label set. But it is possible to add the fine-grained distinctions to the mapping if we are interested in independently analysing each language. In our own experiments, the extensions to nominal dependents for possessive noun modifiers, and noun phrase adverbials are defined in a separate mapping on the abstract syntax. This mapping is defined only for English. Similarly, the extension in Swedish for agents in passive constructions is equally straight forward to define.

When annotating noun phrases, dependents are typically labelled using either `nummod` for numerical modifiers, `appos` for appositions or a generic `nmod` label for all other nominal modifiers like prepositional phrases. There is a `det` label for determiners and an `amod` label for adjectival modifiers. Relative clauses that modify the noun are marked using the `acl` label (*that we saw* in the NP *the children that we saw*).

Prepositions are always marked as modifiers using the `case` label.

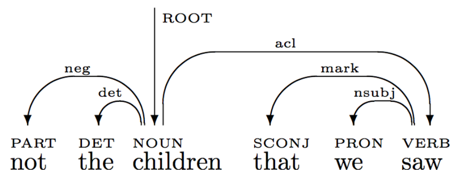

FIGURE 8 UD labels for modifiers in NPs

Other labels defined in the core label set include `cc` and `conj` for coordination constructions (an example of flat-structure provided for coordinations is shown in Figure 9), `compound` for compounding, `mwe` to treat multi-word expressions and a loosely defined `goeswith` label for robust analysis of web and raw texts. Annotation strategies for these specific labels will be clearly explained when we discuss the mapping between the GF-RGL for these phenomena. In this paper, the labels used for defining these loose joining relations will not receive critical attention.

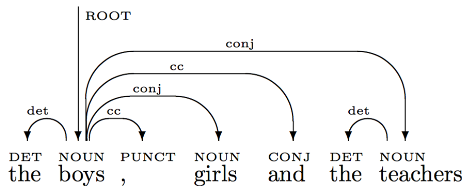

FIGURE 9 UD labels for a coordination of 3 NPs

4 Dependency mappings: straightforward cases

In the general case, we define the mapping between functions in the core RGL and the universal UD labels using dependency configurations to UD labels. The mapping to the core UD labels allows bootstrapping treebanks for new languages using the abstract syntax defined in GF-RGL. Additionally, we also define a fine-grained mapping over these functions to derive language-specific UD labels defined in the annotation scheme. Throughout this paper, we will mainly describe the mapping to the core UD labels.

4.1 Clausal predicates: predication and complementation

We will start by detailing the mappings for functions in the GF-RGL library used to build declarative clauses. The example shown in Figure 5 discussed some of these functions. Table 1 shows the complete set of functions used to construct these clauses, the argument types and value type of the resulting phrase and the dependency configuration for each of these functions.

The `PredVP` and `PredSCVP` functions are the main functions responsible for predication. The interpretation of the type signature shown in column 2 of the table is as follows- the `PredVP` function takes a noun phrase (NP) and a verb phrase (VP) and constructs a clause (Cl). Similarly, the `PredSCVP` function takes an embedded clause SC and a verb phrase and constructs a clause. Figure 10 shows the parse trees for the example sentences, *John killed him* and *that she came will make news*. We map the noun phrase in the `PredVP` function to the `nsubj` label, and in the case of `PredSCVP`, the embedded clause is mapped to the `csubj` label.

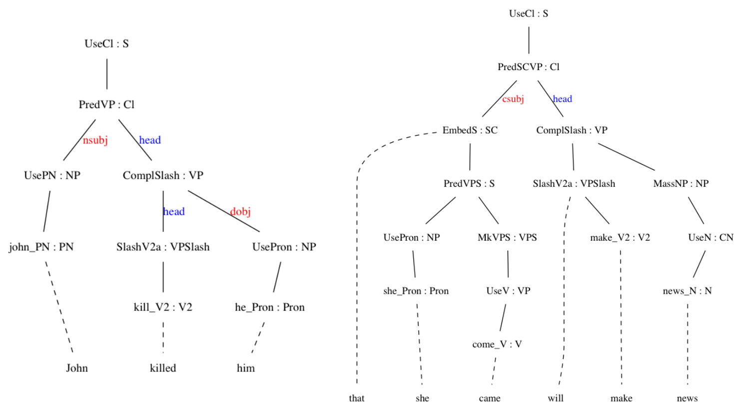

FIGURE 10 Decorated parse trees showing nominal and clausal predication in GF. The parse trees shows the abstract function names, the category of each node and the UD labels.

VP are created using the `ComplSlash` that takes a VPSlash type phrase and NP phrase as a core argument. Alternative complementation functions are `ComplVS` for clausal arguments S (*he says that you want to swim*), `ComplVV` for verb phrase complements (*want to swim*) and `ComplVA` for adjectival phrase complements (*feel bad*). In these cases, we map the NP phrase using a `dobj` label and the clause S using

the `ccomp` label. The complement arguments of `ComplVV` and `ComplVA` functions are mapped using the `xcomp` label. In all complementation functions, the arguments with different verb types (VPSlash, VS, VV, VA) are marked as the head of the phrases.

Verbs that take sentential arguments VS in the GF-RGL library are used in the `ComplVS` function to create a VP. The clausal argument (S), is mapped to the `ccomp` label. Alternatively, the RGL also defines a `ComplVQ` function for verbs that takes question clauses as arguments (*do you know who did it*), here also we map the clausal argument to the `ccomp` label.

Similarly, `SlashVP` and `SlashVS` functions are alternate ways to combine an NP with a VPSlash (verb phrase missing an NP), or ClSlash (clause missing an NP). These types of phrases are used to create question clauses and relative clauses. In both these functions, we map the arguments to `nsubj` and `nsubj` and `ccomp` respectively.

Passive voice constructions in GF are created using a type-raising function to create the VP phrase, by dropping one of the mandatory arguments. In the case of transitive verbs and ditransitive verbs, the VPSlash type is converted into a VP type without any additional NPs. The predication functions `PredVP` and `PredSCVP` remain the same since the voice is localized in the sub-tree corresponding to the VP. In order to distinguish the `nsubj` and `nsubjpass` labels in the case of passive voice, we extend our dependency configuration rules defined on the abstract syntax. We describe the mapping for passive constructions in Section 5.1.

PredVP	NP -> VP -> Cl	nsubj head	*John walks*
PredSCVP	SC -> VP -> Cl	csubj head	*that she came will make news*
ComplSlash	VPSlash -> NP -> VP	head dobj	*love it*
ComplVS	VS -> S -> VP	head ccomp	*say that she runs*
ComplVQ	VQ -> QS -> VP	head ccomp	*wonder who runs*
ComplVV	VV -> VP -> VP	head xcomp	*want to sleep*
ComplVA	VA -> AP -> VP	head xcomp	*become red*
SlashVP	NP -> VPSlash -> ClSlash	nsubj head	*(whom) he sees*
SlashVS	NP -> VS -> SSlash -> ClSlash	nsubj head ccomp	*(who) he says that she loves*

TABLE 1 UD mappings for declarative clauses

It is worth mentioning here that the entire mapping of the UD labels is encoded in a declarative fashion, exactly as shown in Table 1. Our mappings are the first and the third columns in this table. The other two columns are shown for convenience. The declarative style of specification allows for the conversion algorithm from GF-RGL trees to remain independent of the annotation scheme, allowing one to easily switch between different annotation schemes.

4.2 Adverbial modifiers

Adverbial phrases that modify VP phrases are analysed using either `AdVVP` function that takes a VP phrase and a AdV phrase (*always sleep*) or `AdVVPSlash` that modifies a VPSlash instead of a VP phrase. Alternatively, simple adverbs can modify a VP phrase using the `AdvVP` function (*sleep here*). For all these functions, we map the head of the adverbial phrase using the `advmod` label. Table 2 lists the functions used to modify VP phrases.

AdVVP	AdV -> VP -> VP	advmod head	*always sleep*
AdVVPSlash	AdV -> VPSlash -> VPSlash	advmod head	*always use (something)*
AdvVP	VP -> Adv -> VP	head advmod	*sleep here*
AdvVPSlash	Adv -> VPSlash -> VPSlash	advmod head	*use (something) here*
AdvS	Adv -> S -> S	advmod head	*then I will go home*
AdAP	AdA -> AP -> AP	advmod head	*very warm*

TABLE 2 UD mappings for adverbial modifiers

4.3 Questions and relative clauses

The GF-RGL provides a module for questions combining interrogatives (IP, IComp, IAdv) with verb phrases (VP, VPSlash) and clauses (Cl, ClSlash). For example, the function `QuestIAdv` takes a clause like *John walks* and a pronoun like *why* to construct the question *why does John walk*. Similarly, relative clauses are constructed using one of NP, VP or ClSlash types. Table 3 shows the mappings defined for functions used in constructing question and relative clauses.

4.4 Noun phrases and modifiers

The GF-RGL provides two primary kinds of functions to create noun phrases. Functions used to generate noun phrases and phrases with adjectival modifiers and functions used to modify these noun phrases using prepositional phrases, appositional NPs and relative clauses. We will present these two groups separately. There are in all two different categories for nouns defined in the RGL, CN for the basic nouns, N2

QuestIAdv	IAdv -> Cl -> QCl	advmod head	*why does John walk*
QuestIComp	IComp -> NP -> QCl	head nsubj	*where is John*
QuestVP	IP -> VP -> QCl	nsubj head	*who walks*
QuestSlash	IP -> ClSlash -> QCl	dobj head	*who does John love*
QuestQVP	IP -> QVP -> QCl	nsubj head	*who buys what he is selling*
RelSlash	RP -> ClSlash -> RCl	mark head	*whom John loves*
RelVP	RP -> VP -> RCl	who loves John	*mark head*

TABLE 3 UD mappings for questions and relative clauses

for nouns that take a noun complement (*mother of king, list of names*). The primary `DetCN` function takes a determiner and a CN to create a NP (*the boy, this paper*). N2 type nouns take an NP complement phrase to construct a CN. In the `DetCN` function, we map the determiner to the `det` label and make the CN argument head of the NP phrase. In `ComplN2` and `ComplN3` functions, NP arguments are mapped as modifiers of the noun using the `nmod` label (*names* is marked as the child of *list* with `nmod`). Alternative ways to generate NP phrases use `CNIntNP` for phrases like *level 5*, `CNNumNP` for *level five*; in both cases we map the Int/Card argument to the `nummod` label. The `AdjCN` function is used to modify nouns; these nouns still need a Det argument to become noun phrases (*the blue house*). Here, the head of the adjective phrase AP is mapped to the label. Table 4 lists the functions used to construct the basic NP phrases and the respective mappings.

DetCN	Det -> CN -> NP	det head	*the man*
ComplN2	N2 -> NP -> CN	head nmod	*mother of the king*
CNIntNP	CN -> Int -> NP	head nummod	*level 5*
CNNumNP	CN -> Card -> NP	head nummod	*level five*
AdjCN	AP -> CN -> CN	amod head	*big house*
DetQuant	Quant -> Num -> Det	det head	*these five*
DetQuantOrd	Quant -> Num -> Ord -> Det	det head amod	*these five best*

TABLE 4 UD mappings for generating basic noun phrases

The Det type in GF is used for determiners, typically constructed by taking a Quant type and Num type. For the definite article *the*, the analysis would be

```
the:  DetQuant (DefArt) (NumSg)
the:  DetQuant (DefArt) (NumPl)
```

This Det type and the `DetCN` is used to construct determiner phrases like *these five* that can modify a noun or act as noun phrases by themselves. An alternative function to generate the Det type is `DetQuantOrd` using which phrases like *these five best* can be analysed. In both these cases, we map the Quant type as the head of the phrase and the Num type is mapped using the `nummod` label. As such in phrases like *these boys* where there are no numerical modifiers in the phrase, we obtain the same structure as in UD i.e. *these* is given the `det` label and *boys* is the head. However in phrases with numerical modifiers (for example *these five boys*), the numerical modifier *five* is attached to the quantifier *these* and not to the noun *boys*.

In addition to these functions, NPs can be modified using other NPs for apposition, prepositional phrases. The list of functions is shown in

Table 5. `ApposCN` is used for apposition (*Sam, my brother*) and `PossNP` for possessive nominal modifier constructions (*Marie's book*), `PartNP` for partitive constructions (*glass of wine*). The modifier NP phrases are mapped to `appos`, and `nmod` labels. In the `PossNP`, the noun representing the possessive modifier is assigned the `nmod` (or `nmod:poss` for English) label where as in the `PartNP` function, it is the opposite.

ApposCN	CN -> NP -> NP	head appos	*city Paris*
PossNP	CN -> NP -> CN	head nmod	*Marie's brother*
PartNP	CN -> NP -> CN	head nmod	*glass of wine*
ComparA	A -> NP -> AP	amod head	*warmer than I*
RelCN	CN -> RelS -> CN	head acl	*house that John bought*
RelNP	NP -> RelS -> NP	head acl	*Paris, which is beautiful*

TABLE 5 UD mappings for modifying noun phrases

The `RelCN` and `RelNP` functions are used to modify nouns and noun phrases with relative clauses. In this case, we map the head of the relative clausal predicate using the `acl` label.

4.5 Coordination

The RGL defines multiple functions for building coordination constructions by combining lists of phrases (two or more) in the same category using conjunctions. `Base*` functions accepts two phrases of same type and create a tree of type List*. The List* types are combined with conjunctions (marked as Conj) using `Conj*` functions to form trees corresponding to coordination constructions. Additionally, `Cons*` functions recursively add a new phrase to List* phrases. These `Cons*` functions are used when coordinating more than two phrases of a given type. The `Conj*` functions handle both syndetic and asyndetic coordination constructions in all languages. Syndetic coordination is a form of coordination with the help of an explicit coordinating conjunction (*ham and eggs*) while asyndetic coordination allows for conjunctions to be omitted from one or all the conjuncts in the coordination construction (*he came, saw and conquered*).

Let us look at an example of adverbial coordination using the phrase *here, there and everywhere*. The parse tree for this example is shown in Figure 11. Adverbials *there* and *everywhere* are first analysed using the `BaseAdv` function. The `ConsAdv` function adds *here* to the list of adverbs from the `BaseAdv` function. This resulting ListAdv phrase is used by the `ConjAdv` function along with the conjunction *and* to form the AST for the coordinated phrase.

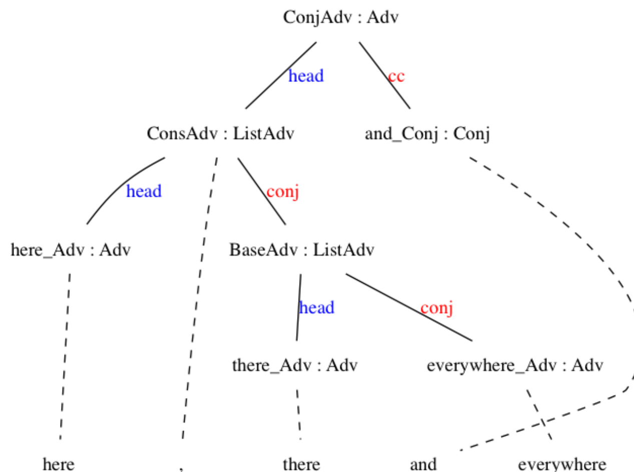

FIGURE 11 Coordination of three adverbials

The existing UD scheme annotates the first conjunct in the construction as the head, and treats the rest of the conjuncts as modifiers attached directly to the head via the `conj` relation. Similarly, conjunctions are attached to the same head via the `cc` relation. Table 6 shows the mapping defined for coordinating any two types in the GF-RGL. Note that when coordinating more than two phrases, the first conjunct always comes from the argument of the `ConsNP` function. The same mapping is defined for the set of these three functions defined for all types in GF-RGL (complex nouns CN, noun phrases NP, adjectival phrases AP, adverbial phrases AdV, simple clauses S and relative clauses RS).

BaseT	T -> T -> ListT	head conj
ConsT	T -> ListT -> ListT	head conj
ConjT	Conj -> ListT -> T	cc head

TABLE 6 UD mappings for generic type T in GF-RGL. T can be CN, NP, AP, AdV, S and RS

5 Dependency mappings: Problematic cases

The mappings described in the previous section are straightforward rules defined over the abstract syntax in GF-RGL. When the equiv-

alent labelled dependency tree for a given phrase can be completely identified from the AST by defining a mapping to dependency labels for the functions over its **immediate** arguments, then such a mapping is sufficient to construct a fully connected dependency tree from the AST.

This is not necessarily the case. Using only rules described in the previous section, the mappings can result in a dependency tree with edges labelled using the `dep` label. The main reason for this is the presence of the syncategorematic words in GF (discussed in Section 2.4) due to the abstraction level of GF-RGL intended to maximize sharedness across languages.

For example, GF-RGL defines multiple functions in abstract syntax for existential clauses (*there is a cat*). Existential clauses are analysed using the `ExistNP` or `ExistNPAdv` function. The `ExistNP` function takes a NP and constructs a clause Cl. The dummy pronoun, an expletive *there* and the copula verb *is* are abstracted by the `ExistNP` function in the AST. Similarly, the `ExistNPAdv` takes a NP and a modifier phrase Adv (for example, *in the bag* to construct the clause. Linearizations of these functions vary across languages depending on whether expletives are necessary and the choice of copula in the language. See Figure 12 for the parse trees of the clause in English and Bulgarian. The Bulgarian translation of the existential clause neither contains the expletive or the copula, instead the verb used here is annotated as the main head in current UD annotation. We will discuss existential clauses in Section 5.6.

We extend the set of mappings defined until now with three additional types of rules:

i) **local abstract rules** are rules on abstract functions addressing their immediate arguments only (all rules defined until this point are of this type)
ii) **non-local abstract rules** allow mappings to be defined on the abstract functions using more context than the immediate arguments of the functions.
iii) **local concrete rules** are defined for each language on the respective linearizations of an abstract function.
iv) **non-local concrete rules** are defined to map the linearization of a function to the UD labels using more contexts like in the case of non-local abstract rules.

Table 7 shows the different types of rules and specification formats used to encode UD information. We previously mentioned that the mapping is encoded in a declarative manner to remain independent of

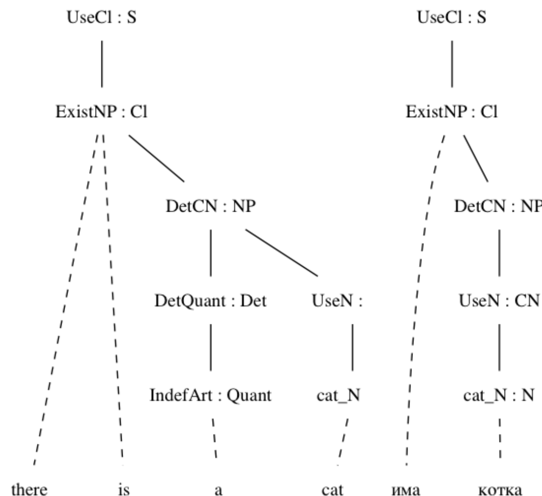

FIGURE 12 Parse trees of the existential clause *there is a cat* and its translation in Bulgarian

	Abstract	**Concrete**
Local	Fun Label+	Fun ReLabel+
Non-local	(Fun args*) Label+	(Fun args*) ReLabel+

TABLE 7 Type of rules in the extended mapping to construct full UD trees

annotation schemes. We briefly explain these new types of rules before explaining how they are used in our conversion. This is discussed in more detail at the end of this section (and Appendix).

The **local abstract rules** discussed previously are encoded using the syntax shown in Section 2. `Fun` here refers to the name of a function in the abstract syntax (`PredVP` or `ComplSlash`). One argument of `Fun` is mapped to a label called `head` and the rest are mapped to corresponding UD labels. In the trivial case, where a function takes only one argument, the argument is always mapped to the `head` label.

Non-local abstract rules are an extension of local abstract rules, that are applied only if the arguments of `Fun` in the abstract syntax tree match the values (or patterns) specified in this rule. args* represents the list of arguments of the function `Fun` (`*` corresponds to the Kleene operator in regular expressions) and each item in this list expresses a pattern for the arguments. We explain these in Section 5.1. These

patterns for arguments encode a context that is outside the scope of what is matched in local abstract rules. Non-local rules can simply be interpreted as multi-level rules in the grammar while local rules are one-level rules in the grammar. We will see these rules in more detail in Section 5.1.

Local concrete rules are introduced to address different realizations of an abstract function across multiple languages. The example of existential clauses mentioned above is an instance of this. These mappings are encoded using a list, similar to local abstract rules. In both cases of local and non-local concrete rules, each item in the list represents a relabelling operation of an edge in the dependency tree. This relabelling can be one of three types: relabel an existing edge in the dependency tree with a new label, relabel an edge with a new label after reversing the direction of the edge or add both an edge and a label to the dependency tree. These rules will be discussed in Section 5.2 for cases of copula constructions and Section 5.3 for verb phrase complements and prepositional verbs.

Non-local concrete rules are again an extension of local concrete rules, where the relabelling operations are applied only if the arguments of the abstract function Fun match the context specified in these rules. These rules are explained in the context of clausal negation and auxiliary verbs in Section 5.4.

The terms "local" and "non-local" refer to the scope over which the conversion algorithm matches the specified mappings. The mappings are always specified over functions in the abstract syntax. The terms "abstract" and "concrete" refers to the layer of syntax on which the mappings are applied. Abstract rules are applied on the AST and concrete rules are applied after abstract rules on labelled dependency trees for the specific language. Both sets of local and non-local concrete rules must be defined for each language in the RGL. In the absence of concrete rules, the conversion results in a connected UD tree, where some edges are connected artificially using the `dep` label. When the edges marked with these dummy labels are *collapsed*, the resulting dependency tree structure can be a representation that is closer to the semantics of the sentence. We will show this in our experiments described in Section 6.

5.1 Passive voice constructions

Passive clauses in GF are constructed using functions that allow a verb to drop one of its obligatory arguments to construct the VP, for example `PassV2` and `PassVPSlash`.

Let us look at the ASTs for the clause *John killed him* and its pas-

sive counterpart (*he was killed*), shown in Figure 13. The grammatical subject of the passive clause is incorrectly mapped to `nsubj` label using only local abstract rules.

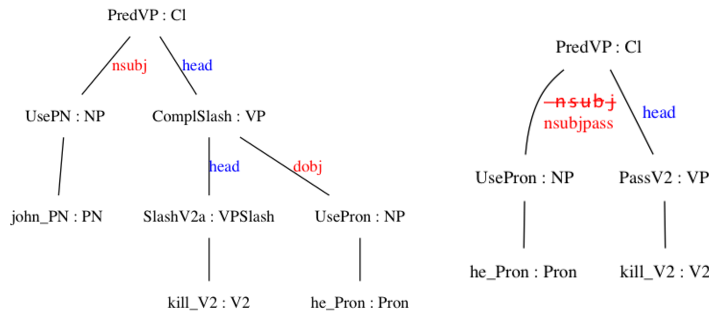

FIGURE 13 Decorated ASTs for the clause *John killed him* and its passive counterpart *He was killed* **using both local and non-local abstract rules**

The `PassV2` function in GF-RGL allows a transitive verb to form a VP phrase without the obligatory NP phrase (`dobj`). The extensions in GF-RGL define a function `PassVPSlash` that can be used to construct passive voice constructions for both transitive and ditransitive verbs. This function can be understood to be a generalization of the `PassV2` defined in the core RGL. However, the top level predication function remains the same (`PredVP`) irrespective of active or passive voice.

The UD annotation scheme distinguishes between subjects of active and passive clauses (NP arguments) using two labels — `nsubj` and `nsubjpass`. In order to make this distinction in our mapping, the rules corresponding to `PredVP` should be enriched with more context. We define the following non-local rules for `PredVP` function for this purpose.

```
(1) (PredVP ? PassV2)      nsubjpass  head
(2) (PredVP ? PassVPSlash) nsubjpass  head
(3)  PredVP                nsubj      head
```

The context in this example is encoded using abstract function names corresponding to passivization of VP phrases. The first rule `(1)` is **only** applicable in the case when `PassV2` appears in the sub-tree corresponding to the VP phrase i.e. transitive verbs in passive voice constructions. Similarly, the second rule `(2)` addresses di-transitive verbs in passive voice, when `PassVPSlash` appears in the sub-tree of the VP phrase. Finally, the general rule (`(3)`, also the local abstract

rule) is applied in all other cases, when none of these two contexts are matched i.e. in active voice constructions. The `?` character represents a meta-variable in the tree. This is interpreted as matching anything i.e. there are no restrictions on the sub-tree corresponding to this argument. In the example of the predication function, there are no restrictions on the sub-tree corresponding to the NP argument. Using these non-local rules, grammatical subjects of passive voice clauses are mapped to the `nsubjpass` label instead of the `nsubj` label.

We also define similar rules for `PredSCVP` function, used in analyses of constructions with clausal subjects. Recall from Section 4 that the `PredSCVP` function is used to construct a clause using an embedded clause and a VP phrase.

The passive voice constructions described above is addressed using a limited non-local context (the daughter nodes). However, in principle it is possible to define contexts corresponding to abstract functions using expressions of arbitrary depth.

Before extending the format of the dependency configurations, we considered the alternative of changing the RGL to resemble UD annotation in this particular case. RGL localizes the voice transformation to the VPs while UD uses a different label for subject arguments that is non-local to the VP, which could be easy to achieve in GF as well. However, if we consider co-ordinated constructions with subject sharing (*he killed his wife and was chased by the police*), the shared subject *he* should be both an `nsubj` and an `nsubjpass`. UD resolves the conflict with its flat structure to coordinations, by choosing the label appropriate for the first conjunct (`nsubj`). However, by localizing the voice to the VP, it is possible to give separate interpretations to the subject.[6]

At this point, it is clear that the addition of non-local rules makes it necessary to introduce a notion of precedence between the local and non-local rules for a specific function. We will formally address this question in Section 5.7. For now, it is enough to note that non-local rules should precede local rules, and this is specified in the semantics of dependency configurations.

5.2 Copula constructions

Copula constructions in GF-RGL are analysed in two steps: any of the `Comp*` functions are used to convert phrases into copula complements (Comp in GF-RGL) followed by `UseComp` function that converts this complement into a VP. The `UseComp` function also introduces the copula verb into the VP (when necessary). The choice of copula verb and its

[6]With the above local rules for `PredVP`, the AST for this example will yield the same dependency tree as the UD tree, which is just what we wanted.

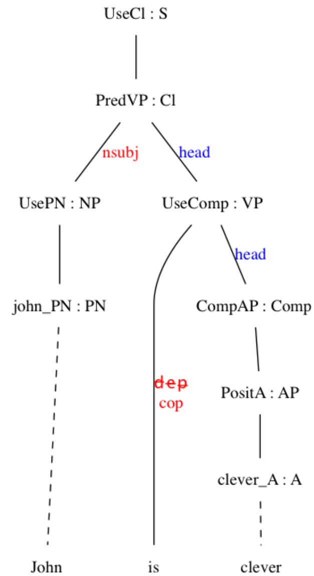

FIGURE 14 Decorated parse tree for a copula construction in English

form is specified in concrete syntax and the abstract syntax tree does not know exactly what the copula verb in the specific language is. This abstraction is desired because the realization of copula constructions varies across languages. For example, the linearization of the `UseComp` in Russian (so called *zero-copula* language) doesnot contain a copula verb in present tense. The linearization rules in English, Finnish and Swedish introduce the verbs *is*, *on* and *är* respectively. In the case of Chinese and Thai, the copula verbs are restricted to only some selected complements. Figure 14 shows the decorated parse tree in English for the sentence *John is clever*. Notice the dummy `dep` label for the copula verb *is* obtained using abstract rules.

The UD scheme in the case of copula constructions annotates the copula verb as a child of the complement using the `cop` label. In order to match this structure in our mapping, we introduce a **local concrete rule**, that marks copula verbs using the `cop` label. Shown below is the concrete rule specified for English:

```
UseComp  head {"am", "is", "was",
            "are", "were", "be", "been", "being"} cop head
```

This rule specifies that if any of the different forms of the copula

verb in English (*am, is, was, are, were, be, been*) are found under the functions in the spine corresponding to the `head` of the `UseComp` function in the AST, they should be attached to the head of the `UseComp` function using the `cop` label. The expression

```
head {"am", "is", ...} cop head
```

specifies a single relabelling operation in the concrete rule.

5.3 Verb phrase complements and prepositional verbs

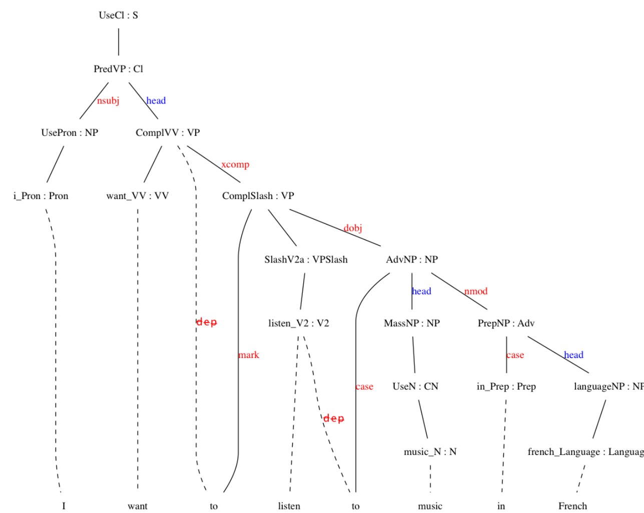

FIGURE 15 Decorated parse tree showing verb phrase complements and prepositional verbs

Let us look at one more example of these local concrete rules used in our mapping for verb phrase complements and prepositional verbs. GF-RGL defines a different function for complementation in case the argument is a verb phrase (these verbs are marked as VV in RGL). The `ComplVV` function is used with verb phrase complements. The `ComplSlash` function used for nominal objects additionally is also used for prepositional verbs (*listen to ...*). For these verbs, the required case of the NP argument is specified in the entry corresponding to the verb in the lexicon. This case information is then propagated to the `ComplSlash` function (specifically the function stores this information in a record with label c2).

Figure 15 shows the parse tree for the sentence *I like to listen to music in French.* The verb *like* is an example of verb phrase complement

and *listen to* is a prepositional verb. Note that in both these functions, the infinitive marker to the verb phrase complement and the case are abstracted from the AST, these are localized in the concrete syntax (seen only in parse tree) of the language. The local abstract rules shown in Table 1 map the head of these arguments to `xcomp` and `dobj`, but leave the infinitive marker *to* and preposition *for* unlabelled.

```
ComplSlash  head .c2      case   dobj
ComplVV     head {"to"}   mark   xcomp
```

The local rules above specify the following relabeling operations: for the `ComplSlash` function, the preposition/case in the .c2 record field should be relabelled as a child of the direct object (marked `dobj`) using the label `case`. In the `ComplVV` function, the infinitive marker *to* is labelled using the `mark` label as a child of the verb complement.

5.4 Auxiliary verbs and verbal negation

Non-local concrete rules address dependencies of words introduced in the concrete syntax where context inside the arguments of an abstract function are necessary to determine the respective UD labels. In order to understand the necessity of these rules, we will look at a concrete example (shown in Figure 16). The motivation in this example is to address cases of auxiliary verbs constructed from tense in GF and clausal negation in our mapping.

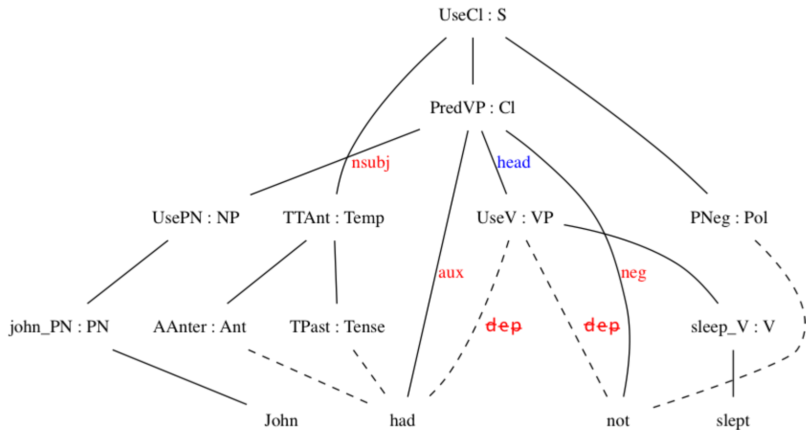

FIGURE 16 Decorated parse tree showing tense and clausal negation with the UD labels

The decorated parse tree in Figure 16 shows the analysis provided

by GF for the sentence *John had not slept.* The parse tree for the clause *John sleeps* has the same functions and categories as this sentence. Why is this? The Cl type resulting from the predication (`PredVP`) function represents an untensed clause, in other words a simple proposition. This proposition is transformed to a sentence when `Tense` and `Polarity` arguments are given to the `UseCl` function along with an untensed clause Cl. The `UseCl` function will then result in a sentence of type S. Note the `PNeg` value for the polarity of the clause and the `TPast,AAnter` for the tense in our example. The *not* appears due to the `PNeg` value and the auxiliary verb *had* due to the anteriority of the tense. The tense parameter dictates the choice of auxiliary verbs and the polarity parameter if *not* occurs in the clause. The equivalent UD labels are also shown in the parse tree. It can be seen that the modifiers according to UD are not always in the same sub-tree as the head (*not* modifies *slept*).

In order to make the mapping to UD labels, we need non-local concrete rules, as the specific auxiliary verbs in a language are abstracted by the Tense parameters in the AST and are only available in concrete rules for the language. The non-locality is also required to handle cases of negation.

```
(1) (UseCl ?     PNeg ?)  head {"not"} neg head
(2) (UseCl tense   ?  ?)  head {*}   aux head
```

The non-local rules for auxiliaries and copulas are defined as shown above. The negation modifier (*not*) is introduced into the clause by the `UseCl` function if its polarity argument has the `PNeg` value. The word *not* is introduced under the `UseV` function, which lies on the spine corresponding to the head of the `UseCl` function. The same is also true for auxiliary verbs, they are introduced by the Tense parameter under the `UseV` function. For clausal negation, the edge between the head word of the `UseCl` function and the word *not* is labelled using the `neg` label, with *not* as the child. Similarly, auxiliary verbs introduced by the `Tense` argument to the `UseCl` function, irrespective of what exactly is the auxiliary are always connected to the head using the `aux` label. The `{*}` in the rule specifies that all and any auxiliary verbs can be matched in this relabelling operation.

5.5 Multi-word expressions

One of the UD taxons we haven't discussed yet are cases of multi-word expressions and some instances of compounding. Compounding encompasses a wide range of linguistic phenomena, from compound nouns and phrasal verbs to non-compositional multi-word expressions. In the UD scheme, this also includes foreign language phrases and two parts of a

single word in poorly written text. The annotation of compound nouns in UD (for example *phone book*, *agent provocateur*) is fairly consistent across languages. However, there is variation in annotations of phrasal verbs. This is not surprising given the variety of "phrasal verb" constructions in different languages.

In the GF-RGL, compounds are handled in two different ways: compositional compounds are analysed using the functions in RGL mentioned previously `CompoundN` and `CompoundAP` (for example *control systems*, *language independent*). Other compounds i.e. non-compositional compounds are addressed inside the lexicon specific to a language. The analyses provided to these types of compounds is very much dependent on the concrete syntax of the language, whether the translation in the specific language is compositional or not. It is also possible that the translation is simply lexicalized, without any compounding in a languages. For these reasons, the abstract syntax shared for these compounds is only the function name, specific analysis of the compound is localized in the concrete syntax. This is because languages allow for discontinuous compounds, particularly in cases of phrasal verbs and light verb constructions (for example *shut off* in *shut it off*, *shut off the TV*).

We provide three dependency labels for all these compounding phenomena in our mapping, it can either be a `compound`, `mwe` or `goeswith`. However, unlike the local abstract rules for the functions that correspond to compositional compounds, these labels are defined on the concrete syntax of a specific language. Shown below are the local concrete rules used for particle verbs *shut off* and multi-word expressions *according to* and *in front of*.

```
shut_off_V2        head {"off"} compound head
according_to_Prep head {"to"} mwe head
in_front_of_Prep  head {"front", "of"} mwe head
```

The relabelling operation in the case of `shut_off_V2` specifies that the particle *off* be added as a child to *shut* with a `compound` label. Similarly, for `in_front_of_Prep`, edges from *in* to *front* and *of* are relabelled using the `mwe` operation. In all these cases, it is implicitly understood that the first word (*shut*, *according*, *in*) is the head of the function. This is in sync with the current UD annotation scheme in its current form. However, the rule specification format is flexible to allow a different choice of the head, say for example, a semantic head for multi-word expressions. The UD annotation scheme where multi-word expressions are concerned might be revised in the future, and our current mapping scheme allows room for any such changes.

5.6 Idiomatic and Semantic (CxG) Constructions

Finally, we discuss semantic constructions that are commonly found in languages and can be translated idiomatically in languages. We discussed at the beginning of this section how GF-RGL defines functions for constructions like existential clauses (*there is a blue house on the hill*) or cleft sentences (*it is money that was owed to him*). There is also a small subset of *semantic* constructions defined in GF-RGL. One example of these semantic constructions is a function that accepts a number, say 10 and renders the semantic linearization of the VP *is 10 years old* in a specific language.

```
has_age_VP n   -- (someone) "is" n "years old"
```

We define the mappings to UD labels over these semantic constructions defined in the RGL. In order to construct the fully connected UD trees, it is necessary to define both abstract rules and concrete rules over these functions. The rules used for the `ExistNP` function are shown below. The expletive pronoun *there* is marked using the `expl` and the copula verb indicating tense has the label `cop`.

```
ExistNP head {"there"}  expl head ; head {*} cop head
```

Notice that in these examples, the concrete mappings specify more than one relabelling operation. This is possible and the `;` is used as a separator between different operations. The conversion algorithm carries out each of these operations in the order specified by the mapping.

5.7 Dependency conversion algorithm and specification language

We show the exact **dependency configuration syntax** used to specify the mappings to UD scheme and more generally to any dependency annotation scheme in Figure 19 (in Appendix). The figure shows the BNF fragment specifying the syntax of the configuration rules.

In the case of local rules, both abstract and concrete, the key to the mapping is the abstract syntax function name `Fun` that specifies where the mapping should be applied. Similarly for non-local rules, this key is extended from `Fun` to a pattern of expression that matches sub-trees. Each `arg` in the non-local rule is a pattern for the arguments of `Fun`, the number of such patterns being the same as the number of arguments. The `?` character is used to indicate match everything.

For abstract rules, the mapping is a list of labels of the same size as the number of arguments to function. The items in these lists are any of the labels specified according to the annotation scheme or the special label `head` that encodes what is the head corresponding to the

abstract function. By definition, there should be one and only one `head` in each mapping specified.

The mapping in the case of concrete rules is a list of relabelling operations. The number of these operations specified by a single rule depends on the number of lexical items introduced by the concrete syntax in the language that are hidden in the AST. There are three possible relabelling operations defined: renaming an existing edge with a new label (retains the already encoded head-modifier relationship), renaming an existing edge after modifying the direction of the edge with a new label or adding a completely new edge that did not exist before. Examples of all three operations have been shown before. Each relabelling operation is specified as

Label Part Label Label

The first `Label` in a relabelling operation is a label specified in the abstract rules, to indicate functions(s) in the AST marked with `Label`. This specifies a spine in the AST used to locate functions under which syncategorematic words can be found. `Part` specifies either a set of words (in the case of copula verbs) or a record label (in the case of prepositional verbs) or a `*` to match all lexical items introduced under a function in the AST. The remaining two `Label` specify the label of the modifier and the label of the head in the AST in that order.

The precedence of the local and non-local rules corresponding to a function `Fun` is defined in the same order as they are specified in the configuration syntax. In other words, non-local rules that must be applied before are specified before the local rules. By choosing to put this in the configuration, we do not modify the conversion algorithm to specially handle non-local rules.

With this dependency configuration syntax, the dependency conversion algorithm is extended to derive the complete UD tree. In Section 2, the conversion method given an AST T and a word sequence S to its corresponding dependency tree was described. The algorithm is divided into two separate steps: conversion using the abstract rules and the concrete rules in that order. Algorithm 1 (in Appendix) describes the extended conversion algorithm we used in this paper.

6 Experiments

In our experiments, we focus on evaluating two things: the mapping proposed in this paper to convert GF trees into UD dependency trees and subsequently how much the abstract syntax in GF-RGL and the mapping is useful in bootstrapping treebanks. We carry out experiments to evaluate the correctness of our mapping and to understand

the systematic similarities between GF-RGL and the UD annotation scheme.

In the first set of experiments, using mappings defined on the abstract syntax alone (both local and non-local rules), we bootstrap dependency trees for 31 languages from a treebank of ASTs. Analyses of these bootstrapped treebanks give insights about the UD scheme and the GF-RGL. The lack of concrete local and non-local rules for a specific language result in trees that resemble *collapsed* dependency trees (de Marneffe and Manning, 2008, Ruppert et al., 2015) by deleting the dummy `dep` labels. Due to the lack of concrete rules so far, the bootstrapped treebanks for languages other than English do not contain labels for syncategorematic words, and these `dep` labels are collapsed. Figure 17 shows examples of bootstrapped and "collapsed" UD trees for Swedish and Bulgarian in the context of this work. When defined, the concrete rules "decollapse" these trees by introducing the right UD labels for syncategorematic words.

6.1 UD test treebank

In order to evaluate the mapping described in this paper, we created two treebanks of ASTs. The first treebank was constructed using examples in UD annotation guidelines for English dependency relations [7]. The ASTs from the GF parser (Angelov and Ljunglöf, 2014) were post-edited to correct the ambiguity choices in these examples. The collection of examples from the UD guidelines ensure good coverage of the dependency relations proposed in UD scheme. Similarly, in an attempt to maximize the coverage of functions defined in the GF-RGL, we created a second treebank composed of minimal examples used to document the linguistic usage of abstract functions in writing resource grammars for new languages[8]. The second treebank is relevant for two purposes: the ASTs from the GF-RGL documentation ensure full coverage over the GF interlingua, and also provide minimal examples/testcases for purposes of UD documentation. These can be used as unit tests to validate the UD annotation efforts. Unit tests are typically used in software engineering to validate the software i.e. if the actual output of the system matches the expected result. Additionally, these examples can be compositionally combined to automatically create a treebank of ASTs which can then be bootstrapped to UD treebanks for languages in GF. We refer to the former treebank as UD treebank and the latter as GF treebanks here, these should not be confused with the treebanks provided in the UD and GF distribution. Using both

[7] http://universaldependencies.github.io/docs/en/dep/all.html
[8] http://www.grammaticalframework.org/lib/doc/synopsis.html

these treebanks we evaluate the precision of the mapping and make qualitative analyses.

The UD treebank contains a total of 104 ASTs, 66 ASTs that are covered by RGL and 38 ASTs that use the robust back-up rules in the grammar enabled for wide coverage parsing and translation. As these examples are hand-picked to document the annotation guidelines for UD treebanks, the treebank does not have a specific genre. We will report the precision of the mapping rules separately for these two sets. One reason to set apart these two sets is the ad-hoc nature of recovering UD labels in case of rules corresponding to chunking that are a small part of the robust back-up rules. Another reason is that these rules are implemented only for 15 of the 31 languages covered by the core RGL. So, the subset of ASTs covered by the RGL are bootstrapped into all 31 languages while the rest of them are bootstrapped into only 15 languages. Many of these examples have been used previously in this paper as examples in Section 3, Section 4 and Section 5.

The GF treebank contains a total of 400 ASTs, that are covered by the core RGL. These examples are used to document the usage of abstract functions while writing concrete rules for a new language. Hence, these examples provide a complete coverage over the RGL-grammar and are minimal examples of different linguistic phenomena that can be tested for cross-lingual consistency in UD annotations. We bootstrap this treebank into all 31 languages in the RGL (these examples have been previously used in this paper in Tables 1 and other tables).

6.2 Evaluation

	Local	Non-local
Abstract	132	11
Concrete	17	29

TABLE 8 Distribution of rules for English according to their types

The complete mapping described in this paper, contains a total of 185 rules, 143 of which are defined on the abstract syntax. The remaining 46 rules are defined as either concrete local and non-local rules for English. Table 8 shows the distribution of the rules in the complete mapping for English. It is necessary to define concrete rules for each language to construct fully labelled UD trees.

Figure 17 shows an example of bootstrapped trees for the Swedish, Finnish and Bulgarian translations of the English sentence *John was killed* generated using only the rules defined on the abstract syntax.

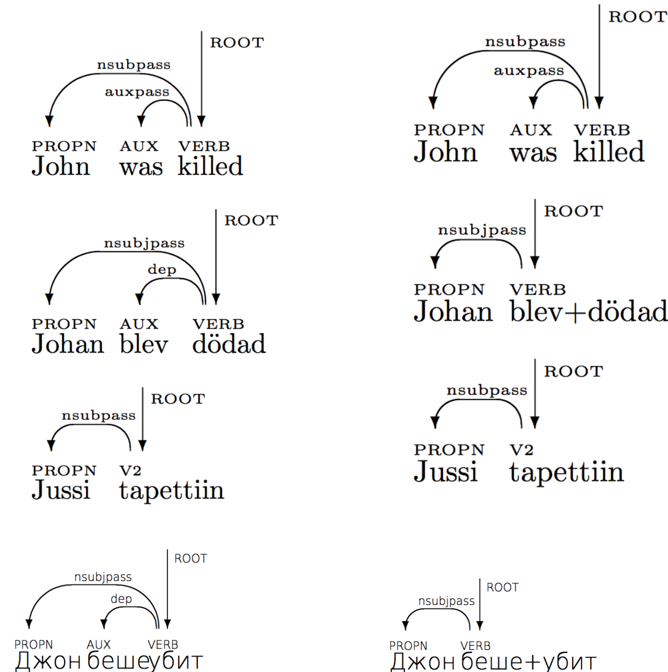

FIGURE 17 Bootstrapped UD trees in Swedish, Finnish and Bulgarian and the complete UD tree in English. The trees on the right are the collapsed variants of the bootstrapped trees.

Also shown is the full UD tree for the English sentence generated using both the abstract and concrete mappings.

Analyses of English dependency trees

The converted UD trees in the UD treebank have a 99% labelled attachment score (LAS) to the gold UD trees. The dissimilarities in English occur only in two cases: modal verbs and noun phrases, specifically noun phrases with `nummod` labels. A 100% LAS score is achieved by adding additional rules to handle modal verbs in our mappings. While our mappings with these additional rules result in perfect matching with the UD trees, we look at these two specific cases in some detail below.

The UD annotation guidelines treat modal verbs in English the same way as auxiliary verbs, with a POS tag AUX and `aux` label. The lexical

features of the word indicate its modality feature, but the label assigned is the same as auxiliary verbs. This is expected since modal verbs in English are realized as auxiliaries in English. However, modal verbs in other languages can be realized in inflectional variants of the main verb, for instance, English *would* as conditional forms in French.

Non-auxiliary modal verbs in GF receive a structure that is shared across languages: they are verbs of the category VV, i.e. verbs that take VP complements, similar to verbs like *want* or *like* in *I want to ask you something* and *I like to run.* This analysis of modal verbs from GF-RGL maps the modal verb to the head and the main verb to the `xcomp` label as a child of the modal verb using the mapping defined for the `ComplVV` function (complementation using non-finite arguments). The labels to the arguments of the main verb remain the same though.

It is possible to give modals in English the `aux` label by adding non-local abstract rules for the `ComplVV` function to match only modals:

```
(ComplVV can_VV ?)   aux head
(ComplVV must_VV ?)  aux head
```

However, by defining these mappings on the abstract functions, we would map modal verbs in all the thirty languages to the `aux` label, even in languages where modal verbs are not realized as auxiliary verbs. The UD treebanks in some languages treat the modal verb as the head of the clause, while other languages treat them consistently as auxiliaries (English and Swedish in particular). The problem is that modal verbs are not always realized as auxiliaries across languages. For instance, *can* and *must* in English translate to *pouvoir* (or *savoir*) and *devoir* in French, which are semantically modal but work otherwise just like normal verbs. In the other direction, *want* is not modal in English, but its equivalent *wollen* in German is. The RGL choice is to have a uniform abstract category VV of VP-complement verbs and to treat the property of being "modal" in the concrete syntax of each language[9].

Alternatively, the mapping of modal verbs can be addressed using concrete rules specific to a language. This allows our conversion method to address the differences in the way modal verbs are treated in different languages in UD. These rules are shown below. These concrete rules relabel the edges of the converted tree in the case of modal verbs to obtain the exact UD tree. What these rules say is that the head of the `ComplVV` function i.e modal verbs in this case (*can* or *must*) are relabelled using the `aux` label while the other argument i.e. the verb is

[9]More concrete syntax distinctions are needed in languages like Finnish, where VV can take its complement in at least five different infinitival forms, and Hindi, where modal verbs interfere with verbalizers.

relabelled to be the head of the `ComplVV` function. Defining the mapping for modal verbs using concrete rules for English is the best way to address the different annotations used for modal verbs.

```
(ComplVV can_VV ?)  head {*} aux head; xcomp {*} head head
(ComplVV must_VV ?) head {*} aux head; xcomp {*} head head
```

The other divergence from UD is in the case of noun phrases with numerical modifiers: UD annotation scheme treats all numerical modifiers occurring in NPs as modifiers of the head noun using the `nummod` label. While this is valid in almost all cases, it is possible for the number to require another label than a modifier. For example the phrase *level 3* is treated as a compound in GF, one where both *level* and *3* contribute to the meaning. We retain this analysis by mapping this to the `compound` label, rather than using the `nummod` label. Also, in instances of how noun phrases with numerical heads, as in the case of *these five will come with me*, there is a difference in how the NPs are analysed. As explained previously, we treat the quantifier as the head in the `DetQuant` function, as such these modifiers are attached to the quantifier using the `nummod` label. These can be addressed using non-local abstract rules for the `DetCN` function where the Det argument is an ordinal number.

Analyses of bootstrapped dependency trees

Table 9 shows the percentage of labelled edges in the bootstrapped treebanks. We report these numbers by calculating the fraction of edges in the treebank for the language labelled using the `dep` relation. The numbers in turn suggest the reduction in UD annotation efforts by bootstrapping using the GF RGL and wide-coverage abstract syntax. Note here that for these bootstrapping experiments, we only use the rules on abstract syntax (for all languages, English especially). One interesting result is that partial UD treebanks can be obtained even for languages with incomplete grammars. If the linearization rules for all the functions defined by the core RGL are missing in the concrete syntax for a specific language, we say that the RGL grammar for the specific language is incomplete. In Table 9, Amharic is one example of a language with an incomplete grammar implementation.

In the case of the UD treebank, it can be seen that on average across all languages, about 80% of the edges are labelled with the UD labels from the RGL bootstrapped treebanks. There is a decrease in this when we go to the wide coverage treebank. This is because of the semantic constructions introduced in the wide coverage grammars, necessary to achieve the abstractions required for interlingua-based MT. These constructions require concrete mappings in order to construct

Language	UD treebank		GF treebank
	RGL	Wide-coverage	
Afrikaans	81.57	-	81.73
Amharic	73.60	-	75.29
Bulgarian	76.60	80.53	88.13
Catalan	82.18	76.13	83.10
Chinese	84.89	77.33	82.46
Danish	80.49	-	87.45
Dutch	83.62	68.10	84.23
English	84.12	81.17	93.13
Estonian	82.38	79.10	86.52
Finnish	81.84	74.09	91.27
French	82.81	79.61	93.47
German	83.09	77.47	95.27
Greek	83.09	-	88.13
Hindi	72.63	69.34	84.18
Italian	81.79	77.52	90.47
Japanese	71.39	71.09	84.94
Latvian	72.11	-	89.10
Maltese	83.92	-	90.29
Mongolian	72.22	-	87.34
Nepali	82.91	-	83.19
Norwegian	80.37	-	86.36
Persian	83.87	-	84.40
Punjabi	72.37	-	82.82
Polish	83.05	-	87.38
Romanian	69.58	-	86.75
Russian	87.30	-	87.92
Sindhi	68.21	-	84.59
Spanish	81.41	79.15	91.28
Swedish	82.56	83.05	93.89
Thai	83.72	73.12	85.29
Urdu	72.86	-	84.67

TABLE 9 Percentage of completeness in the bootstrapped dependency treebanks. The Amharic grammar (underlined) is incomplete, i.e. does not implement all RGL functions.

the dependency trees. In comparison, the GF treebank results in much better scores. The percentage of labelled edges is higher in the GF treebank because syncategorematic words have a much lesser incidence in the unit tests corresponding to functions defined in the GF-RGL.

From the collapsed trees in the bootstrapped treebanks, we primarily learned that the missing labels correspond to auxiliary verbs and copula verbs in most instances. The fact that these are frequent in all languages is clearly the reason for this. But this also suggests that defining a very small fragment of concrete non-local rules will give us huge improvements in connecting the bootstrapped treebanks with the correct UD labels.

A full investigation of the quality of these bootstrapped treebanks is pending due to the lack of manually annotated UD treebanks for this collection of languages.

6.3 GF Penn Treebank

Previous work on parsing in GF has resulted in a version of the Penn treebank mapped to the GF-RGL functions (Angelov and Ljunglöf, 2014). The GF-Penn treebank was created by mapping the annotation schema used in Penn treebanks with the abstract functions in the RGL using hand-crafted rules. This treebank is used to train statistical models for disambiguation in the parser and the wide-coverage translator. In cases where a fragment of the sentence is not covered by the grammar, the resulting AST is a tree, except that the missing functions in the grammar are replaced by missing nodes. Hence, the GF-Penn treebank is a GF annotation of the trees in the original Penn treebank where constructions outside the scope of the grammar are marked using a default function indicating incompleteness of the grammar. Nonetheless, the partially complete trees in the GF-Penn treebank are useful in estimating the probability distributions over the grammar.

We performed experiments with converting the GF treebank into the UD annotation scheme using our mapping. The entire treebank has about 5% of missing functions, on average each such function has two arguments. So, 10% of the edges on average are simply assigned the `dep` label because the algorithm can not determine the mapping corresponding to this function. We did not make any changes to the mappings or the conversion procedure to address the case of missing nodes.

The converted treebank is evaluated against a UD version of the Penn treebank, these UD annotations are obtained from the Stanford parser distribution (de Marneffe and Manning, 2008, de Marneffe et al., 2014). We evaluate our converted treebank against this UD treebank by

Section	LAS
WSJ-22	81.34
WSJ-23	83.21
WSJ-24	85.12

TABLE 10 LAS scores to compare our mappings to UDs against Penn UD treebank

computing labelled attachment scores used in evaluating dependency parser performance. This metric calculates the percentages of edges in the treebank that are attached to their correct head and assigned the correct dependency label. Table 10 shows the LAS scores on Sections 22, 23 and 24 of the GF-Penn treebank, against standard partitions of the Penn treebank that are used to report parser accuracies.

Analysis of the converted UD treebank showed three major reasons for failure to map the entire treebank to the UD labels.

i) About 5% of the edges in these sections are wrongly labelled due to divergence in our mapping from UD annotation scheme. Most of these correspond to the treatment of modal verbs (mentioned previously) and a small fraction of these edges correspond to numerical modifiers mapped as children of the quantifiers in our mapping.

ii) About 7% of the wrongly labelled edges correspond to proper nouns in the corpus. The UD annotation scheme selects the first token in a name to be the head and all the rest of the tokens in the name are labelled using the `name` label. Contrary to this, GF treats the entire name as a single token and does not assign edges between tokens in a proper noun.

iii) About 7% of the edges are wrong labelled due to the missing functions in the GF treebank. We mentioned previously we did not make any changes to the conversion algorithm or the mappings to address these cases.

7 Conclusion

In this paper, we investigate the similarities and differences between GF-RGL and Universal Dependencies (UDs). Our main result is a conversion from GF-RGL abstract syntax trees to UD dependency trees.

The conversion from abstract syntax trees in GF to UD trees is parameterized by a mapping defined on functions in the abstract syntax, to encode the notion of "head" and the grammatical relation of other modifiers with respect to the head. The mapping described in this work has has two levels – rules defined on the abstract syntax

and rules defined on the concrete syntax. Rules defined on the abstract syntax are by definition language-independent and "shared". However, rules defined on concrete syntax are defined for each language separately, even if the UD labels for these may be shared across languages. Furthermore, both abstract and concrete rules have two types – "local" and "non-local". In the case of "local" rules, the mapping between function and its arguments and their UD annotations (both the identity of head and the UD labels) is determined using only the function, unlike "non-local" rules where a mapping applies only in certain contexts.

The mapping from GF-RGL to UDs showed that a large part of the GF-RGL and UD annotation scheme is language-independent. About 30 of the 40 UD labels are mapped using the abstract "local" and "non-local" rules in our experiments. These parts of both GF-RGL and UD scheme can be said to be language-independent. The parts of the mapping defined on the concrete syntax interestingly falls into two taxons within the UD annotation scheme (clausal dependents such as `aux`, `auxpass`, `cop`, `mark`, `expl` and compounding labels `compound`, `mwe`).

One application of the conversion detailed in this paper is to automatically create UD treebanks from GF treebanks. Our experiments using a small treebank showed that about 80% of UD treebank annotations can be ported using the abstract syntax in GF-RGL. The remaining annotations can also be derived from GF, provided a small effort is put into defining concrete rules in the mapping for languages of interest. Other plausible applications and future directions of our work include multilingual consistency checking in the UD annotated treebanks and using these treebanks to estimate probabilistic ranking of GF abstract syntax trees.

Acknowledgements

We would like to thank Krasimir Angelov, Filip Ginter, Richard Johansson, Joakim Nivre and the two anonymous referees for their valuable comments and suggestions. Special thanks to Annie Zaenen for her technical comments in addition to all editorial help. The research was funded by the REMU project (Reliable Multilingual Digital Communication, Swedish Research Council 2012-5746).

References

Aho, Alfred V. and Jeffrey D. Ullman. 1969. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences* 3(1):37–56.

Angelov, Krasimir, Björn Bringert, and Aarne Ranta. 2014. Speech-enabled hybrid multilingual translation for mobile devices. In *Proceedings of the*

Demonstrations at the 14th Conference of the European Chapter of the Association for Computational Linguistics, pages 41–44. Gothenburg, Sweden: Association for Computational Linguistics.

Angelov, Krasimir and Peter Ljunglöf. 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.

Appel, Andrew. 1998. *Modern Compiler Implementation in ML*. Cambridge University Press.

Bender, Emily M. and Dan Flickinger. 2005. Rapid Prototyping of Scalable Grammars: Towards Modularity in Extensions to a Language-Independent Core. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing IJCNLP-05 (Posters/Demos)*. Jeju Island, Korea.

Böhmová, Alena, Jan Hajič, Eva Hajičová, and Barbora Hladká. 2003. The Prague dependency treebank. In *Treebanks*, pages 103–127. Springer.

Butt, Miriam, Helge Dyvik, Tracy Holloway King, Hiroshi Masuichi, and Christian Rohrer. 2002. The Parallel Grammar Project. In *COLING 2002, Workshop on Grammar Engineering and Evaluation*, pages 1–7.

Collins, Michael. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 184–191. Association for Computational Linguistics.

Curry, Haskell B. 1961. Some Logical Aspects of Grammatical Structure. In *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Dannélls, Dana, Mariana Damova, Ramona Enache, and Milen Chechev. 2012. Multilingual Online Generation from Semantic Web Ontologies. In *Proceedings of the 21st International Conference on World Wide Web*, pages 239–242. Lyon, France: ACM.

de Marneffe, Marie-Catherine, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D. Manning. 2014. Universal Stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 4585–4592. Reykjavik, Iceland: European Language Resources Association (ELRA).

de Marneffe, Marie-Catherine and Christopher D. Manning. 2008. The Stanford Typed Dependencies Representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8. Manchester, UK: Coling 2008 Organizing Committee.

Dowty, David R. 1979. *Word Meaning and Montague Grammar*. Dordrecht: D. Reidel.

Dymetman, Marc, Veronika Lux, and Aarne Ranta. 2000. XML and Multilingual Document Authoring: Convergent Trends. In *Proceedings of the 18th International Conference on Computational Linguistics, COLING 2000*, pages 243–249. Saarbrücken, Germany.

Gazdar, Gerald, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. 1985. *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

Hallgren, Thomas and Aarne Ranta. 2000. An Extensible Proof Text Editor. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Proceedings*, vol. 1955 of *LNCS/LNAI*, pages 70–84. Springer.

Kaljurand, Kaarel and Tobias Kuhn. 2013. A Multilingual Semantic Wiki Based on Attempto Controlled English and Grammatical Framework. In *Proceedings of The Semantic Web: Semantics and Big Data: 10th International Conference, ESWC 2013*, pages 427–441. Springer.

Khegai, Janna. 2006. GF Parallel Resource Grammars and Russian. In *Proceedings of the COLING/ACL 2006 Main Conference*, pages 475–482. Sydney, Australia: Association for Computational Linguistics.

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

McCarthy, John. 1962. Towards a mathematical science of computation. In *Proceedings of the Information Processing Congress (IFIP) 62*, pages 21–28. Munich, West Germany: North-Holland.

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

Nivre, Joakim. 2015. Towards a Universal Grammar for Natural Language Processing. In *CICLing 2015: Proceedings of Computational Linguistics and Intelligent Text Processing*, vol. 9041 of *LNCS*, pages 3–16. Cairo, Egypt.

Nivre, Joakim, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. Paris, France: European Language Resources Association (ELRA).

Petrov, Slav, Dipanjan Das, and Ryan McDonald. 2012. A Universal Part-of-Speech Tagset. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 2089–2096. Istanbul, Turkey: European Language Resources Association (ELRA).

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

Ranta, Aarne. 2009a. Grammars as Software Libraries. In *From Semantics to Computer Science. Essays in Honour of Gilles Kahn*, pages 281–308. Cambridge University Press.

Ranta, Aarne. 2009b. The GF Resource Grammar Library. *Linguistic Issues in Language Technology* 2(2).

Ranta, Aarne. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications.

Ranta, Aarne, Ramona Enache, and Grégoire Détrez. 2012. Controlled Language for Everyday Use: The MOLTO Phrasebook. In *Controlled Natural Language: Second International Workshop, CNL 2010, Revised Papers*, vol. 7175 of *LNCS/LNAI*, pages 115–136. Springer.

Rayner, Manny, David Carter, Pierrette Bouillon, Vassilis Digalakis, and Mats Wirén. 2000. *The Spoken Language Translator*. Cambridge: Cambridge University Press, 1st edn.

Ruppert, Eugen, Jonas Klesy, Martin Riedl, and Chris Biemann. 2015. Rule-based Dependency Parse Collapsing and Propagation for German and English. In *Proceedings of International Conference of the German Society for Computational Linguistics and Language Technology*. Essen.

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

1 Appendix: GF-RGL and UD Reference

This Appendix gives a listing of GF-RGL categories and UD tags and labels.

1.1 GF-RGL categories

Figure 18 shows the hierarchy of categories in the core RGL. The same picture appears in (Ranta, 2011), whereas Ranta (2009b) provides a systematic linguistic discussion of the RGL categories and functions. Table 11 lists the same categories with explanations, minimal linguistic examples and corresponding UD parts of speech.

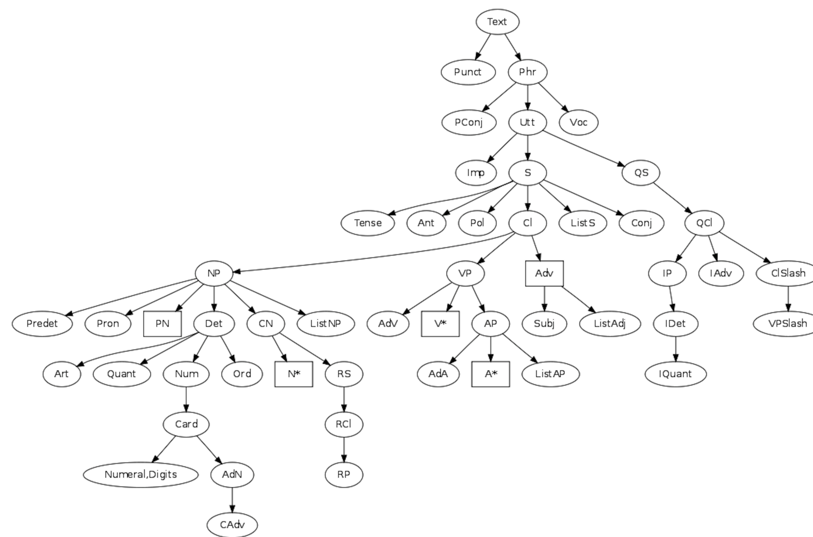

FIGURE 18 The categories of core RGL.

1.2 UD tags and labels

Table 12 shows the hierarchy of core UD labels used to annotate treebanks in the UD annotation project. Table 13 shows the universal part-of-speech tags and the morphological features annotated in the UD treebanks. We recommend the UD Documentation [10] for readers interested in further details about the UD annotation efforts.

[10] http://universaldependencies.org/docs/

Category	Explanation	Example	UD POS
A	adjective	*old*	ADJ
AP	adjectival phrase	*very warm*	phrasal
Adv	adverb or adverbial phrase	*in the house*	ADV
Ant	anteriority	simultaneous, anterior	syncat
CN	common noun (without determiner)	*red house*	phrasal
Card	cardinal number	*seven*	NUM
Cl	declarative clause, with all tenses	*she looks at this*	phrasal
Comp	complement of copula, such as AP	*very warm*	phrasal
Conj	conjunction	*and*	CONJ
Det	determiner phrase	*those seven*	DET,phrasal
IAdv	interrogative adverb	*why*	ADV
IComp	interrogative complement of copula	*where*	phrasal
IDet	interrogative determiner	*how many*	DET,phrasal
IP	interrogative pronoun	*who*	PRON
Imp	imperative	*look at this*	phrasal
Interj	interjection	*alas*	INTJ
N	common noun	*house*	NOUN
NP	noun phrase (subject or object)	*the red house*	phrasal
Num	number determining element	*seven*	phrasal
Numeral	cardinal or ordinal in words	*five/fifth*	NUM
Ord	ordinal number (used in Det)	*seventh*	NUM
PConj	phrase-beginning conjunction	*therefore*	CONJ
PN	proper name	*Paris*	PROPN
Phr	phrase in a text	*but be quiet please*	phrasal
Pol	polarity	positive, negative	syncat
Predet	predeterminer (prefixed Quant)	*all*	DET
Prep	pre/postposition, or just case	*in*	ADP
Pron	personal pronoun	*she*	PRON
Punct	punctuation mark	*!*	PUNCT
QCl	question clause, with all tenses	*why does she walk*	phrasal
QS	question	*where did she live*	phrasal
Quant	quantifier ('nucleus' of Det)	*this/these*	DET
RCl	relative clause, with all tenses	*in which she lives*	phrasal
RP	relative pronoun	*in which*	PRON
RS	relative clause, tense fixed	*in which she lived*	phrasal
S	declarative sentence	*she lived here*	phrasal
SC	embedded sentence or question	*that it rains*	phrasal
Subj	subjunction	*if*	SCONJ
Temp	temporal and aspectual features	past anterior	syncat
Tense	tense	present, past, future	syncat
Text	text consisting of several phrases	*He is here. Why?*	phrasal
Utt	sentence, question, word...	*be quiet*	phrasal
V	one-place verb	*sleep*	VERB
V2	two-place verb	*love*	VERB
V2V	verb with NP and V complement	*cause*	VERB
V3	three-place verb	*show*	VERB
VA	adjective-complement verb	*look*	VERB
VP	verb phrase	*is very warm*	phrasal
VPSlash	verb phrase missing complement	*give to John*	phrasal
VQ	question-complement verb	*wonder*	VERB
VS	sentence-complement verb	*claim*	VERB
VV	verb-phrase-complement verb	*want*	VERB
Voc	vocative	*my darling*	phrasal

TABLE 11 Main RGL categories and corresponding UD POS tags. The tag "phrasal" means that the category has only complex phrases. "syncat" means that the category is linearized to abstract features, to which words are assigned syncategorematically.

Core dependents of clausal predicates		
Nominal dep	*Predicate dep*	
nsubj	csubj	
nsubjpass	csubjpass	
dobj	ccomp	xcomp
iobj		
Non-core dependents of clausal predicates		
Nominal dep	*Predicate dep*	*Modifier word*
nmod	advcl	advmod
		neg
Special clausal dependents		
Nominal dep	*Auxiliary*	*Other*
vocative	aux	mark
discourse	auxpass	punct
expl	cop	
Noun dependents		
Nominal dep	*Predicate dep*	*Modifier word*
nummod	acl	amod
appos		det
nmod		neg
Case-marking, prepositions, possessive		
case		
Coordination		
conj	cc	punct
Compounding and unanalysed		
compound	mwe	goeswith
name	foreign	
Loose joining relations		
list	parataxis	remnant
dislocated		reparandum
Other		
Sentence head	*Unspecified dependency*	
root	dep	

TABLE 12 Dependency labels used in UD, organized as a taxonomy, as shown in Nivre et al. (2016)

Open class words		Closed class words		Other	
ADJ	adjective	ADP	preposition/postposition	PUNCT	punctuation
ADV	adverb	AUX	auxiliary	SYM	symbol
INTJ	interjection	CONJ	coordinating conjunction	X	unspecified POS
NOUN	noun	DET	determiner		
PROPN	proper noun	NUM	numeral		
VERB	verb	PART	particle		
		PRON	pronoun		
		SCONJ	subordinating conjunction		

Lexical	Inflectional	
	(Nominal)	(Verbal)
PronType	Gender	VerbForm
NumType	Animacy	Mood
Poss	Number	Tense
Reflex	Case	Aspect
	Definite	Voice
	Degree	Person
		Negative

TABLE 13 Top table shows the Part-of-speech tags in UD. Table below shows morphological features in UD. Both tables shown verbatim as-in Nivre et al. (2016)

2 Appendix: Dependency conversion algorithm and specification language

Figure 19 shows the BNF grammar of dependency configurations.

```
Rule     ::= Fun  Label+
           |  Fun  Relabels
           |  Tree Label+
           |  Tree Relabels
Relabels ::= Relabel ; Relabels
           |  Relabel
Relabel  ::= Label Part Label Label
Part     ::= "." Field
           |  "(" Words ")"
           |  "{*}"
Tree     ::= "(" Fun Arg* ")"
Arg      ::= "(" Tree ")"
           |  "?"
Fun      ::= Ident
Label    ::= Ident
Words    ::= QuotedString
           |  QuotedString "," Words
```

FIGURE 19 BNF specification of the dependency configuration syntax. Label+ refers to one or more labels using the syntax of regular expressions.

Shown below in Algorithm 1 is the conversion algorithm used to obtain dependency trees from abstract syntax trees.

```
Data:  Abstract syntax tree T, configuration on abstract syntax C_abstract
       and concrete syntax C_concrete
Result: Dependency tree D
# Decorate T with labels to get T_L and list of concrete configurations
T_L, concreterelabels ← decorate(T)
# Convert T_L to dependency tree D
Dpartial ← getdependencies(T_L)
# Apply concrete mappings to get complete dependency tree
UD ← relabeledges(Dpartial, concreterelabels)

Function decorate(tree)
    decoratedtree ← tree
    concreterelabels ← Table() # initialize an empty table
    foreach node in AST tree do
        funcontext ← get subtree dominated by node
        funname ← get function name at node
        abstractconfigs ← C_abstract[funname]
        for config in abstractconfigs do
            context, labels ← unpack(config)
            if match(context, funcontext) then
                add labels to children of node in decoratedtree
                break loop
        concreteconfigs ← C_concrete[funname]
        for config in concreteconfigs do
            context, relabelops ← unpack(config)
            if match(context, funcontext) then
                store relabelops for node in table concreterelabels
                break loop
    return decoratedtree, concreterelabels

Function getdependencies(tree)
    D ← DependencyTree() # initialize an empty dependency tree
    foreach word in Linearization(tree) do
        find lowest node parent in tree spanning the word
    foreach leaf in AST T do
        find UD label by traversing unlabelled edges up the AST
        find head by traversing unlabelled edges from the dominating node
        store the word, parent, head function and UD label in D
    return D

Function relabeledges(deptree, concreterelabels)
    foreach node in Table concretelabels do
        relabelops ← concreterelabels[fun]
        foreach relabelop in relabelops do
            (label, words, newlabel, newhead) ← unpack(relabelop)
            for word in children of node in deptree do
                if word matches words then
                    assign new UD label newlabel and head newhead in
                      deptree
    return deptree
```

Algorithm 1: Algorithm for converting ASTs to dependency trees

```
Data: Context for a node in AST nodecontext and context pattern in
      configuration configcontext
Result: True or False

remove branches from nodecontext that are not heads from node context
foreach arg in nodecontext do
    pat ← next(configcontext) # pattern of next argument
    if argmatch(arg, pat) then
        return False;
return True;

Function argmatch(tree, treepattern)
    if treepattern is "?" then
        return True # match everything
    funname, patargs = unpack(treepattern)
    topnode, args = unpack(tree)
    if funname does not match name of topnode then
        return False
    else
        if patargs is Empty then
            return True # Local rule
        foreach argument in args and pat in patargs do
            if not argmatch(argument, pat) then
                return False
        return True
```

Algorithm 2: Algorithm used to match contexts in configurations to AST