Linguistic **I**ssues in **L**anguage **T**echnology – **LiLT**
December 2009

The GF Resource Grammar Library

Aarne Ranta

Submitted, September 2008
Revised, December 2009
Published by CSLI Publications

LiLT volume 2, issue 2 December 2009

The GF Resource Grammar Library

Aarne Ranta, *Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, email* `aarne@chalmers.se`

Abstract

The GF Resource Grammar Library is a set of natural language grammars implemented in GF (Grammatical Framework). These grammars are in a strong sense parallel: they are built upon a common abstract syntax, i.e. a common tree structure. Individual languages are obtained via compositional mappings from abstract syntax trees to feature structures specific to each language. The grammar defines, for each language, a complete set of morphological paradigms and a syntax fragment comparable to CLE (Core Language Engine). It is available as open-source software under the GNU LGPL License. The current coverage is fourteen languages: Bulgarian, Catalan, Danish, English, Finnish, French, German, Italian, Norwegian (bokmål), Polish, Romanian, Russian, Spanish, and Swedish. More languages are under construction.

The library can be used as a resource for language processing tasks, such as translation, multilingual generation, software localization, natural language interfaces, and spoken dialogue systems. The library may also have some language-typological interest as an experiment showing how much grammatical structure can be shared between languages. The focus of this paper is on the linguistic aspects of the library—in particular, what syntactic structures are covered and how the problems arising in different languages have been solved. We will also discuss the nature of the common abstract syntax with respect to translation equivalence.

LiLT Volume 2, Issue 2, December 2009.
The GF Resource Grammar Library.
Copyright © 2009, CSLI Publications.

1 Introduction

GF (Grammatical Framework, Ranta 2004) is a grammar formalism for multilingual grammars and their applications. The design and implementation of GF follows state-of-the-art programming language technology, so that GF has a separate low-level run-time format, a multi-pass compiler, and an interactive development environment. The GF Resource Grammar Library plays the role of the standard library. In this perspective, it is a **software library**, similar to the Standard Template Library of C++ or the Java API. In another, linguistically more familiar perspective, the GF Resource Grammar Library is a set of **parallel grammars**, similar to the Core Language Engine (Alshawi, 1992), the LinGO Matrix (Bender and Flickinger, 2005), and the ParGram grammar collection (Butt et al., 2002).

1.1 Grammars as software libraries

The purpose of a software library is to gather and encapsulate low-level program details, which require expert knowledge, and make them available for non-expert application programmers. In the case of grammars, the most obvious example of such knowledge is morphological inflection patterns. Consider, for instance, regular verb inflection in English. It is presented to the user by a type signature

```
mkV : Str -> V
```

introducing a function named `mkV`, which takes a string (`Str`) and returns a verb (`V`), i.e. a complete inflection table listing all forms of a verb. Thus the function application `(mkV "walk")` yields the forms *walk, walks, walked, walked, walking*. The function `mkV` implements what we call a **smart paradigm**: it analyses of its input string and recognizes regular variations, so that, for instance, `(mkV "cry")` yields *cry, cries, cried, cried, crying*.

By the use of smart paradigms, an application programmer can quickly build a morphological lexicon, which may in itself be the goal of the project. But the lexicon may also be a component of a larger program, such as a translator. In such projects, the GF Resource Grammar Library helps by providing a set of syntactic functions as well. As an example, the function

```
mkCl : NP -> V -> Cl
```

builds a clause (`Cl`) from a noun phrase (`NP`) and a verb. This function takes care of agreement (*she walks* vs. *they walk*), negation (*she doesn't walk*), question forming (*does she walk*), and tenses. The user of the library does not need to worry about agreement features, constituent order, etc. Getting such syntactic functions right requires even more specialized linguistic expertise than morphological functions.

The idea of using a grammar as a software library is, to our knowledge, new in GF. It has required a considerable effort in the design and implementation of the GF programming language: a type system, a module system, compilation techniques, and optimizations. These aspects are covered in other publications (Ranta, 2004, 2007b, 2009, Angelov et al., 2009). The effort made in this work is supplemented by a substantial effort in the library itself. The code included in the library is more than twice in size compared to the implementation of GF.

The GF Resource Grammar Library implements a comprehensive fragment of fourteen natural languages: Bulgarian (Angelov, 2008), Catalan, Danish, English, Finnish (Ranta, 2008), French, German, Italian, Norwegian (bokmål), Polish, Romanian, Russian (Khegai, 2006a), Spanish, and Swedish, and the artificial language Interlingua (Union Mundial pro Interlingua, 2001). Also fragments of Arabic (Dada and Ranta, 2007), Dutch, Hindi/Urdu (Humayoun et al., 2007), Latin, Thai, and Turkish are available, and about 15 more languages are under construction. The availability of such a set of languages as an easy-to-use library has made GF into a tool for several projects in multilingual authoring and generation (Khegai et al., 2003, Burke and Johannisson, 2005), dialogue systems (Ljunglöf et al., 2006, Perera and Ranta, 2007), technical translation (Khegai, 2006b, Caprotti, 2006), multilingual web pages moises-gotal, and tools for controlled languages (Ranta and Angelov, 2009).

1.2 Multilinguality

The GF Resource Grammar Library is roughly divided into morphological and syntactic compoments. The morphological component varies from one language to another, as regards the sets of inflection forms and the inflection engine itself, but uses common naming conventions, such as `mkV` for verb inflection whenever possible. The syntax component displays a stronger parallelism: all languages in the library have a common representation of syntactic structures and structural words such as pronouns. Functions like `mkCl` above are implemented for all languages covered by the library, and so are pronouns like `we_NP`. Thus the function application

```
mkCl we_NP (mkV "walk")
```

yielding the English clause *we walk* can be turned into producing Finnish by just changing the lexical item `"walk"`:

```
mkCl we_NP (mkV "kävellä")
```

yielding *me kävelemme*. The syntax function `mkCl` forms correct sentences in both languages. The morphology function and `mkV` builds inflection tables for verbs in both languages, producing five forms in English and 103 forms in Finnish. The application programmer's advantage is great: she has to learn only one set of functions to master 14 languages, instead of 14 sets.

Now, the linguist reader will certainly have questions concerning the radical approach to multilinguality enabled by GF. The first question is whether all languages really have the same categories and structures. Aren't, for instance, English and Finnish so different that it hardly makes sense to define the same syntactic constructions for them? The second question is whether the same structures, even if they exist, can be used as translation equivalents by just changing lexical items, as in the example with *we walk* above. These two questions will motivate much of the discussion in this paper.

To address the first question, we will show that the GF grammar formalism can indeed give common structures to seemingly very different languages. But we will not make this ideal into a straight-jacket: we will also leave room for structures that are defined only for some languages. However, since the engineering gains of common structures are so high, and since they also give interesting linguistic parallels, we will try to maximize the set of common structures and see how far we can get. Whole Section 4 and the Appendix can be seen as a demonstration of this.

The second question, about translation equivalence, has to do with how the library is applied. We approach this question by separating the levels of **resource grammars** and **application grammars**. Resource grammars are about syntactic structures, such as noun-phrase verb-phrase predication, whereas application grammars are about semantic structures, such as logical predicates. Only semantic structures are expected to guarantee structure-based translation equivalence. Thus the sentences *we walk* and *me kävelemme*, even though they have the same syntactic structure, are not always translation equivalents, because the Finnish sentence often has to be translated by the progressive *we are walking*. To take a more complex example from translation theory, the predicate contained in the English sentence *she swam across the river* is in English expressed by the verb *swim* and the preposition *across*. In the French equivalent, *elle a traversé la rivière en nageant*, a transitive verb *traverser* ("to cross") is used together with an adverbial *en nageant* ("by swimming"). We will show in Section 6 how such situations are handled in GF, and keep in mind in the entire discussion that resource grammar structures do not always support translation equivalence.

1.3 The structure of this paper

To make the discussion technically precise, we introduce the main concepts and notations of GF in Section 2. Section 3 gives an overview of the design principles and the structure of the GF Resource Grammar Library. Section 4 is the central part of this paper, giving a tour of the syntactic structures in the Library. It discusses and motivates the most controversial choices and points out implementation issues for different languages; the full set of rules is given in the Appendix. Section 5 explains how morphology is implemented in the library. Section 6 takes a look at how the library is used, both in application grammars and as a general linguistic resource. Section 7 contains an evaluation of the library, addressing correctness, coverage, and usability. Section 8 gives a brief survey of related work, and Section 9 concludes.

2 GF and multilingual grammars

The most fundamental concepts of GF are **abstract syntax** and **concrete syntax**. An abstract syntax defines a set of **tree structures**. A concrete syntax maps trees into strings and, more generally, to **records**, which resemble **feature structures** as used in many other grammar formalism. Both trees and records are **typed**, by a type system similar to the functional programming languages ML (Milner et al., 1990) and Haskell (Peyton Jones, 2003). While GF has some built-in types and type constructors—strings, record types, function types—types in general are user-defined.

2.1 Abstract syntax

A grammar is a set of **judgements**. The first kid of judgements are those that introduce **categories**, i.e. basic types in an abstract syntax. They are marked by the keyword `cat`. Thus the judgement

```
cat CN
```

defines a category named `CN`, used in the library for common nouns. A category is a set of **trees**, and trees are built by using **functions**, which are defined by judgements marked by the keyword `fun`. Thus the judgement

```
fun House : CN
```

defines a zero-place function (a **constant**) `House`, which does not take any arguments and is hence itself a tree, of type `CN`. Given that we have a category `AP` (adjectival phrases), we can also write a judgement

```
fun Mod : AP -> CN -> CN
```

for building trees by adjectival modification. Thus `Mod` takes an adjectival phrase and a common noun and produces another common noun. If `Big` is a constant of type `AP`, we can build the tree

```
Mod Big House
```

of type `CN`. The type of `Mod` permits iteration, so that we also have

```
Mod Big (Mod Big House)
Mod Big (Mod Big (Mod Big House))
```

and so on. As for the notation, notice that a function is just prefixed to its arguments, and parentheses are only needed for grouping together complex arguments. Thus trees in GF look like Lisp terms, except that the outermost parentheses are usually omitted.

2.2 Concrete syntax

A concrete syntax defines **linearization types** for every category in an abstract syntax. This is done by judgements marked by the keyword `lincat`. Thus the judgement

```
lincat CN = Str
```

says that common nouns have the linearization type `Str` of **strings** (which are actually **token lists**). Linearization types must be obeyed in the judgements that assign **linearization functions** to abstract syntax functions, marked by the keyword `lin`. Thus the judgement

```
lin House = "house"
```

says that the tree `House` is linearized to the string `"house"`. The judgement

```
lin Mod ap cn = ap ++ cn
```

says that a tree of the form `Mod ap cn`, where `ap` is a variable for the first argument and `cn` for the second, is linearized by concatenating (`++`) the linearization of `ap` with the linearization of `cn`. To be type-correct, this rule of course presupposes that the linearization type of `AP` is also `Str`.

Judgements of the `lin` form can be seen as clauses in the definition of a recursive linearization function. The linearization of trees into strings is straightforward evaluation of function applications. Parsing strings into trees is a more involved operation, but it has a general solution that constructs a polynomial parsing algorithm for any given GF grammar (Ljunglöf, 2004).

2.3 Parameters, tables, and variable features

Common nouns in English have two forms, singular and plural (ignoring the genitive case to keep the example simple). They could be handled by distinguishing two categories in abstract syntax, say, `CNSg` and `CNPl`. But a more elegant and powerful solution is to introduce the distinction in concrete syntax only, by defining a **parameter type** of number, and making the linearization of `CN` produce these two forms.

Parameter types are defined by judgements marked by the keyword `param`. Thus the judgement

```
param Number = Sg | Pl
```

defines the parameter type `Number` with its two elements, `Sg` and `Pl`. The judgement

```
lincat CN = Number => Str
```

says that the linearization type of `CN` is a **table** that assigns strings to numbers.

Notice the usage of the double arrow `=>` for table types, in contrast to the single arrow `->` for function types. A table object is defined by using the following syntax, exemplified by a modified linearization rule for `House`:

```
lin House = table {Sg => "house" ; Pl => "houses"}
```

Thus a table must assign a value of the expected value type to every element of the argument type. This can be done by explicitly listing all cases, as here; it can also be done by passing a variable of the argument type from the left of the arrow to the right of it. Such is the case with the number-sensitive linearization rule for `Mod`:

```
lin Mod ap cn = table {n => ap ++ cn ! n}
```

Here the variable `n` of type `Number` is passed to the common noun argument `cn`, which itself is a common noun linearization and hence a table from numbers to strings. The operator `!` is used for **selecting** the value assigned to `n` from the table that is the value of `cn`. As a result of these definitions, the linearization of the tree `Mod Big House` computes to the table

```
table {Sg => "big house" ; Pl => "big houses"}
```

A constant like `House` is thus a **lexeme** rather than a single word: it contains all forms of the word. The concept of a lexeme is in GF generalized from words to complex phrases, such as `Mod Big House`. Since common nouns have different forms depending on number, we say that number is a **variable feature** of common nouns.

2.4 Records and inherent features

Varying the parameter types and linearization types in a concrete syntax is what ultimately enables GF to give the same structure to different languages. Working with the same abstract syntax as before, we can now add a concrete syntax for French. Anticipating the addition of new nouns and adjectives, we have to account for gender agreement (*grande maison* "big house" vs. *grand arbre* "big tree") as well as for the position of adjectives (*grande maison* vs. *maison blanche* "white house").

Agreement is expressed by passing values of **inherent features** to variable features. In French, the gender of nouns is inherent, whereas the gender of adjectives is variable. This is easy to see by looking at any French dictionary, where the entries for nouns indicate what gender a noun has, and there is (with some rare exceptions) no such thing as a masculine and a feminine form of a noun. If we define the gender type by

```
param Gender = Masc | Fem
```

we can write

```
lincat CN = {s : Number => Str ; g : Gender}
```

to indicate that common nouns in French have a variable number (as in English) and an inherent gender (which English doesn't have). The linearization type is now a **record type**, with two **fields**, each having a **label** and a type. The first field here has the label `s`, whose type is tables from numbers to strings. The second field has the label `g` and the type gender.

Adjectives and adjectival phrases in French have both a variable gender and a variable number. Thus we write

```
lincat AP = Gender => Number => Str
```

The adjectival modification rule expresses agreement as follows (ignoring the position for a little while):

```
lin Mod ap cn = {
  s = table {n => cn.s ! n ++ ap ! cn.g ! n} ;
  g = cn.g
  }
```

The value here, as expected, is a **record** giving values of expected types to both fields. The variable `cn` is a common noun, thus itself a record, from which the values of the two fields are extracted by the **projection** operator (`.`). Thus `cn.g` is a gender, and `cn.s` is a table from numbers to strings. The rule can be read as follows:

> In adjectival modification, the noun is prefixed to the adjective. The variable number of the complex noun formed is passed to both the noun and the adjective. The inherent gender of the noun is passed to the adjective, and also inherited by the complex noun that is formed.

Designing the parameter system and, in particular, deciding what is inherent and what is variable, is one of the main tasks in GF grammar writing, and makes it different from unification grammars, where the distinction is not made (Cooper, 2008, Ranta, 2007a). Traditional dictionaries give good hints for how this works for lexical categories, and these hints can be generalized to phrases where these words function as heads. Some traditional grammars make the distinction explicit; for instance, the authoritative French grammar Grevisse (1993) carefully lists the two kinds of features separately when introducing parts of speech. The type checker of GF grammars ensures that features are passed in correct ways; if it reports on a missing feature, this will force the grammarian to think about where it is inherited from.

Interestingly, the features of the main parts of speech tend to be similar in different languages. For instance, if a language has number, gender, and case, nouns tend to have a variable number and case and an inherent gender, whereas adjectives are like nouns except that the gender is variable. These similarities help in building concrete syntaxes for new languages. But some features are not uniform across languages. The position of adjectives in

French (and other Romance languages) is a case in point. A straightforward solution to the problem is to introduce an inherent Boolean feature that says whether an adjective comes before the noun it modifies:

```
lincat AP = {
  s : Gender => Number => Str ;
  isPre : Bool
  }
```

Adjectival modification can use this feature to select the order of the noun and the adjective, as shown below in Section 2.7.

2.5 Discontinuous constituents

A record in GF may contain several fields for strings. In the Resource Grammar Library, this has proved particularly useful for the category of verb phrases, `VP`. An obvious example is from Germanic languages, where inversion places the subject between the finite verb and the rest of the verb phrase. Thus, in Swedish, *hon köper bröd* ("she buys bread") becomes in questions and also after leading adverbials *köper hon bröd*, where the verb phrase is *köper* - *bröd* ("buys" - "bread"). The minimal model for this is a record with two string fields, where the first field is the finite verb and the second field the complement (ignoring tense and agreement):

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

An inversion question, with abstract syntax

```
fun Quest : NP -> VP -> Q
```

can now be linearized in an obvious way:

```
lin Quest np vp = vp.fin ++ np.s ++ vp.compl
```

As an example of how discontinuous `VP`s are built, consider complementation with transitive verbs (`V2`):

```
fun Compl : V2 -> NP -> VP
lin Compl v np = {fin = v.s ; compl = np.s}
```

The technique of discontinuous constituents scales up to the topological structure of Germanic clauses (Diderichsen, 1962) and also to Romance clitics. It thus "rescues" the NP-VP analysis of these languages and thereby permits a compact abstract syntax for predication in the Resource Grammar Library.

2.6 Compositionality

Linearization, a recursive function from trees to strings, looks very powerful—almost like transformations (Chomsky, 1957) or natural language generation (NLG). But linearization is less powerful than transformations and NLG, because it is **compositional**: a concrete syntax is a homomorphic mapping of the free algebra defined by the abstract syntax to the universe of records, parameters, and strings. In precise terms, if

fun f : C_1 -> ... -> C_n -> C

then

lin f : C_1* -> ... -> C_n* -> C*

where the type C* is the linearization type the category C. This notion of compositionality is the same as the one satisfied by *semantics* in Montague grammar (Montague, 1974). Montague's linearization rules, however (the rules converting analysis trees to strings), are not compositional. In fact, he did not formalize these rules at all. Concrete syntax in GF is technically just another semantics of analysis trees, in the model of records, parameters, and strings.

Compositionality is a strong constraint on linearization. It implies that linearization cannot be just whatever function on trees, like transformations or NLG. No reanalysis of subtrees is permitted in linearization, but every tree must have a fixed linearization that it retains in all constructs in which it serves as a subtree. In other words, all abstract syntax trees correspond to fixed concrete syntax objects. This means that a multilingual grammar establishes a non-trivial relation between the languages it involves, and is thereby potentially interesting as a cross-linguistic hypothesis.

2.7 Auxiliary operations

GF is a functional programming language, which means that it uses functions to capture patterns of computation to avoid repeating them. Such functions are recognized by the keyword `oper`. An example is an operation that decides on the mutual order of two strings, as a function of a Boolean parameter:

```
oper prefixIf : Bool -> Str -> Str -> Str =
  \p,x,y -> case p of {
     True  => x ++ y ;
     False => y ++ x
     } ;
```

This definition introduces two new GF constructs: the **lambda binding** of variables to form a function (`\p,x,y ->`), and `case` expressions. These are standard functional programming constructs, which contribute to the power of GF as a programming language.

An example of using the operation `prefixIf` is the French adjectival modification rule, now taking into account that some adjectives appear before the noun, some after (cf. Section 2.4 above):

```
lin Mod ap cn = {
  s = table {n =>
    prefixIf ap.isPre (ap ! cn.g ! n) (cn.s ! n)
```

```
      } ;
    g = cn.g
    }
```

The same function can be used for several tasks, such as ordering Romance clitics and distinguishing between prepositions and postpositions in Finnish. This helps to maintain a language-independent abstract syntax, since we can include clitics in the normal category of noun phrases, and need not have postpositions separate from prepositions.

The keyword `oper` distinguishes auxiliary concrete-syntax functions from the abstract syntax `fun` functions, which construct trees. The term **top-level grammar** is used for the set of rules defined by the `cat`, `fun`, `lincat`, and `lin` judgements. Only top-level grammars are used in parsing and linearization, whereas `oper` and `param` are just auxiliaries: they are eliminated in the process of grammar compilation. In this sense, operations are like macros. Technically, however, they are different from macros, since they enjoy separate compilation: they are not expanded syntactically before compilation, but type-checked and evaluated on their own.

Operations can be **overloaded**, which means that one and the same name can be used for different operations. This is technically possible for any set of operations that have different types: the type checker of GF can then perform **overload resolution** on the actual function calls. Overloading is useful for managing large libraries, since it decreases the number of names the programmer has to memorize.

As an example of overloading, the name `mkCl` is used for building clauses (`Cl`) in different ways. Most of these ways have a noun phrase (`NP`), which can be followed by either a verb phrase (`VP`), a verb (`V`), an adjective (`A`), or a two-place verb (`V2`) with its complement `NP`, to give just a few examples:

```
mkCl : NP -> VP -> Cl
mkCl : NP -> V  -> Cl
mkCl : NP -> A  -> Cl
mkCl : NP -> V2 -> NP -> Cl
```

As we will see in Section 3.4, the Library is rather promiscuous in introducing new instances of overloaded constructors to help the user; since these constructors are operations and not abstract syntax functions, this redundancy does not compromise the compactness of the core of the grammar.

2.8 Grammar composition and partial evaluation

Top-level grammar rules can be reused as operations, via the following mappings:

- a pair `cat` C ; `lincat` $C = T$ becomes `oper` C : `Type` $= T$
- a pair `fun` f : A ; `lin` $f = t$ becomes `oper` $f : A^* = t$, where A^* is the

homomorphic linearization type of *A*

The `opers` created in this way become usable for concrete syntax definitions of other grammars. The combinations of operations correspond exactly to the syntax trees in the original abstract syntax. This technique is usually called **grammar composition**: the abstract syntax of one grammar is used as the concrete syntax of another grammar. The most typical example of this is the use of the resource grammar library in application grammars: the semantic structures of the application are mapped into the syntactic structures of the resource.

Proper examples of grammar composition will be given in Section 6. Just to give a brief example, assume we have the question function `Quest` and the categories shown in Section 2.5 in a resource. Assume a domain semantics that implements a query system about songs and artists (as in Perera and Ranta 2007), with functions such as

```
fun DoesSing : Artist -> Song -> Query
```

(*does x sing y*). Using the library starts with assigning resource categories as linearization types:

```
lincat Artist, Song = NP ; Query = Q
```

Given the Swedish transitive verb `sjunga_V2` (*sjunga*, "sing"), we can now write the linearization rule as

```
lin DoesSing x y = Quest x (Compl sjunga_V2 y)
```

The GF grammar compiler evaluates this into a run-time linearization rule

```
lin DoesSing x y = "sjunger" ++ x.s ++ y.s
```

where the discontinuity of `VP`s is no longer visible. The technique is known as **partial evaluation**. It evaluates everything in the linearization except the subtree variables, which can be evaluated only at run time. All linearization types disappear, except those that are types of some run-time variables. These are exactly the ones that are used in `lincat` judgements in the application grammar—in the present example, `NP` and `Q`, but not `VP` or `V2`.

Partial evaluation can have dramatic effects on the grammar size and its parsing behaviour. For instance, discontinuous constituents are usually associated with "theoretical" categories such as `VP`, and they disappear in compilation if `VP` is not used on the top level in the application grammar. Hence, for instance, an application grammar can be context-free even though the resource grammar isn't. It is due to partial evaluation that applications can use large-scale grammars as libraries and still remain light-weight in processing tasks.

3 An overview of the library

3.1 Background

With a theoretical background in Ranta (1994), GF was first implemented as a full-fledged grammar formalism in 1998 within the project "Multilingual Document Authoring" at Xerox Researche Centre Europe in Grenoble (Dymetman et al., 2000). The leading idea of this project was to develop a tool for producing documents in multiple languages simultaneously. A multilingual document is produced by constructing an abstract syntax tree and linearizing it to different languages. To support meaning-preserving translations, the abstract syntax has to encode a **domain semantics**, i.e. the semantic structures of some domain; it was not considered feasible to have a universal abstract syntax applicable to all domains.

The concept of Multilingual Document Authoring had previously appeared in the WYSIWYM system (Power and Scott, 1998). which was implemented by hard-coded generation rules. GF replaced the hard-coded rules with declarative grammars, which made the generation rules into a replaceable component of the system, and moreover supported the reverse operation of parsing.

Writing multilingual grammars for various domains in GF turned out to be possible: prototypes were built for mathematical proofs, business letters, restaurant phrasebooks, database queries, and medical drug descriptions, covering up to seven languages. At the same time, even simple grammars turned out to require considerable linguistic knowledge. "Difficult" structures such as German word order and French clitics were soon encountered in most domains. Since the rules governing them are independent of domain, the idea arose to create libraries that take care of the linguistic details and give application programmers high-level access to them.

The development of the library started in 2003. Since the first stable version in 2006, two releases have been made every year. While the coverage of grammatical structures and languages is growing, the ambition is to keep the library backward compatible, so that old applications grammars continue to work with new versions of the library.

3.2 Coverage

The coverage of the GF resource grammar library has been guided by its growing set of applications. The first major application was an authoring and documentation system for the KeY software specification system (Burke and Johannisson, 2005). It was followed by the somewhat related WebALT mathematical exercise translator (Caprotti, 2006). These applications demanded a coverage of written technical language and accurate rendering of logical formulas. The TALK project on spoken dialogue systems (Ljunglöf et al., 2006) extended the scope to spoken, more casual language.

In addition to applications, the coverage of the library was defined by *a priori* considerations: we wanted to give, for each language, a complete inflectional morphology, as well as a syntax comparable to the Core Language Engine (Alshawi, 1992), in the form described in Rayner et al. (2000). The lexicon was considered to have a secondary importance: instead of a static lexicon, we wanted to make it easy to build lexica on demand by high-level morphological paradigms (Section 5.3).

The **core syntax**, which is implemented for all languages, has around 60 categories and 200 construction functions. It covers texts built from declaratives, questions, and imperatives, which in turn are built from noun phrases and interrogative phrases, verbs with various subcategorization patterns, embedded sentences and questions, relative clauses, and adverbials. Coordination on different levels (sentences, noun phrases, etc) is also covered.

The lexicon has about 500 lexemes, including 100 structural words, 50 words for the numeral system, and 350 content words. The structural words are presented as core facilities provided by the library, on a par with the syntactic constructors. The content words are used mostly for test and illustration purposes. Since they include many of the oldest and most "worn" words, such as the 207 "basic words" of Swadesh (1955), they usually cover quite well the inflection patterns of each language.

In addition to the core syntax and the lexicon, the library has language-dependent extensions given in separate modules. For instance, the rich tense system of Romance languages is not completely covered by the core syntax, but only by the Romance-specific extensions. For some languages, a lexicon of irregular verbs is provided. Wide-coverage lexica have been constructed for some languages, usually by code generation from other sources, such as the Finnish word list from Kotimaisten Kielten Tutkimuskeskus (2006), the Swedish lexicon SALDO (Borin et al., 2009) with 70,000 lemmas, the Oxford Advanced Learner's Dictionary of English, and the Bulgarian OpenOffice spell checking dictionary.

3.3 Module structure

The GF resource grammar library has two layers: the **user API** and the **core grammar**. The user API for every language *X* consists of three modules:

- `Syntax`*X*, giving high-level access to the core syntax
- `Paradigms`*X*, giving the inflection paradigms used for building a lexicon
- `Extra`*X*, giving access to extensions of the core syntax

The `Syntax` modules have a common interface for all languages in the library, providing exactly the same categories and functions. The `Paradigms` modules are language-specific, but maintain as much uniformity in naming as possible. For instance, every language has functions `mkN`, `mkA`, and `mkV` for constructing nouns, adjectives, and verbs from the forms conventionally used in dictionaries of that language (see Section 5.3). The `Extra` modules are partly overlapping, and less developed for some languages than for others.

The library API `Syntax` is built by using grammar composition (see Section 2.8) from the core grammar, which is not a library but a top-level grammar that permits parsing and generation. Thus the linguists who build the core grammar can test it independently of applications. The main parts of the core grammar are the following:

- `Grammar`*X*, all syntactic combination rules
- `Lexicon`*X*, a test lexicon
- `Lang`*X*, combining `Grammar`*X* with `Lexicon`*X*
- `All`*X*, combining `Lang`*X* with `Extra`*X* and a language-specific lexicon

Thus the syntactic constructs presented in this section and Appendix A are those belonging to the module `Grammar` and reused in `Syntax`. Section 5 gives an overview of what is contained in `Paradigms`.

3.4 Presentation

The user API is a set of overloaded functions, which follows three naming conventions:

- `mk`*C* is an overloaded function that construct objects of type *C*
- *word_C* is a lexical item of type *C*
- *descrC* is any other operation constructing *C*, with the description *descr*

The first two conventions have turned out to be intuitive and easily memorizable for users; the third is avoided as much as possible, but cannot be avoided when there are two or more functions of the same type. The API covers all ways of forming trees that are provided in the core grammar, both directly and with shortcuts. Shortcuts are functions that construct trees directly from subtrees of subtrees, omitting intervening levels. For example, the clause *this is cool* can in English be constructed in several ways, including the following two:

```
mkCl
  this_NP
  (mkA "cool")

mkCl
  (mkNP (mkDet this_Quant sgNum))
  (mkVP (mkComp (mkAP (mkA "cool"))))
```

In the core grammar, there is only one tree, corresponding to the latter one above:

```
PredVP
  (DetNP (DetQuant this_Quant NumSg))
  (UseComp (CompAP (PositA (mkA "cool"))))
```

Its outermost form is a clause built by predication (`PredVP`) from a noun phrase built from a determiner and a verb phrase built with a copula and its complement, which in turn is an adjectival phrase, and so on. The core grammar aims at a minimal set of constructors, which usually take just one or two arguments and result in deep tree structures.

Now, the combination shown by *this is cool* is so common that the user API has chosen to provide a shortcut for building a clause directly from a noun phrase and a lexical adjective; the definition is, indeed,

```
mkCl np adj =
  PredVP np (UseComp (CompAP (PositA adj)))
```

The noun phrase `this_NP` is a shorcut for the quantifier `this_Quant` used without a noun. In addition to conciseness and flatness, the shortcuts have the virtue of skipping theory-laden categories like verb phrase and complement, which would otherwise add to the cognitive load of the non-linguist library user.

Separating the user API from the internals is normal software engineering practice: it allows the library programmer to change the internals while keeping the API constant. This has happened several times during the evolution of the GF Resource Grammar Library, when new linguistic ideas have permitted more general constructors; it is important that such ideas can be implemented without destroying the applications of the library.

Put in a different way, the API and the core grammar are kept separate because their users and goals are different: the API should be easy to *use* for the non-linguist, whereas the core grammar should be easy to *write* for the linguist. **Redundancy** is a distinguishing property: it is useful in the API but harmful in the core.

4 Syntactic structures

In this section, we will go through the syntactic structures of the GF resource grammar library. The purpose is twofold: to give an exact characterization of the linguistic coverage of the library, and to discuss some aspects that *a priori* create problems for a common abstract syntax. The complete list of categories and constructors is given in the Appendix, which is divided into subsections with the same numbers as this section.

4.1 An introductory example

To get an overview of the syntax, let us first consider the tree in Figure 1. It is an abstract syntax tree, whose linearizations in 15 languages are shown in

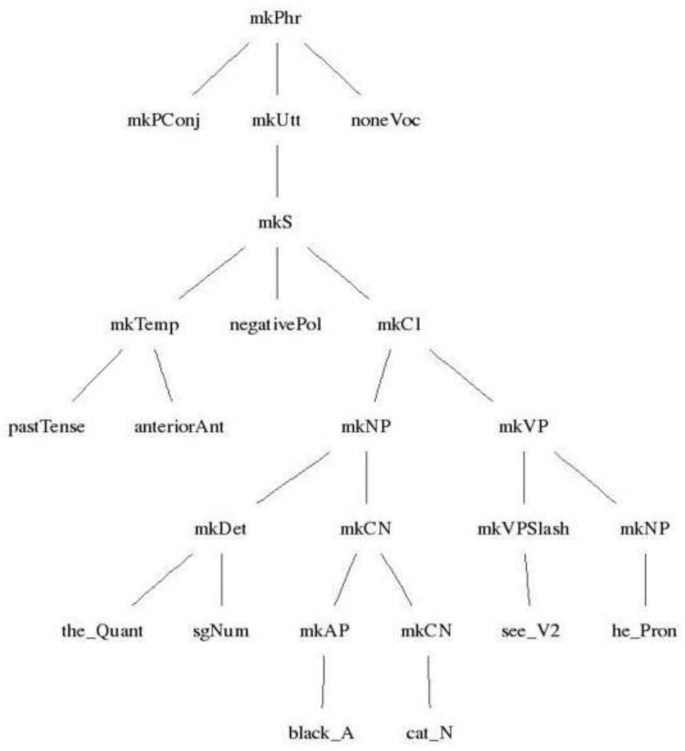

FIGURE 1 Abstract syntax tree.

Figure 2. The construction steps used in Figure 1 are the minimal constructors of the core resource grammar, but we have used the constructor names of the API, which follow the conventions listed in Section 3.4. The relation of each word into its smallest spanning subtree in the abstract tree defines a relation of **word alignment** between languages, a part of which is shown in Figure 3. Both the tree and the alignment figure have been produced by GF's visualization tools.

4.2 Suprasentential level: texts and utterances

The topmost category of the library (not shown in Figure 1) is `Text`, comprising lists of phrases (`Phr`), each terminated with punctuation marks. The punctuation mark is independent of the type of phrase, so that we can write *John walks. John walks? John walks!*

Phrases are build from utterances (`Utt`), which can be declarative sen-

```
черната котка не беше видяла нас
el gat negre no ens havia vist
den sorte kat havde ikke set os
the black cat hadn't seen us
musta kissa ei ollut nähnyt meitä
le chat noir ne nous avait pas vus
die schwarze Katze hatte uns nicht gesehen
le catto nigre non habeva vidite nos
il gatto nero non ci aveva visti
den svarte katten hadde ikke sett oss
czarny kot nie zobaczył nas
pisica neagră nu ne văzuse
чёрная кошка не видела нас
el gato negro no nos había visto
den svarta katten hade inte sett oss
```

FIGURE 2 Linearizations in 15 languages (from above: Bulgarian, Catalan, Danish, English, Finnish, French, German, Interlingua, Italian, Norwegian, Polish, Romanian, Russian, Spanish, Swedish).

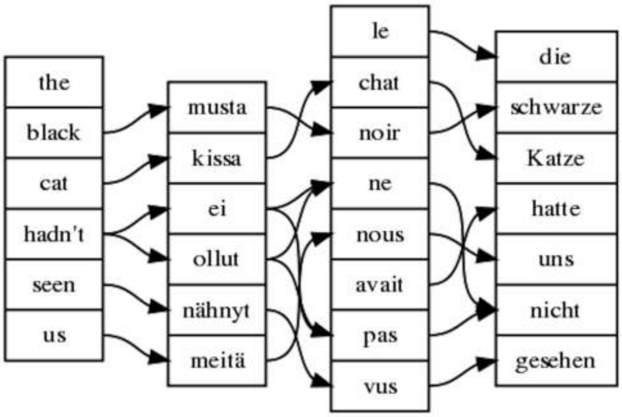

FIGURE 3 Word alignment between English, Finnish, French, and German.

tences, questions, or imperatives, but also subsentential, e.g. noun phrases (*John*), adverbs (*here*), and interrogatives (*who*). When an `Utt` is made into a `Phr`, it can be preceded by a phrasal conjunction and have a vocative attached (*but come here, John*). Subsentential utterances are needed in answers in spoken dialogues and also in titles in written texts.

4.3 Sentence level: polarity, tense, and mood

The main categories from which utterances are constructed are sentences `S`, question sentences `QS`, and imperatives `Imp`. The difference between `S` and `QS` is that `QS` may contain interrogative noun phrases and adverbials. Relative sentences (`RS`) are in many respects similar to questions, but they occure mainly in embedded positions rather than as top-level utterances.

Sentences are constructed from clauses (`Cl`) by fixing their **tense** and **polarity**. A clause itself has variable tense and polarity: parsing *John walks*, *John wouldn't have walked*, etc, yields one and the same clause, but different sentences. Thus the clause category corresponds to the logical predication, or the **sentence radical**, which is common to the different tense and polarity forms.

Imperatives are not formed from clauses, but directly from subjectless verb phrases. They have no tense distinctions, but just polarity (*walk* vs. *don't walk*) and, in various combinations depending on language, person, number, and politeness. Thus e.g. English *walk* has three German variants: the familiar singular *geh*, the familiar plural *geht*, and the polite singular *gehen Sie*.

While polarity is uniformly either positive or negative, the tense parameter varies from one language to another. Germanic languages have a two-dimensional structure, in which one can distinguish **anteriority** (whether the tense has an auxiliary *have* or not) and temporal order (past, present, future). The conditional mood can be seen on a par with the temporal order; morphologically, it is indeed often realized as "the past of the future", both in Germanic (*John would walk* vs. *John will walk*) and Romance (French *Jean marcherait* vs. *Jean marchera*).

The bivalent polarity with the Germanic tense system thus gives us 2*2*4 = 16 different forms of a clause (`Cl`), each of which corresponds to a different top-level sentence (`S`). This system is close to the semantic tense system of Reichenbach (1948), based on the distinction between event and reference times. Romance languages add to this system a specific "simple past" tense, and Slavic languages have a rather different system where **aspect** is central.

The common abstract syntax has constructors for the two polarities and eight Germanic tenses, which are used for building sentences from clauses. This means that the clauses of all languages in the library can be "inflected" in the Germanic system, and hence mapped into the Reichenbachian semantics. However, this mapping can be very rough in languages whose native tense systems are different. Even among Germanic languages, the tense system exemplifies a case in which the resource grammar does *not* guarantee translation equivalence. For instance, the English simultaneous past (*he slept*) is in German normally translated as anterior present (*er hat geschlafen*), whereas the German simultaneous past (*er schlief*) often corresponds to the English progressive past (*he was sleeping*).

Despite the semantic discrepancies, it has turned out useful to have a language-independent abstract syntax of tenses. This is illustrated by the previous example: even though "simultaneous past", etc, doesn't have the same *use* in English and German, the same *forms* do exist, and we can use the same grammatical terms to speak about these different uses. Languages that have a different set of tenses even morphologically, such as French, Finnish, and Russian, have their proper tense systems defined in language-specific modules. The common abstract syntax can still give an approximation of translation equivalence. For instance, Finnish has no form for the future tense, but realizing the abstract future tense with the present usually captures the right meaning.

In addition to tense, many languages have distinctions of **mood**. For instance, Romance sentences come in indicative and subjunctive moods. The mood distinction is, however, grammatically determined: the subjunctive is compulsory in certain kinds of subordinate clauses. Therefore, it does not surface in the abstract syntax at all. Instead, mood is preserved as a variable feature in the linearization type of sentences:

```
lincat S = {s : Mood => Str}
```

The corresponding type for clauses is

```
lincat Cl = {s : Pol => Tense => Mood => Str}
```

Thus, after the tense and polarity are fixed when a sentence is formed, the linearization is still a table that has both indicative and subjunctive forms. The mood is fixed when the sentence is put to a larger context. For instance, sentence-complement verbs in French select the mood of their complent sentence as function of the polarity of the verb. Thus *croire* selects the indicative when used positively (*je crois qu'il dort* "I believe that he sleeps") and the subjunctive when used negatively (*je ne crois pas qu'il dorme* "I don't believe that he sleeps").

In addition to mood, sentences can have a contextually determined word order. German and Scandinavian are prime examples, distinguishing (roughly) between main, inverted, and subordinate clause order. Thus there is, for instance, no abstract syntax category of subordinate clauses, but the word orders are treated as "inflection forms" of sentences, which are in complementary distribution: *he is tired* in German is *er ist müde* in a main clause, *ist er müde* (inverted) after an adverbial, and *er müde ist* in a subordinate clause.

Variable word order is even more common in questions. Most languages have some distinction between direct and indirect questions, which are likewise treated as inflection forms depending on a variable feature of questions.

4.4 Clause level: predication and complementation

The main way of forming clauses is by **predication**: by combining a noun phrase (`NP`) **subject** and a verb phrase (`VP`) **predicate** into a clause (`Cl`). Thus *John* and *walk* are combined to *John walks*. As seen in the previous section, the clause formed in this way covers all polarity and tense combinations of the noun phrase and the verb phrase.

A verb phrase is primarily formed from a verb and its required complements. There are several categories of verbs, corresponding to the possible complements and their combinations, and shown in Appendix, Section 1.2. The division of verbs by the complement is known as **subcategorization**. In the GF resource grammar library, subcategorization is more abstract than in many other grammars, since there are no `PP` (prepositional phrase) complements separate from `NP`. Prepositional phrases are treated like different **cases** of noun phrases. The case is a language-dependent concrete syntax feature, and complement cases are inherent features of verbs. For instance, the linearization type of `V3` is formed from the linearization type of one-place verbs by adding two cases (the operator `**` expresses record extension):

```
V ** {c2 : Case ; c3 : Case}
```

In English, a case is simply a preposition, which can be expressed as a string. Thus the verb *talk* (*to* sb *about* st) has the linearization

```
mkV "talk" ** {c2 = "to" ; c3 = "about"}
```

where `mkV` yields the regular verb inflection. In Finnish, the complement case is a combination of an inflectional case and an optional preposition or postposition. The verb corresponding to *talk* is in a Finnish dictionary given as "*puhua* (jlle jsta)", which means that the first complement is in the allative and the second in the elative case. This linearization is given by

```
mkV "puhua" ** {
 c2 = {c = allative ; p = [] ; isPrep = False} ;
 c3 = {c = elative  ; p = [] ; isPrep = False}
 }
```

An example of a verb with a non-empty postposition is "*puhua* (jnk *puolesta*)" ("speak for sb"),

```
mkV "puhua" ** {c2 =
  {c = genitive ; p = "puolesta" ; isPrep = False}
}
```

When declaring a verb in the abstract syntax, we only need to decide the number of `NP` complements and not whether, for instance, the verb takes direct objects. Abstracting away from complement case is useful because it often happens that a verb takes a direct object in one language but its equivalent in another language takes a prepositional phrase. The price to pay is that we cannot have `V2` coordination in the common abstract syntax; cf. Section 4.8.

The category `V2V` comprises both subject-control verbs, like *promise*, and object-control verbs, like *force*. These categories are distinguished in concrete syntax by a parameter telling where the agreement features of the verb phrase complement come from: *he promised us to behave himself* vs. *he forced us to behave ourselves*. An application grammar may want to distinguish them in abstract syntax because they have different semantic interpretations.

Some of the complementation rules are given explicitly in Appendix, Section 4. Some of them are derived from more primitive rules, which also support the extraction of `NP` complements ("wh movement"), in Appendix, Section 5. Notice that the tense and polarity of sentence and question complements are fixed independently of the main clause. Therefore we use `S` and `QS` rather than `Cl` and `QCl` as complements.

The **copula** (the verb *be*) is not treated as a lexical verb. In some languages (e.g. Russian and Arabic) it is not necessarily expressed by a verb at all. The complement (`Comp`) of a copula can come from different categories: noun phrase, adjectival phrase, and adverbial phrase. The category `Comp` gathers all these complement alternatives and is then made into a `VP` by adding a copula if needed.

The verb phrase category is recursive because of the category `VV` of verbs that take verb phras complements (*want to sleep*), and also because given verb phrases can be modified by adverbs. Sentential adverbs (`AdV`, e.g. *always*) are distinguished from ordinary adverbs (`Adv`, e.g. *here*), not because of their placement (which could be regulated by a parameter) but because they have different combinatorial properties. For instance, a sentential adverb cannot serve as a complement of the copula (*I am here* vs. **I am always*).

4.5 Questions and relative clauses

The category `Cl` gives the basis of forming sentential questions, since a clause can always be converted to a question clause (`QCl`): *does John walk*, *wouldn't John have walked*, etc. Another way to form questions from clauses is by adding interrogative adverb (`IAdv`, e.g. *why*): *why does John walk*. Notice that this cannot be applied to question clauses: **why who walks*. Constructions such as *where is John*, with a noun phrase subject and a copula with an interrogative complement (`IComp`) do not reduce to any of these cases and are hence treated separately.

The most part of the question forming grammar is concerned with clauses that contain interrogative phrases (`IP`, e.g. *who*) as subjects or complements: *who walks*, *whom does John love*. Such questions are very similar to relative clauses (`RCl`), which are from relative pronouns (`RP`, e.g. *that*) as subjects or complements: *that walks*, *that John loves*. In many languages, when interrogative and relative pronouns are used in non-subject noun phrase positions, they are "moved" from that position and put to the front of the clause. The terms used for this phenomenon are **wh movement** and **extraction**; we will mostly stick to the latter term.

In extraction, the role of a complement is occupied by an interrogative or a relative pronoun, which typically appears in the beginning of the clause: *whom did he see*, *that he saw*. The extracted position can also be deep inside a complement, which is what makes extraction to create **unbounded dependencies**: *whom did he say that John saw*.

Extraction is in the resource grammar treated by means of **slash categories**, inspired by GPSG (Gazdar et al., 1985). A slash category *C*`Slash` is like the category *C* but "missing" a noun phrase. Four slash categories are used: Slash sentences (`SSlash`) are formed from slash clauses (`ClSlash`) by fixing tense and polarity. Slash clauses are formed from slash verb phases (`VPSlash`) by adding a subject noun phrase. Finally, slash verb phrases are formed from various kinds of verbs by providing all their complements except one of type `NP`. For `V3`, which has two `NP` complements, there are two ways to do this. The rules for forming `VPSlash` correspond to **slash termination** in GPSG, and they are shown in Appendix, Section 5. The missing complementation rules of Appendix, Section 4 can be derived from these rules by means of the constructor that forms a `VP` from a `VPSlash` and an `NP`.

Yet another way to form a slash clause is by adding a slash adverb (`AdvSlash`) to a complete clause: *he walks with*. This rule permits extraction from an adjunct rather than a complement. With the constructors of Appendix, Section 5, extraction sites can be pushed inside verb phrase complements: *want to use (it)*, *force me to use (it)*. These rules are examples of **slash propagation** in GPSG. They permit, by iteration, missing `NP`s at unbounded depth.

Rather than a general slash category constructor as in GPSG, we have chosen to use a small number (four) of slash categories with fixed names. A general constructor would be technically possible by using dependent types (Ranta, 2004), but, since the number of slash categories in actual use is limited and moreover language-dependent, a productive form would be overgenerating. The correspondences between our categories and those of GPSG are the following:

- `ClSlash` corresponds to `S/NP` and `S/PP`
- `SSlash` corresponds to `S/NP` and `S/PP`

- `VPSlash` corresponds to `VP/NP` and `VP/PP`
- `AdvSlash` corresponds to `PP/NP`

Thus GPSG does not make the `Cl` vs. `S` distinction, whereas we don't make the `NP` vs. `PP` distinction.

Semantically, extraction (slash formation) can be interpreted as function abstraction. Thus *C*`Slash` is as semantic type equal to `NP -> ` *C*. Slash propagation is function composition. Both these interpretations could be expressed by using **higher-order abstract syntax** (functions that take functions as arguments; see Ranta 2004). But the results would be overgenerating, for instance, because of island constraints (Ross, 1967). Thus we have chosen to approach full coverage from below, by using weak rules to cover only the cases that are certain. The approach is inspired by Combinatory Categorial Crammar, CCG (Steedman, 1988, 2000), where a set of combinators, such as function composition, are used instead of unrestricted lambda binding.

Although the abstract syntax type has no cross-categorial slash mechanism, all slash rules are internally defined by a small common set of operations. The linearization type of every slash category is that of the "mother" category extended with an expected `NP` case. For instance,

```
lincat VPSlash = VP ** {c2 : Case}
```

The case is in obvious ways inherited from verbs in slash formation; as explained in Section 4.4, verbs that take complements have complement cases as inherent features.

Relative clause construction with relative pronouns is similar to question construction with interogative phrases. However, while questions are normally used as top-level utterances, relative clauses are used lower in syntax trees for modifying common nouns and noun phrases; see Appendix, Section 6. They must therefore agree in gender and number, and even person: *the system which damages itself*, *we who damage ourselves*. The relative pronoun inherits the agreement features from the noun phrase, and passes them to other parts of the relative clause. With simple pronouns, the features inherited are the same as the features passed, but with modified pronouns (such as *only one of which*), they may be different: *three cars only one of which is black*.

With both interrogative and relative pronouns, the inherent case and preposition of a slash clause usually attaches to the pronoun: French *nombre par lequel 24 est divisible*, in English *number by which 24 is divisible*. This behaviour is known as **pied-piping**. In English and the Scandinavian languages, there is also a variant behaviour, **preposition stranding**, in which the preposition is left in place in the verb frame: *number which 24 is divisible by*. Pied-piping is presented as default behaviour in the library, whereas preposition stranding is accessible via language-specific extensions.

Interrogative pronouns cover, in addition to lexical ones such as *who* and *what*, complex ones formed with interrogative determiners (*which five songs*) possibly without a noun (*which five*). An interrogative can be modified with an adverb (*who in Paris*). Interrogative adverbs can be formed from noun phrases by prepositions (*with whom*), but there are also lexical ones (*when*, *where*, *why*). Relative pronouns are built recursively from a base "identity pronoun" (*which, who*) by adding prepositional phrases (*only one of the parts of which*).

4.6 Noun phrases and determiners

Noun phrases are formed in three main ways: from common nouns (`CN`) using determiners (`Det`), e.g. *the man*; from proper names (`PN`), e.g. *John*, and from personal pronouns (`Pron`), e.g. *he*. Determiners can also form noun phrases alone, without a noun (*these five*). Similarly, nouns without determiners can be used as noun phrases, so-called mass terms (*beer*). A complete noun phrase can be modified by a predeterminer (`Predet`), such as *only*. It can also be postmodified by an adverb (*Paris today*) or by a relative clause (*Paris, which is here*).

Determiners are a category with a complex internal structure. Following Rayner et al. (2000), we distinguish three parts: `Quant` (quantifier), `Num` (cardinal numeral), and `Ord` (ordinal, including superlatives). The quantifier is the main part of the determiner, in the sense that both `Num` and `Ord` are optional. Quantifiers have both singular and plural forms, which are sometimes distinct (*this-these*), sometimes identical (*some*), depending on language. Possessives pronouns are quantifiers formed from personal pronouns, e.g. *his* from *he*. Also the definite and indefinite articles (*the*, *a*) are treated as quantifiers.

The inherent number of the determiner is set by the `Num` component. This component can be phonologically dummy: there is no number word, but just an inherent grammatical number. Non-dummy numerals are either verbal (*fifty-six*) or symbolic (*56*). The ordinal is an optional part of the determiner, but can likewise be verbal (*fifty-sixth*) or symbolic (*56th*). Also superlative forms of adjectives behave syntactically like ordinals (*best*).

The verbal numeral system from 1 to 999,999 has been implemented for 90 languages in GF (Hammarström and Ranta, 2004). This implementation shows some of the power of concrete syntax in GF, which makes it possible to deal with different groupings, such as Hindi *lakh*s (hundreds of thousands) as opposed to Western thousands, and even to some extent with different bases of the numeral system. Adapting these number systems to the resource library required supplementary work on case and gender inflection and on the ordinals, since the original grammar set covered only the enumeration cardinals.

Even symbolic numerals are language-dependent. This is obvious with ordinals (*3rd* in English is *3e* in French), but occurs even with cardinals (1.000.000 in Finnish is 1,000,000 in English). In Finnish, numerals even get case endings (*13:ssa* "in 13").

Using a quantifier alone as a noun phrase often requires a special substantival form. Thus Italian *quel* ("that", masculine) has the substantival form *quello*, and English *my* has the form *mine*. Articles can sometimes be used substantivally (e.g. German *der* "the" and *ein* "a", with some case forms differing from the use with nouns). We have in those cases used equivalents of *it* for the definite and equivalents of *one* for the indefinite article, which also matches with the standard translations of the German determiners when used substantivally.

Articles are in many languages realized by non-lexical means. For instance, the definite article in Scandinavian languages is an inflectional feature passed to the noun that is determined. In Finnish, it is a feature shown only in some constructs, e.g., when a numeral is present. Thus *kolme taloa* covers both (*the three houses* and *three houses*), but the definite use behaves as plural in the agreement with a verb, whereas the indefinite use behaves as singular. Because of such subtle differences, the definiteness distinction has been found appropriate even for languages like Finnish, where it sometimes leaves no trace and therefore creates ambiguities.

In Romance languages, pronouns have unstressed clitic forms which, when used as complements (Section 4.4), are placed before the verb. From the abstract syntax point of view, clitics are just inflection forms of pronouns. Their placement is regulated by the use of parameters and discontinuous constituents, largely in the same way in the different languages of the family; cf. Section 7.4.

4.7 Common nouns, adjectives, and adverbs

Common nouns (`CN`) are phrases ultimately formed from lexical nouns (`N`) as their heads, i.e. parts determining most inherent features and receiving most variable features. Common nouns can be modified by adjectival phrases (*old man*) and relative clauses (*man who walks*). There are also relational nouns, which take noun phrase arguments, one for `N2` (*mother of x*) and two for `N3` (*connection from x to y*). Nouns can moreover be modified by embedded sentences (*fact that she sleeps*) and questions (*question where she sleeps*). For most nouns such combinations make little sense, but the library leaves this for applications to decide.

The library does not distinguish mass nouns from other common nouns. This can result in semantically odd expressions, when countable nouns are used as mass terms. But the distinction can be forced in an application grammar by splitting the common noun category into two semantic categories; these two categories can both use `CN` as their linearization type.

Tha category `CN` is the same as what is called `NBAR` in the Core Language Engine and many other grammars, with reference to the X-bar theory (Jackendoff, 1977). The most logical name would be `NP`, in analogy with `AP` and `VP`, but this name is traditionally (and in X-bar theory) used for determinate noun phrases, as we did in Section 4.6. The naming `N`-`N2`-`N3` is analogous to `V`-`V2`-`V3` in Section 4.4 above. It can be motivated by semantics: these expressions denote 1-, 2-, and 3-place relations, respectively.

Adjectival modification has in many languages variable word order, e.g. in French: *grande maison* ("large house") vs. *maison bleue* ("blue house"). The order is regulated by using a Boolean parameter saying whether the adjective is postfix (cf. Section 2.7). In some languages, unmodified and modified common nouns have different properties, which can be controlled by using a parameter. Thus the Swedish definite article is realized by just inflection for simple nouns, whereas modified nouns require an additional article word (*bil* "car", *bilen* "the car", *den svarta bilen* "the black car").

The principal ways of forming an adjectival phrase (`AP`) are as positive and comparative forms of one-place adjectives (*warm*, *warmer than Paris*), and by providing arguments to relational adjectives (`A2`: *divisible by 3*). Sentence and question complements are defined for all adjectival phrases (*good that she sleeps*), although the semantics is clear only for some adjectives. An adjectival phrase can be modified by an adadjective (`AdA`, *very good*).

No distinction is made between adjectives that have degrees and ones that do not. This can lead to semantic overgeneration, in both comparison (*3 is more prime than 2*) and adadjective application (*3 is very prime*). The iterated application of `AdA` is likewise overgenerating: while *very very good* is all right, *very too good* is not. This problem is related to the previous one: while *good* has degrees, applying *too* to it results in a non-degree adjectival phrase.

The two main ways of forming adverbs are from adjectives (*quickly*) and by prepositions from noun phrases (*in the house*). Comparison adverbs have a noun phrase or a sentence as object of comparison (*less quickly than John*, *less quickly than I ran*). Adverbs can be modified by adadjectives, just like adjectives (*very quickly*). Subordinate clauses, formed from subjunctions with sentences, can function as adverbs (*when he arrives*). Adadjectives can be formed from adjectives, either by the same morphological means as adverbs (*unexpectedly*) or by different means (in Finnish, *odottamaton*, "unexpected", gives the `Adv` *odottamattomasti* and the `AdA` *odottamattoman*).

Some types of adverbs are missing, e.g. prepositionless time adverbials (*this year*). They can of course be easily introduced in the lexicon, but a set of functions forming time adverbials syntactically would be more satisfactory.

4.8 Coordination

Coordination means the use of conjunctions (such as *and*) for combining lists of expressions in the same category. The list can have length 2 (*X and Y*) or more (*X, Y, Z and U*), and the library allows coordination of five categories: adverbs (*here or there*), adjectival phrases (*cold and warm*), noun phrases (*she or John*), relative clauses (*who walks or whom she loves*), and sentences (*he walks and she runs*). Conjunctions are given in the lexicon of structural words. They can be either simple (*and*, *or*) or discontinuous (*both-and*, *either-or*).

Many categories are missing from the list of coordination rules in the common abstract syntax, because it would be difficult to maintain them. For instance, two-place verbs (`V2`) can generally be coordinated only if they have the same complement case; see Section 4.4. Since cases are language-specific, and we don't want to reject constructions at linearization phase (which would make the library unreliable), we cannot include `V2` coordination in the language-independent resource. In a language-dependent extension, one can of course define special categories of two-place verbs at a higher precision to permit their coordination, and coerce these categories to `V2` to inherit the already defined `V2` combinations.

Verb phrase (`VP`) coordination is another interesting issue. For most languages, full coordination in the category `VP` is neither possible nor powerful enough. One reason is tense: if the coordination of verb phrases were to produce a verb phrase, this verb phrase would have to be inflectible in all tenses. But the conjuncts need not have the same tense: *John has left or will leave*. Language-specific extensions implement different variants of `VP` coordination.

The abstract syntax of the library makes no generalization amounting to the coordination of an arbitrary category *X*. But the concrete syntax does so by using generic, dependently typed coordination operations. These operations deal with the propagation of variable and inherent features in a general manner. Thus, if

```
lincat X = {s : V => Str ; i : I}
```

then for `[X]` (lists of trees of type `X`),

```
lincat [X] = {s1,s2 : V => Str ; i : I}
```

where the string field `s` is split into two fields. The conjuction word is placed between these fields, whereas the elements of `s1` are separated by commas. The variable `V` feature is propagated to both these fields. For the inherent features, a calculus for conjoining the values is needed. For instance, gender in Romance works like a product in which the Masculine is 0 and the Feminine is 1. This is definable as the operation

```
conjGender : Gender -> Gender -> Gender = \m,n ->
```

```
    case m of {Fem => n ; _ => Masc}
```

The above explanation of coordination gives another reason why `VP` is problematic: the general formula for coordination does not apply to discontinuous constituents.

4.9 Idiomatic constructs and structural words

The library contains some functions that are common to languages but not as uniform syntactically as the constructs introduced so far. Existentials is a typical example. They have a common semantic structure, which is expressed by different syntactic means. English uses a copula with a special formal subject *there*: *there is a house*. German has an ordinary pronoun as formal subject *es* ("it"), and the verb is *geben* ("give"): *es gibt ein Haus* ("it gives a house"). Even the closely related Romance languages differ from each other: the French phrase is *il y a une maison* ("it has a house there"), the Italian is *c'è una casa* ("there is a house"), and the Spanish, *hay una casa* ("(it) has a house", with a special form of the verb "have").

Constructs like existentials are very common and therefore often needed by the users of the library. Usually they can be constructed in the "regular" resource grammar, but since they can get widely different syntax trees in different languages, this treatment would miss an important structure. We have thus decided to have a collection of idiomatic constructs, which cannot be captured by regular grammatical means. They include existentials, impersonal constructs (*it is cold*), generics (*one sleeps well here*), and cleft constructions emphasizing a noun phrase or an adverb (*it is John who did it*; *it is here she slept*).

The repository of idiomatic constructions is no doubt incomplete. In addition to common constructs, there are many language-specific ones. For instance, Finnish has localized existentials (*tuolla vuorella on talo*, "there is a house on that mountain"), which cannot be reduced to ordinary existentials like their English translations can.

The language-independent library includes a list of structural words. Their constructors in the abstract syntax use English words postfixed by category names, such as `here_Adv`, `by_Prep`. The constructors are meant to stand for precise syntactic and semantic functions; in some cases they are therefore disambiguated by additional notes, e.g. *by* expressing agent as opposed to *by* expressing means. The disambiguated variants of an English word are likely to get different linearizations in different languages. More structural words are added in language-specific lexica, often expressing distinctions that not all languages make. A typical example of this is the personal pronoun system, where the gender distinctions made in each person vary from one language to another.

The division between open and closed lexical classes is not rigid. Thus, for instance, while verb-complement verbs (`VV`) in general appear in the open lexicon, some of them occur among the structural words. This is mainly for traditional reasons: e.g. in English, verbs like *must* have a special behaviour of auxiliaries, and are traditionally explained in the grammar and not only in the lexicon. In other languages, e.g. French, there is nothing special about such verbs, except that they may be irregular.

5 Morphology and lexicon

The resource grammar library aims to provide a complete inflectional morphology for each language it includes. By this we mean that

- all lexical categories are included;
- the linearization types of lexical categories comprise all forms;
- there are paradigm functions sufficient for constructing all forms.

The morphology component is usable as a linguistic resource independently of the syntax component. This is the main reason to make it complete, even to cover forms that are not used in the syntactic constructs of the library. The technique we use is based on typed functional paradigms as in the Zen toolkit (Huet, 2005). This idea is in GF complemented with regular expressions, inspired by XFST (Xerox Finite-State Tool, Beesley and Karttunen 2003).

5.1 Inflection paradigms

A **paradigm**, in the sense of GF, is an operation (`oper`), which takes as its arguments one or more strings and yields as its value a linearization with all forms required by the type. For example, the linearization type of Italian nouns is

```
  N = {s : Number => Str ; g : Gender}
```

The first step of a morphology implementation is to privide a **worst-case function**, which is in all cases sufficient for determining an object of the desired type. For Italian nouns, this function takes three arguments: the singular and plural form strings, and the gender:

```
  mkN : (uomo, uomini : Str) -> Gender -> N
```

The function type notation (*x,...,y* : *A*) −>*B* displaying variable names *x,...,y* is equal to *A* −> ... −> *A* −> *B*, with as many *A*'s as there are variables in the list. The morphology libraries make a heavy use of this notation, because it provides documentation: the variable names can be chosen to look like examples of strings that can be given to the function as arguments.

Paradigms such as usually listed in grammar books can be defined by using the worst-case function. Here is a set of ones for Italian, with names that display example words as a part of their documentation. The operation `init` returns the initial segment of a string, dropping its last character, e.g. `init "vino"` is `"vin"`.

```
vinoN  x = mkN x (init x + "i")  Masc
melaN  x = mkN x (init x + "e")  Fem
olioN  x = mkN x (init x)        Masc
saccoN x = mkN x (init x + "hi") Masc
saleN  x = mkN x (init x + "i")  Masc
carneN x = mkN x (init x + "i")  Fem
tramN  x = mkN x x               Masc
```

The lexicon writer assigns one of these paradigms to each word she wants to add to the lexicon. If the word is so irregular that there is no suitable paradigm, she uses the worst-case function.

5.2 Smart paradigms

The above rather traditional notion of paradigms was the one primarily used in the early versions of the library. Its advantage is its familiarity from school grammars; its disadvantages are that the number of paradigms can grow quite large, and that it can be difficult for the user to make correct choices. A considerable improvement results from the introduction of **smart paradigms**, which analyse the ground form and select the inflection accordingly. Smart paradigms use pattern matching with regular expressions. Here is a smart paradigm dealing with a part of Italian nouns. It is called `regN`, because it captures the regular inflection behaviour.

```
regN : Str -> N = \x -> case x of {
  _ + "io"              => olioN x ;
  _ + ("c"|"g") + "o"   => saccoN x ;
  _ + "o"               => vinoN x ;
  _ + ("c"|"g") + "a"   => mkN x (init x+"he") Fem ;
  _ + "a"               => melaN x ;
  _ + "ione"            => carneN x ;
  _ + "e"               => saleN x ;
  _ + "à"               => mkN x x Fem ;
  _                     => tramN x
  }
```

The regular expression _ matches any string. *P* + *Q* matches any concatenation of *P* and *Q*. *P* | *Q* matches anything that either *P* or *Q* matches.

Since the introduction of regular pattern matching and smart paradigms, morphology implementations in GF don't necessarily need the traditional paradigms at all. Notice that the "intelligence" of a smart paradigm can be increased almost *ad libitum* by adding special cases such as the ending *ione* above. A limiting case would be to include all irregular words as patterns in a smart paradigm. But this would still not cover cases like the English verb *lie*, which is inflected in two different ways (*lie-lay-lain* vs. *lie-lied-lied*).

The lexicographer now needs only the smart paradigm and the worst case function to define Italian nouns. To make the naming still simpler, the library uses the overloaded name `mkN` for both functions—thus, both the syntax functions and paradigms that construct a *C* have the name `mk`*C*.

```
mkN : (vino : Str) -> N
mkN : (uomo, uomini : Str) -> Gender -> N
```

5.3 Paradigm libraries

What forms and parameters are needed in paradigms depends on language and category. However, just like in syntax, the user is helped by uniformity in the library. Thus the library defines, in all languages, the functions `mkN`, `mkPN`, `mkA`, and `mkV`, which take just one string as their argument and produce the most likely inflection of desired type. More arguments can be provided to give more forms and parameters, the most complex case being the worst-case function.

A paradigm library exports all functions that are needed for building a lexicon. In addition to inflection paradigms, this includes names of inherent features (such as `feminine`) and functions for words that take arguments. Thus for two-place verbs, there are two variants of `mkV2`: the default one that forms regular transitive verbs, and the worst-case one that can handle any verb inflection and any complement case or preposition:

```
mkV2 : Str -> V2          -- kill
mkV2 : V -> Prep -> V2    -- speak about
```

6 Using the library

The main usage of GF Resource Grammar Library is as a standard library for programmers writing natural language applications. The library is expected both to save work and to improve quality, as compared with hand-written grammars. At the same time, the library is a linguistic resource that can be used in traditional natural language processing. In this section, we give brief descriptions of a few different ways of using the library. In the next section, we refer to these applications when evaluating the library.

6.1 Application grammars

An application grammar usually starts with a **domain ontology**, which is expressed as an abstract syntax in GF. Concrete syntaxes permit natural language input and output of semantic content in the domain. Multilingual domain grammars can be used for translation via the abstract syntax, but this is not always exploited: the purpose of multilingual grammars may be just

```
abstract Music = {
cat
  Request ; Kind ; Property ;
fun
  Play : Kind -> Request ;
  PropKind : Kind -> Property -> Kind ;
  Song : Kind ;
  American : Property ;
}

concrete MusicGer of Music =
  open SyntaxGer, ParadigmsGer in {
lincat
  Request = Phr ;
  Kind = CN ;
  Property = AP ;
lin
  Play k = mkUtt (mkImp famImpForm (mkV2 "spielen")
                                         (indefNP k)) ;
  PropKind k p = mkCN p k ;
  Song = mkCN (mkN "Lied" "Lieder" neuter) ;
  American = mkAP (mkA "amerikanisch") ;
}
```

FIGURE 4 Abstract and concrete syntax of an application grammar.

to localize the same system in different languages. The abstract syntax is the interface between the languages and the rest of the system. It also provides a completeness check of concrete syntaxes, guaranteeing that exactly the same domain ontology is covered in all languages.

Domain ontology and concrete syntax

As an example of an application grammar, let us consider a music player dialogue system (a subset of the one studied in Perera and Ranta (2007)). The user of the system can say e.g. *play an American song*. To make the structure of the application clear, we show in Figure 4, for the first time in this paper, entire GF modules: an `abstract` syntax module `Music` and a concrete syntax `MusicGer`. (cf. Ranta 2007b for details of the module system).

The concrete syntax in Figure 4 uses resource categories as linearization types of the domain categories, and resource functions for defining the linearizations of domain functions. Using a resource is indicated by the keyword `open`. For instance, the tree `Play (PropKind Song American)` gets linearized *spiele ein amerikanisches Lied*.

```
interface LexMusic = open Cat in {
oper
  play_V2 : V2 ;
  song_N : N ;
  american_A : A ;
}

incomplete concrete MusicI of Music =
  open Syntax, MusicLex in {
lincat
  Request = Phr ;
  Kind = CN ;
  Property = AP ;
lin
  Play k = mkUtt (mkImp famImpForm play_V2
                                    (indefNP k)) ;
  PropKind k p = mkCN p k ;
  Song = mkCN song_N ;
  American = mkAP american_A ;
}
```

FIGURE 5 A lexicon interface and a functor for Music.

Functor implementation

The concrete syntax for other library languages is basically the same as for German, except for the lexical units included. This is because the resource interface Syntax provides the same functions for all languages, and they are most of the time used in the same ways in different languages. This common structure can be exploiting for sharing code between languages. The technique that makes this possible is **functors**, also known as **parametrized modules**. A functor is a module that depends on some **interfaces**, i.e. modules that declare constants but don't implement them. The resource grammar API module Syntax is such an interface; its implementations are the library modules SyntaxEng, SyntaxGer, etc. A typical functor in GF depends on two interfaces: the Syntax API provided by the library and a **domain lexicon** written by the application programmer.

In the case at hand, the domain lexicon interface declares a verb *play*, a noun *song*, and an adjective *American*. Their types are resource grammar categories inherited from the module Cat. The GF keyword for functors is incomplete, prefixed to the module header. Figure 5 shows an interface and a functor for Music.

The code in a functor works for any language for which its interfaces are instantiated. To make the functor MusicI in Figure 5 work for Finnish,

```
instance LexMusicFin of LexMusic = CatFin **
  open ParadigmsFin in {
oper
  play_V2 = mkV2 "soittaa" ;
  song_N = mkN "laulu" ;
  american_A = mkA "amerikkalainen" ;
}

concrete MusicFin of Music = MusicI with
  (Syntax = SyntaxFin),
  (LexMusic = LexMusicFin) ;
```

FIGURE 6 A Finnish lexicon instance and functor instantiation.

we write an instance `LexMusicFin` of the domain lexicon. An instance of `Syntax` is given in the resource grammar library. Hence we can instantiate both interfaces of the functor `MusicI` and obtain a complete concrete syntax for Finnish, as shown in Figure 6. When porting the grammar to a new language, we need only these two modules: a domain lexicon interface (which requires only knowledge of words and their inflection paradigms), and a functor instantiation (which is mechanical to write).

By-passing the functor implementation

We said above that languages use the same structures "most of the time". For the cases where they don't, GF provides three possibilities. Let us use the function `Play` from `Music` as an example. The functor `MusicI` above implements `Play` with a structure using the familiar form of the second person singular imperative (`famImpForm`): *spiele ein Lied* ("play a song"). Assume the German grammarian wants the polite form instead: *spielen Sie ein Lied*. Then she wants to use another linearization rule,

```
lin Play k =
  mkUtt (mkImp polImpForm play_V2 (indefNP k))
```

The first way to force this is not to define the linearization of `Play` in the functor at all, but to do it separately in each language. This approach gives maximal freedom for using different structures. Following it to the extreme is to avoid the use of functors altogether. This way can be chosen by grammarians who are pessimistic about the similarities between languages. They can still use the resource grammar library, but separately for each language.

The second way to by-pass the functor implementation is a moderate variant of avoiding the functor. The functor maybe works fine for most languages, with just a few exceptions. Then we can use it as it is for those languages, and make explicit exceptions for the others. The GF mechanism for this is called **restricted inheritance**, which is expressed by the minus symbol (−) prefixed to a list of excluded functions:

```
concrete MusicGer of Music = MusicI - [Play] with
  (Syntax = SyntaxGer),
  (LexMusic = LexMusicGer) **
 open SyntaxGer, LexMusicGer in {
  lin Play k =
    mkUtt (mkImp polImpForm play_V2 (indefNP k)) ;
  }
```

The restriction list (here `[Play]`) can contain any number of categories and functions, which can then be defined separately for the language in question.

However, abandoning the functor in this example would miss an important point: that an imperative is used in all languages, just in different forms of politeness. This leads us to the third solution: to make the functor more general, by introducing in `LexMusic` a parameter that decides what form of imperatives the user can choose when addressing the system:

```
oper systemImpForm : ImpForm
```

This parameter is defined separately in each domain lexicon instance, and the functor uses the rule

```
lin Play k =
  mkUtt (mkImp systemImpForm play_V2 (indefNP k))
```

Generalizing functors can go further than this, because functor parameters need not be atomic constants but can be functions as well. Consider the case in which a French grammarian wants to use the infinitive instead of the imperative: *jouer la chançon* ("to play the song"). The parameter we need is a function for building requests from verb phrases:

```
oper systemRequest : VP -> Utt
```

The functor now says

```
lin Play k =
  systemRequest (mkVP play_V2 (indefNP k))
```

The domain lexica define `systemRequest` in different ways:

```
systemRequest vp
  = mkUtt vp                       -- Fre
  = mkUtt (mkImp polImpForm vp)   -- Ger
  = mkUtt (mkImp famImpForm vp)   -- Fin
```

In general, by-passing a functor implementation in a multilingual grammar is similar to structure-changing **transfer** in translation systems. Due to partial evaluation, transfer such as shown in this section can be performed at compile time, so that run-time translation is purely interlingual.

Variants in concrete syntax

In the dialogue system experiment of (Perera and Ranta, 2007), the German corpus permitted several ways of making requests to the system: the familiar and the polite imperatives, as well as the infinitive: *spiele ein Lied*, *spielen Sie ein Lied*, *ein Lied spielen*, and also *kannst du ein Lied spielen*, *können Sie ein Lied spielen* ("can you play a song"). Since other languages may have different sets of such ways and since there is no semantic difference between these ways from the functionality point of view, we don't want to distinguish them in the abstract syntax. Instead, we can use the GF construct for **variants** (|), which, for any type in concrete syntax, constructs a list of alternative objects of that type. If we choose the last alternative functor implementation of the previous section, we thus write

```
systemRequest vp =
  mkUtt vp |
  mkUtt (mkImp (polImpForm | famImpForm) vp) |
  mkUtt (mkQCl (you_NP | youPol_NP) can_VV vp)
```

Another, simpler example of the use of `variants` is in the lexicon:

```
song_N =
  mkN "Lied" "Lieder" neuter |
  mkN "Song" "Songs" masculine
```

Combinations of variants may easily produce hundreds of alternative expressions for some requests.

Variants are useful in application grammars, since they neutralize syntactic and lexical distinctions that are irrelevant on the chosen level of abstraction. In the resource grammar itself, however, they are avoided. This is because the library cannot anticipate what distinctions may turn out relevant in future applications. Contracted and uncontracted negations (*don't sleep* vs. *do not sleep*) in English are a typical example. In many applications, they can certainly be treated as variants. But in some applications, stylistic reasons may exclude one or the other. The resource grammar library, which has to cater for all uses, treats them as separate constructs and leaves it to the user to decide whether they are variants or not. In general, grammars used for parsing need more variants than grammars used for generation.

6.2 Corpus and treebank generation

Any GF grammar can be used for generating corpora, the richest form of which is a **multilingual treebanks**: a list of trees with linearizations in different language. This has been an important tool in regression testing of the library itself, but it is also usable for other linguistic tasks, such as in the evaluation of translation systems, and even for training statistical language processing tools with synthesized data (Jonson, 2006).

A treebank can be generated from a list of trees (by linearizing them) or strings (by first parsing and then linearizing). GF development tools also support random and exhaustive generation. Using the raw resource grammar (`Lang`*X*) for this gives lots of semantically anomalous sentences, and certainly a strange distribution, because rare constructs are on a par with common ones. For instance, embedded sentences have the same probability as noun phrases to become subjects of clauses. GF supports the assignment of weights to tree constructors to relieve this problem. But the best synthetic corpora and treebanks are obtained from semantics-driven application grammars.

6.3 Reusing linguistic information

The library contains linguistic information of a very general kind, and this information can be reused in other formats. For instance, a morphological lexicon created by using the resource paradigms can be converted to finite state transducers in the XFST format (Beesley and Karttunen, 2003) and to SQL databases by using techniques developed in Forsberg and Ranta (2004).

Any GF grammar can be converted to a context-free approximation in the language model format required by speech recognition systems such as Nuance (Bevocal Inc., 2005), and further into finite-state approximations as required by HTK/ATK (Young et al., 2002); see Bringert (2007) for the supported formats and conversions. There is also a lossless compilation of grammars to Javascript code. This makes it possible to construct dynamically changing multilingual web pages via grammar-based translation (Meza Moreno and Bringert, 2008).

In each of the cases mentioned, GF can be seen as a high-level front end to the other formats, exploiting their standardized and optimized back-ends but adding a module system and a type-checker that help in large-scale engineering and prevent run-time errors.

6.4 Text parsing

For the task of **text parsing**, the library alone is usually not sufficient. There are three reasons for this. First, the coverage is too restricted, and parsing unconstrained text would require constructs that would compromise the quality of the library. Secondly, due to its abstractness, the raw resource grammar is not optimal for large-scale parsing; some exact figures can be found in Section 7.2. Thirdly, the combinations provided by the library may create undesired ambiguities. However, all these problems can be solved by building extensions to the grammar (the coverage problem), and by changing the tree structures via an application grammar (the efficiency and ambiguity problems). The key technique is the **flattening** of tree structures, of which examples are shown in Section 2.8 and Section 3.4.

The semantics project of Bringert (2009) targeted the coverage of the FRACAS test suite (Kamp et al., 1994). In addition to a lexicon, he needed to add a dozen rules to the English resource grammar, i.e. less than 10% of the size of the available resource grammar.

6.5 Semantics

Since the abstract syntax of the resource grammar is not a semantic structure, its trees can be meaningless or ambiguous. However, the structure is not far from the analysis trees of Montague grammar (Montague, 1974) and the Quasi-Logical Form (QLF) of CLE (Alshawi, 1992). Much of the ambiguity can therefore be treated as underspecification. Therefore, it is possible to translate most of the abstract syntax of the resource grammar to predicate calculus by compositional semantic rules (Bringert, 2009). This translation gives logical semantics simultaneously to all resource languages. On the top of the translation, Bringert built an interface to the first-order theorem prover Equinox (Claessen, 2005) and a web interface for natural language inference tasks (Bringert, 2009).

7 Evaluation

Corpora and treebanks are traditionally the main devices for testing grammar implementations. They are important for the GF Resource Grammar Library as well, but they are not sufficient when evaluating a grammar that is not only used for parsing and generation but also as a software library.

In this section, we will first summarize the goals against which the library should be evaluated. Then we will summarize some results in relation to these criteria.

7.1 Criteria

The following have been used as the main criteria of evaluation for the library.

- **Syntactic correctness**. The user must be able to trust the library: whatever use of library functions is type-correct and passes the GF grammar compiler must result in grammatically correct expressions in the target language.
- **Semantic coverage**. The user must be able to find some ways to express any content she wants to express.
- **Usability**. The library must be intuitive and easy to use for an average programmer who knows some GF and is fluent in the target language. In particular, it should save work in comparison to writing grammars by hand.

The following properties are important for grammar libraries in general, but they are automatically fulfilled by any implementation that passes the GF grammar compiler.

- **API completeness**. A grammar must implement all constructs of the API.
- **Failure-freeness**. Linearization must give a result for all combinations of syntactic constructions.

The following properties are desirable, but secondary to the main criteria.

- **Efficiency**. Parsing and linearization must take only a short time and a small amount of memory.
- **Naturalness**. The linearizations must in each language be natural and stylistically correct.
- **Elegance**. The implementations must have good programming style and satisfy the critical linguistically trained reader.

The following properties, although good by themselves, have *not* been used as criteria for the library.

- **Syntactic coverage**. A resource grammar implementation must cover all constructs of the target language.
- **Semantical correctness**. The grammars may only generate meaningful expressions.
- **Translation equivalence**. The linearizations of an abstract syntax tree must be valid translations of each other.
- **Linguistic innovation**. The implementation must solve open linguistic problems.

In addition to the libraries once produced, it is interesting to evaluate the feasibility of the whole project. We want to reach a situation in which writing a resource grammar for a new language is a reasonable undertaking in terms of skill and effort. This involves the following factors:

- **Development effort**. Writing a new resource grammar implementation should not take too much time.
- **Developer skill**. Resource grammar writing should be accessible for persons with standard training in linguistics and programming.
- **Reuse of code and experience**. The growth of the library should make it increasingly easy to add new languages.

7.2 Results

Correctness

The main way of controlling syntactic correctness has been the creation of test suites, which are rerun whenever the library changes. Test suites can be organized as multilingual treebanks, with which one can assess a new grammar by comparing its linearizations with an old, trusted grammar. The trusted grammar can be a grammar of the same language as is being tested, but also of some other language. No trees are then needed to make comparisons, and the evaluation can be carried out without a knowledge of GF: it is enough to be sufficiently fluent in both involved languages. Thus one can verify, not only that a linearization is grammatically correct, but also that it corresponds to a trusted linearization. English is typical as such a trusted language, both because it is the oldest and the most well-proven language in the library and because it is widely known. For instance, when evaluating a French grammar, a sentence pair

```
she has always loved us
elle nous a aimés
```

would indicate that the French linearization, literally translated as *she has loved us*, mistakenly suppresses sentential adverbs. This would reveal an error in the grammar. But a bad translation might also be considered harmless, because we do not require translation equivalence. Such is the case with

```
she arrived yesterday
elle arrivait hier
```

where the English past tense is translated by the French imperfect. To get the correct form, the composed past *elle est arrivée hier*, a more fine-grained semantic model of the tenses would be needed (cf. Section 4.3).

Coverage

As for coverage, targeted applications have played a crucial role, as described in Section 3.2. Usually the applications have led to extensions in the library, without any explicit measurement of the sufficiency of the existing library. An exception is the GF-Sammie project (Perera and Ranta, 2007), which was formulated as an evaluation of the library in comparison to other grammar writing methods.

The in-car dialogue system Sammie was built within the European dialogue system project TALK. It targeted English and German, and was based on corpora collected for both languages in Wizard-of-Oz experiments (Becker et al., 2007). It used hand-written context-free grammars in the Nuance speech recognition format (Bevocal Inc., 2005). GF-Sammie was an experiment in which the grammars were rewritten in GF and ported to four new languages (Finnish, French, Spanish, Swedish) by using the resource grammar library. The experiment aimed to test both coverage and usability.

The starting point was the English Sammie grammar, hand-written in the Nuance format, with 2,400 context-free rules. This grammar was automatically translated to GF format, to generate a test corpus. But the automatic translation did not give a good basis for making the grammar multilingual. For this purpose, an abstract syntax was created on the basis of the semantic actions returned by the Nuance parser. For this abstract syntax, a functor-based concrete syntax was built with the purpose of covering all functionalities of Sammie, with at least some possible utterances for every semantic action. As another experiment, a full coverage of the German original grammar was attempted.

Two aspects of coverage were thus considered. The first aspect was semantic coverage: whether the library gives adequate expressions for all semantic actions required in the application. This aspect is classified as essential for the library, and it was indeed satisfied by the library without any additions (other than in the lexicon). The second aspect was syntactic coverage, which we have classified as less important for the library. Here the result was that about 70% of the coverage was reached. This was mostly due to colloquial, often telegraphic, expressions in the German corpus.

Usability

A central question in the GF-Sammie experiment (Perera and Ranta, 2007) was how easy is it to port a dialogue system grammar to a new language by using the library, compared with hand-writing it? The abstract syntax of the grammar included 10 categories and 50 rules (plus a vocabulary of artist, album, and song names). Covering this domain with a functor and instantiating it to one language took one working day. Each addition of a new language to the functor-based system took less than an hours' work, to reach a minimal coverage. The extension of the German grammar to the coverage of the original was is harder to evaluate, since the full user input coverage was not reached by the resource grammar.

One clear gain in GF vs. Nuance was due to the use of a feature-based, as opposed to a string-based, approach: GF-Sammie avoided, for instance, the expansion of the 16 forms of German noun phrases (combinations of number, gender, and case) into different rules, which means an order of magnitude less to write.

The overloaded API was not yet available at the time of the GF-Sammie experiment. It simplifies the use of the library considerably, since it decreases both the number of names to be memorized and the complexity of trees (Section 3.4). As a general library-related issue, an IDE (Integrated Development Environment) was put on the wish list, to match the customary skills and working habits of programmers.

Another evaluation of usability concerns the smart inflection paradigms, as used for lexicon construction in Finnish (Ranta, 2008). In this experiment, it was tested how many forms need to be given to the smart paradigms (Section 5.2) to get all the 26 inflection forms of Finnish nouns correct. It turned out that 82% of nouns were covered by the 1-argument paradigm and 96% by the 2-argument paradigm. In average, 1.42 word forms per noun were needed. Verbs, even though they have many more forms than nouns, were even more

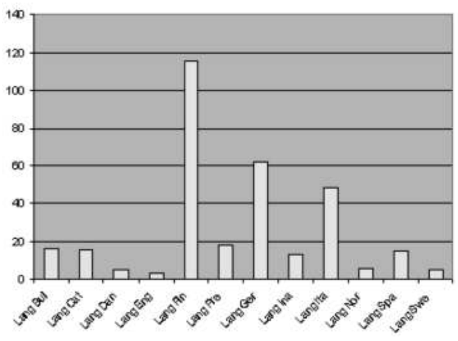

FIGURE 7 Parser speed in milliseconds per word. (Figure by Krasimir Angelov.)

predictable than nouns. This confirmed that the smart paradigms system gives a productive lexicon building tool to a lexicographer who knows how to inflect the target language in practice.

Efficiency

The efficiency of the library can be measured on several dimensions: the time and space needed for parsing and generation; the time and space needed for compiling the library and applications that use it; and the object file size created in compiling the library and its applications. One desired property of a library is that it should not come with a run-time penalty when compared with hand-written code. In GF, the partial evaluation performed in grammar compilation (see Section 2.8) usually takes care of this.

Careful optimization work on the resource grammars and parser generation have led to a situation in which all current resource grammars can be used for parsing by themselves, not only via application grammars. The sizes of the binary parser files for `Lang`*X* vary from less than 400kB (English and Scandinavian) to more than 3,000kB (Bulgarian, Finnish, French); see Angelov et al. 2009 for more details. The parsing speed varies from 4ms to 120ms per word, as shown in Figure 7 (on a MacBook Intel Core 2 Duo 2 GHz). What is even more interesting is that parsing is in practice linear, as shown in Figure 8, which shows the time (in milliseconds) needed for parsing a sentence of a given number of words. Finnish is the slowest language, followed by Italian and German, whereas English and Scandinavian are the fastest, reaching a speed of ten 20-word sentences per second.

Application grammars built using the resource behave much better, due to partial evaluation. The improvement of parsing times can easily be by an order of magnitude, even if the coverage of the grammar remains the same. One can even return the same trees as the original grammar, by defining the

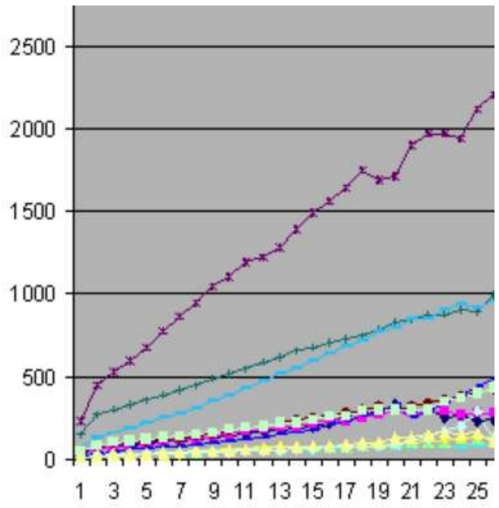

FIGURE 8 Parser speed in milliseconds as a function of sentence length. Finnish is the slowest, followed by Italian and German; English and Scandinavian are the fastest. (Figure by Krasimir Angelov.)

application's constructors in terms of the resource grammar's ones by means of GF's semantic definitions (Ranta, 2004).

Linearization in GF is fast regardless of grammar. The generation of 1,000 sentences and their linearization takes less than a second in any of the languages of the library. Thus a 1,000,000-sentence corpus used for training a speech recognition language model (Jonson, 2006) can be synthesized in less than 20 minutes.

In addition to parsing and linearization, the compilation of grammars is a relevant performance factor, since partial evaluation takes time and memory. The current resource grammars compile from source code to a run-time linearization grammar in 1 to 20 seconds, using 50M to 400MB of RAM, depending on language. If a precompiled parser is wanted, more time is needed—from 2 seconds to several minutes.

The performance results above are, of course, not only results about the library, but also about GF itself. The size and complexity of the resource grammars have been a driving force for performance improvements of GF.

7.3 Work and skills required

In the beginning of the project in 2001, writing a resource grammar was a demanding research task, which involved at the same time descriptive work on the target language and contributions to the design of the common abstract syntax. Thus the Russian grammar implementation was a part of the PhD thesis of Khegai (2006c), with an invested work between one and two person years, going through two major revisions of the abstract syntax. This time has been going down as the abstract syntax has become stable and as experience and code has been gained from more and more languages.

Thus the Bulgarian grammar by Angelov (2008) was finished in two months as a part of Angelov's PhD project, and the Romanian by Ramona Enache in 2009 was a three-month MSc-level project. The current rule of thumb is that adding a language to a library is a task between the Master's and Doctoral level and takes two to six months. It needs solid theoretical knowledge of the target language, but this knowledge can usually be acquired from literature, and doesn't hence require a native speaker's competence. In fact, it is in applications targeted for end users that native-speaker expertise is needed, often in combination with domain expertise, so that the resource is used in natural and idiomatic ways for expressing the domain concepts. The resource itself can only guarantee grammatical correctness.

Code size

Figure 9 is a summary of the numbers of lines of code in version 1.6 implementation. The figures marked by an asterisk (*) include low-level code generated from a source written in Haskell using the Functional Morphology library (Forsberg and Ranta, 2004). The table shows that one language implementation requires 5,000 lines of code in average. If we eliminate the generated code in Italian and Spanish, and use the French code size as estimate for the morphology for these languages, we get the average of around 3,700 lines per language.

7.4 Sharing code between languages

In Figure 9, there are rows for Romance and Scandinavian, which are not languages but language families. Within these families, common modules are used for implementing individual languages. Thus Catalan, French, Italian, and Spanish result from common Romance code together with language-specific code. Danish, Norwegian, and Swedish result from Scandinavian in the same way. The common code is used for syntactic combinations: no attempt was made to share code in morphology and lexicon. So, if we consider syntax alone, we see that 74–76% of the code for each Romance language is shared. The corresponding figure for Scandinavian is even higher, 82–83%.

The technique used for sharing code is functors, as explained in Ranta (2007b) and also briefly in Section 6.1 above. The idea is that the common Romance or Scandinavian concrete syntax is written relative to a set of parameters, which are then instantiated separately for each languages. In Romance, for instance, there are around 30 such parameters. Most of them are

language	**syntax**	**morpho**	**lex**	**total**	**months**	**start**
common	413	-	-	413	2	2001
abstract	729	-	468	1197	24	2001
Bulgarian	1200	2329	502	4031	3	2008
English	1025	772	506	2303	6	2001
Finnish	1471	1490	703	3664	6	2003
German	1337	604	492	2433	6	2002
Polish	2466	5370	508	8344	6	2008
Russian	1492	3668	534	5694	18	2002
Romanian	1713	3513	666	5892	3	2009
Romance	1346	-	-	1346	10	2003
Catalan	521	*9000	518	*10039	4	2006
French	468	1789	514	2771	6	2002
Italian	423	*7423	500	*8346	3	2003
Spanish	417	*6549	516	*7482	3	2004
Scandinavian	1293	-	-	1293	4	2005
Danish	262	683	486	1431	2	2005
Norwegian	281	676	488	1445	2	2005
Swedish	280	717	491	1488	4	2001
total	16724	*43636	7892	*70199	112	2001

FIGURE 9 Programming effort in lines of code (* = includes generated code).

lexical units that are baked in into syntax rules, such as the definite and indefinite articles, and the verb used as the passive auxiliary (*être* in French, *venire* in Italian, *ser* in Spanish). Some are more fundamental, for instance, the operation

```
clitInf : Bool -> Str -> Str -> Str
```

which decides how infinitives and clitics are placed relative to each other (French *la voir*, Italian *vederla*, "to see her"). Scandinavian has 23 parameters.

Using functors instead of separate implementations saves a lot of work in the implementation of new languages and, in particular, in the maintenance of the code. At the same time, it is an interesting linguistic experiment, which illustrates the abstraction capabilities of GF. On the other hand, precisely because functors work on a higher level of abstraction, they are more difficult to get started with than separate implementations. They won't necessarily give any practical advantage for languages that are more remotely related than the ones in the Romance family; therefore, we have not attempted to build functor implementations for Germanic and Slavic languages. Also the Romanian grammar was built independently of the Romance functor.

The common abstract syntax is of course also a way to share code between languages, even unrelated ones. If we count it as a part of the syntax implementation for a language, then the amount of shared syntax code in the library is 32% in the worst case (Polish) and 89% in the best case (Danish).

8 Related work

8.1 Multilingual grammars

The closest inspiration of the GF Resource Grammar Library is CLE, as presented in Rayner et al. (2000). They report on covering four languages: Danish, English, French, and Swedish. The described coverage was used as a benchmark for the GF library. Some of the more language-specific constructs of CLE are not yet available in the GF library, e.g., French complex inversion questions (*Jean aime-t-il Marie?*). The heritage of CLE is used in the Regulus resource grammar library (Rayner et al., 2006), which provides grammars for seven languages: Catalan, Finnish, English, French, Greek, Japanese, and Spanish. Their coverage varies, but it is smaller than CLE. The focus of Regulus is very much on spoken language translation. CLE has a proprietary license, which is difficult to obtain, whereas Regulus is open-source.

Despite the formal difference between GF and CLE/Regulus's unification grammars, there are three major similarities: the specialization of large grammars to domain-specific applications, the reuse of code between languages, and the compilation to speech recognition language models. Specialization in CLE/Regulus is performed using **explanation-based learning**, which implements an idea that could be characterized as intersecting the grammar with a training corpus. The partial evaluation used in GF has similar effects in practice, although the technique itself is different. Code sharing among e.g. the Romance languages has similar effects in both Regulus and GF; in Regulus it is implemented by the low-end techniques of macros and file inclusions, whereas GF uses separately compiled functions and functors, as explained in Section 6.1. These latter techniques are, as it were, internalizations of the low-end techniques as first-class language constructs, which is a general trend in the development of programming languages.

The LinGO Matrix/DelphIn grammar set (Bender and Flickinger, 2005) is based on large-scale grammars from earlier projects for English, German, Japanese, and Spanish, and many more languages are under construction. The languages are related to each other by a common representation in Minimal Recursion Semantics (Copestake et al., 2001), using transfer rules. There is, moreover, a "Matrix questionnaire", which is a set of questions whose answers generate a basic grammar implementation for a language. This questionnaire can be seen as a semi-formal version of an abstract syntax in GF. LinGO also has a grammar-driven treebank (Oepen et al., 2004) supporting the development. The grammar formalism is HPSG (Pollard and Sag, 1994), as implemented in the LKB framework (Copestake, 2002). In most applications, HPSG grammars are used as stand-alone wide-coverage parsers communicating with the rest of the system via pipes, rather than via grammar specialization. LingGO Matrix/DelphIn grammars are available under open source licenses.

In the Pargram project (Butt et al., 2002), the aim is to "produce wide coverage grammars for English, French, German, Norwegian, Japanese, and Urdu which are written collaboratively within the linguistic framework of LFG and with a commonly-agreed-upon set of grammatical features" (according to the Pargram project description at `www.ist-world.org`). Several more languages are treated in ongoing projects: Arabic, Chinese, Georgian, Hungarian, Tigrinya, Turkish, Vietnamese, and Welsh. The English grammar is used in a commercial prototype within the Powerset company; since the goal is to parse documents on the web, wide coverage is essential.

8.2 Grammar formalisms

As a grammar formalism, GF belongs to the family of **categorial grammars**, in the broad sense that syntactic categories are encoded as types in typed lambda calculus and syntactic construction is function application. The most well-known categorial grammars are those based on Lambek calculus (Lambek, 1958), which treats the linear order of constituents with the same type system as constituency, and is therefore not usable for multilingual grammars.

The fusion of linear order and constituency in Lambek calculus (as well as in Chomsky's transformational grammar) was criticized by Curry (1961), whose own suggestion was to separate the **tectogrammatical** rules, which define constituency, from the **phenogrammatical** ones, which define linear order. This distinction is in computer science known under the terms abstract and concrete syntax (McCarthy, 1962), which terms have been adopted in GF. GF can thus be seen as an implementation of Curry's grammar architecture, adding the parameter system (see Section 2.3) to phenogrammatical descriptions. Also Montague grammar (Montague, 1974) can been seen as an instance of Curry's architecture (Dowty, 1979). The separation between ID and LP rules (immediate dominance vs. linear precedence) in GPSG (Gazdar et al., 1985) is another approach with similar effects. The grammar composition idea, where trees on higher abstraction levels are mapped to trees on lower levels, also appears in the Meaning-Text theory of Mel'cuk (1997).

A new wave of categorial grammars in Curry style started at the same time as GF, in the late 1990's. ACG (Abstract Categorial Grammars, de Groote 2001), is more general than GF in the sense that linearization types can be function types just like the abstract syntax types, but less general in the sense that functions have to be linear (that is, use every argument exactly once). HOG (Higher Order Grammars, Pollard 2004) aims to solve some problems of HPSG by introducing a Curry-style architecture. In the published examples of HOG grammars, language-specific features are integrated in abstract syntax types, which is not compatible with GF-style multilinguality. Pollard's later formalism, convergent grammars (Pollard, 2009), takes a step back in the direction of HPSG. Lambda grammars (Muskens, 2001) is yet another formalism in this new wave, focusing on semantics. These formalisms have not yet received substantial computer implementations or produced sizable grammars. In the categorial grammar tradition, it is CCG (Steedman, 1988, 2000) that has produced the most substantial grammars, which are moreover combined with statistic parsing and generation (Bos et al., 2004, Espinosa et al., 2008). The variable-free categorial treatment of binding in the GF Resource Grammar Library was originally inspired by CCG; see Section 4.5.

9 Conclusion

The experience from the GF Resource Grammar Library can be summarized as follows:

- It formalizes the inflectional morphology and a comprehensive fragment of the syntax of 14 languages.
- It uses a common abstract syntax that is feasible to implement for new languages.
- It provides a platform on which non-linguist application programmers can easily write multilingual application programs.

Applications using the library have been built on various domains of multilingual generation, technical text translation, and spoken dialogue systems. The library can also be used as a general resource for computational linguistics, and it can be converted to several standard formats with GF grammar compiler tools, e.g., to language models for speech recognition and to JavaScript code for web applications.

The library is being extended to several new languages, around 20 at the moment. The library makes no claim to be a universal grammar, which would fit all languages: we keep the question of applicability to different languages open and challenge grammarians to try it out and see how far they can get.

The extension of the grammar of one language to a full coverage of that language yet largely unexplored. It requires the addition of lexical entries and language-specific rules. It can also be useful to write the parser as an application grammar that flattens the tree structures and reduces ambiguity.

Acknowledgments

The author is grateful to Robin Cooper and Elisabet Engdahl, as well as to two anonymous referees, for comments on earlier versions of this paper, and to Annie Zaenen for editorial support. The resource grammar project itself is a collaborative effort, where substantial contributions have been made by at least Inger Andersson, Krasimir Angelov, Jean-Philippe Bernardy, Ludmilla Bogavac, Björn Bringert, Ali El Dada, Ramona Enache, Markus Forsberg, Harald Hammarström, Muhammad Humayoun, Janna Khegai, Ilona Komorowski, Anni Laine, Wanjiku Ng'ang'a, Andreas Priesnitz, Jordi Saludes, Adam Slaski, Therése Söderberg, and Shafqat Virk. Moreover, the 30 participants of the Resource Grammar Summer School 2009 will certainly have produced important contributions by the time this article is being read. The library has enjoyed financial support from several grants, including Interactive Language Technology (Vinnova P20018-2A), Records, types and computational dialogue semantics (VR 2002-4879), Library-based Grammar Engineering (VR 2005-4211), TALK (IST-506802), WebALT (EDC-22253), and JEM (ECP-2005-EDU-038208).

References

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

Angelov, K. 2008. Type-Theoretical Bulgarian Grammar. In B. Nordström and A. Ranta, eds., *Advances in Natural Language Processing (GoTAL 2008)*, vol. 5221 of *LNCS/LNAI*, pages 52–64. URL http://www.springerlink.com/content/978-3-540-85286-5/.

Angelov, K., B. Bringert, and A. Ranta. 2009. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information* To appear.

Becker, T., N. Blaylock, C. Gerstenberger, A. Korthauer, N. Perera, M. Pitz, P. Poller, J. Schehl, F. Steffens, R. Stegmann, and J. Steigner. 2007. In-Car Showcase Based on TALK. TALK. Talk and Look: Tools for Ambient Linguistic Knowledge. IST-507802. Deliverable 5.3. URL http://www.talk-project.org/.

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

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. URL http://faculty.washington.edu/ebender/papers/modules05.pdf.

Bevocal Inc. 2005. Nuance GSL Grammar Format. URL http://cafe.bevocal.com/docs/grammar/gsl.html.

Borin, L., M. Forsberg, and L. Lönngren. 2009. SALDO 1.0 (Svenskt associationslexikon version 2). Språkbanken, Göteborgs universitet. URL http://spraakbanken.gu.se/personal/markus/publications/saldo_1.0.pdf.

Bos, J., S. Clark, M. Steedman, J. Curran, and J. Hockenmaier. 2004. Wide-Coverage Semantic Representations from a CCG Parser. In *Coling 2004*. URL http://www.iccs.inf.ed.ac.uk/~stevec/papers/bos_etal.pdf.

Bringert, Björn. 2007. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*, pages 1–8. Association for Computational Linguistics. URL http://www.cs.chalmers.se/~bringert/publ/gf-srg/gf-srg.pdf.

Bringert, B. 2009. Delimited Continuations, Applicative Functors and Natural Language Semantics. Technical Report, Chalmers University of Technology. URL http://digitalgrammars.com/gf/demos/mosg/.

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

Butt, M., H. Dyvik, T. Holloway King, H. Masuichi, and C. Rohrer. 2002. The Parallel Grammar Project. In *COLING 2002, Workshop on Grammar Engineering and Evaluation*, pages 1–7. URL http://www2.parc.com/isl/groups/nltt/pargram/buttetal-coling02.pdf.

Caprotti, O. 2006. WebALT! Deliver Mathematics Everywhere. In *Proceedings of SITE 2006. Orlando March 20-24*. URL http://webalt.math.helsinki.fi/content/e16/e301/e512/PosterDemoWebALT_eng.pdf.

Chomsky, N. 1957. *Syntactic Structures*. The Hague: Mouton.

Claessen, K. 2005. Equinox, A New Theorem Prover for Full First-Order Logic with Equality. In *Dagstuhl Seminar 05431 on Deduction and Applications*. URL http://www.cs.chalmers.se/~koen/pubs/entry-dagstuhl05-equinox.html.

Cooper, R. 2008. The abstract-concrete syntax distinction and unification in multilingual grammar. URL http://publications.uu.se/abstract.xsql?dbid=8933.

Copestake, A. 2002. *Implementing Typed Feature Structure Grammars*. CSLI Publications.

Copestake, A., D. Flickinger, C. Pollard, and I. Sag. 2001. Minimal recursive semantics: An introduction. *Language and Computation* 1:1–47.

Curry, H. B. 1961. Some logical aspects of grammatical structure. In R. Jakobson, ed., *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Dada, A. El and A. Ranta. 2007. Implementing an Open Source Arabic Resource Grammar in GF. In M. Mughazy, ed., *Perspectives on Arabic Linguistics XX*, pages 209–232. John Benjamin's.

de Groote, Ph. 2001. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Toulouse, France*, pages 148–155. URL http://www.loria.fr/~degroote/papers/acl01.pdf.

Diderichsen, Paul. 1962. *Elementaer dansk grammatik*. Kobenhavn: Gyldendal.

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

Dymetman, M. V. Lux, and A. Ranta. 2000. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249. URL http://www.cs.chalmers.se/~aarne/articles/coling2000.ps.gz.

Espinosa, D., M. White, and D. Mehay. 2008. Hypertagging: Supertagging for Surface Realization with CCG. In *ACL 2008*. Columbus, Ohio.

Forsberg, M. and A. Ranta. 2004. Functional Morphology. In *ICFP 2004, Showbird, Utah*, pages 213–223. URL http://www.cs.chalmers.se/~markus/FM/FM_ICFP2004.pdf.

Gazdar, G., E. Klein, G. Pullum, and I. Sag. 1985. *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

Grevisse, Maurice. 1993. *Le bon usage, 13me edition*. Paris: Duculot.

Hammarström, H. and A. Ranta. 2004. Cardinal Numerals Revisited in GF. In *Workshop on Numerals in the World's Languages, Dept. of Linguistics, Max Planck Institute for Evolutionary Anthropology, Leipzig*. URL http://www.cs.chalmers.se/%7Eharald2/numabstract.pdf.

Huet, Gerard. 2005. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. *The Journal of Functional Programming* 15(4):573–614.

Humayoun, M., H. Hammarström, and A. Ranta. 2007. Urdu Morphology, Orthography and Lexicon Extraction. In *CAASL-2: The Second Workshop on Computational Approaches to Arabic Script-based Languages, LSA 2007 Linguistic Institute, Stanford University, July 21-22, 2007*.

Jackendoff, R. 1977. *X-Bar Syntax: A Study of Phrase Structure*. MIT Press.

Jonson, Rebecca. 2006. Generating statistical language models from interpretation grammars in dialogue system. In *Proceedings of EACL'06, Trento, Italy*.

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

Khegai, J. 2006a. GF Parallel Resource Grammars and Russian. In *Coling/ACL 2006*, pages 475–482.

Khegai, J. 2006b. Grammatical Framework (GF) for MT in sublanguage domains. In *Proceedings of EAMT-2006 (11th Annual conference of the European Association for Machine Translation, Oslo, Norway*, pages 95–104.

Khegai, J. 2006c. *Language Engineering in Grammatical Framework (GF)*. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology and University of Gothenburg. URL http://www.cs.chalmers.se/~janna/Janna_Khegai_phd.pdf.

Khegai, J. B. Nordström, and A. Ranta. 2003. Multilingual Syntax Editing in GF. In A. Gelbukh, ed., *Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico City, February 2003*, vol. 2588 of *LNCS*, pages 453–464. Springer-Verlag. URL http://www.cs.chalmers.se/~aarne/articles/mexico.ps.gz.

Kotimaisten Kielten Tutkimuskeskus. 2006. KOTUS Wordlist. URL http://kaino.kotus.fi/sanat/nykysuomi.

Lambek, J. 1958. The mathematics of sentence structure. *American Mathematical Monthly* 65:154–170.

Ljunglöf, P. 2004. *The Expressivity and Complexity of Grammatical Framework*. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology and University of Gothenburg. URL http://www.cs.chalmers.se/~peb/pubs/p04-PhD-thesis.pdf.

Ljunglöf, P., G. Amores, R. Cooper, D. Hjelm, O. Lemon, P. Manchón, G. Pérez, and A. Ranta. 2006. Multimodal Grammar Library. TALK. Talk and Look: Tools for Ambient Linguistic Knowledge. IST-507802. Deliverable 1.2b. URL http://www.talk-project.org/fileadmin/talk/publications_public/deliverables_public/TK_D1-2-2.pdf.

McCarthy, J.i962. Towards a mathematical science of computation. In *Proceedings of the Information Processing Cong. 62*, pages 21–28. Munich, West Germany: North-Holland.

Mel'cuk, I. 1997. Vers une linguistique Sens-Texte. Leçon inaugurale. URL http://olst.ling.umontreal.ca/pdf/MelcukColldeFr.pdf.

Meza Moreno, M. S. and B. Bringert. 2008. Interactive Multilingual Web Applications with Grammarical Framework. In B. Nordström and A. Ranta, eds., *Advances in Natural Language Processing (GoTAL 2008)*, vol. 5221 of *LNCS/LNAI*, pages 336–347. URL http://www.springerlink.com/content/978-3-540-85286-5/.

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

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

Muskens, R. 2001. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, eds., *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155. Amsterdam. URL http://let.uvt.nl/general/people/rmuskens/pubs/amscoll.pdf.

Oepen, S., D. Flickinger, K. Toutanova, and C. D. Manning. 2004. LinGO Redwoods, A Rich and Dynamic Treebank for HPSG. *Research on Language and Computation* 2:575–596. URL http://www.springerlink.com/content/t851781443373812/.

Perera, N. and A. Ranta. 2007. Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*. URL http://www.cs.chalmers.se/~aarne/articles/perera-ranta.pdf.

Peyton Jones, S. 2003. Haskell 98 language and libraries: the Revised Report. URL http://www.haskell.org/haskellwiki/Language_and_library_specification.

Pollard, C. 2004. Higher-Order Categorical Grammar. In M. Moortgat, ed., *Proceedings of the Conference on Categorial Grammars (CG2004), Montpellier, France*, pages 340–361. URL http://www.ling.ohio-state.edu/~hana/hog/pollard2004-CG.pdf.

Pollard, C. 2009. Convergent Grammar web page. URL http://www.ling.ohio-state.edu/~scott/cvg/.

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

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

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

Ranta, A. 2004. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming* 14(2):145–189. URL http://www.cs.chalmers.se/~aarne/articles/gf-jfp.ps.gz.

Ranta, A. 2007a. Features in Abstract and Concrete Syntax. In *NODALIDA Workshop on Typed Feature Structure Grammars, Tartu, 24 May 2007*. URL http://www.cs.chalmers.se/~aarne/articles/ranta-tfsg2007.pdf.

Ranta, A. 2007b. Modular Grammar Engineering in GF. *Research on Language and Computation* 5:133–158. URL http://www.cs.chalmers.se/~aarne/articles/multieng3.pdf.

Ranta, A. 2008. How predictable is Finnish morphology? an experiment on lexicon construction. In J. Nivre and M. Dahllöf and B. Megyesi, ed., *Resourceful Language Technology: Festschrift in Honor of Anna Sågvall Hein*, pages 130–148. University of Uppsala. URL http://publications.uu.se/abstract.xsql?dbid=8933.

Ranta, A. 2009. Grammars as Software Libraries. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, eds., *From Semantics to Computer Science*. Cambridge University Press. URL `http://www.cs.chalmers.se/~aarne/articles/libraries-kahn.pdf`.

Ranta, A. and K. Angelov. 2009. Implementing Controlled Languages in GF. In N. Fuchs, ed., *CNL 2009, Controlled Natural Languages, Marettimo, Sicily*, vol. 448 of *CEUR Proceedings*. URL `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-448/`.

Rayner, M., D. Carter, P. Bouillon, V. Digalakis, and M. Wirén. 2000. *The Spoken Language Translator*. Cambridge: Cambridge University Press.

Rayner, M., B. A. Hockey, and P. Bouillon. 2006. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications.

Reichenbach, Hans. 1948. *Elements of Symbolic Logic*. New York: The MacMillan Company.

Ross, J. 1967. *Constraints on Variables in Syntax*. Ph.D. thesis, Massachusetts Institute of Technology.

Steedman, M. 1988. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler, eds., *Categorial Grammars and Natural Language Structures*, pages 417–442. Dordrecht: D. Reidel.

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

Swadesh, Morris. 1955. Towards Greater Accuracy in Lexicostatistic Dating. *International Journal of American Linguistics* 21:121–137.

Union Mundial pro Interlingua. 2001. Interlingua Homepage. URL `http://www.interlingua.com/`.

Young, Steve, Gunnar Evermann, Dan Kershaw, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, Valtcho Valtchev, and Phil Woodland. 2002. The HTK Book, Version 3.2. Cambridge University Engineering Dept, December 2002, URL `http://www.htk.eng.cam.ac.uk.`

Appendix

The core syntax categories and functions

This Appendix gives a complete summary of the language-independent core syntax. Section 1 shows the categories, and the remaining sections show the construction functions. The subsection numbering corresponds to the numbering in Section 4, where the syntax is discussed.

The syntax rules are given in a BNF format, where each line defines an abstract syntax constructor. Thus, for instance,

`Phr`	::=	`PConj Utt Voc`	*but walk, my friend*

means that there is a constructor of type `PConj -> Utt -> Voc -> Phr`. Identifiers in angle brackets, such as `<fullstopPunct>`, are not categories but atomic constructors. After each rule, an English example is given. Most examples can be automatically translated to all library languages, since they use the vocabulary included in the common test lexicon.

The entire library with on-line documentation is available as open-source software under the GNU LGPL license in `digitalgrammars.com/gf`.

1 Categories

1.1 Phrasal and closed lexical categories

`AdvSlash`	adverb missing complement (= `Prep`)	*with*
`Ant`	anteriority	simultaneous, anterior
`CAdv`	comparative adverb	*more*
`CN`	common noun (without determiner)	*red house*
`Card`	cardinal number	*seven*
`Cl`	declarative clause, with all tenses	*she looks at this*
`Comp`	complement of copula, such as AP	*very warm*
`Conj`	conjunction	*and*
`Det`	determiner phrase	*those seven*
`Digits`	cardinal or ordinal in digits	*1,000/1,000th*
`IAdv`	interrogative adverb	*why*
`IComp`	interrogative complement of copula	*where*
`IDet`	interrogative determiner	*how many*
`IP`	interrogative pronoun	*who*

`Imp`	imperative	*look at this*
`NP`	noun phrase (subject or object)	*the red house*
`Num`	number determining element	*seven*
`Numeral`	cardinal or ordinal in words	*five/fifth*
`Ord`	ordinal number (used in Det)	*seventh*
`PConj`	phrase-beginning conjunction	*therefore*
`Phr`	phrase in a text	*but be quiet please*
`Pol`	polarity	positive, negative
`Predet`	predeterminer (prefixed Quant)	*all*
`Prep`	preposition, or just case	*in*
`Pron`	personal pronoun	*she*
`QCl`	question clause, with all tenses	*why does she walk*
`QS`	question	*where did she live*
`Quant`	quantifier ('nucleus' of Det)	*this/these*
`RCl`	relative clause, with all tenses	*in which she lives*
`RP`	relative pronoun	*in which*
`RS`	relative sentence	*in which she lived*
`S`	declarative sentence	*she lived here*
`SC`	embedded sentence or question	*that it rains*
`Subj`	subjunction	*if*
`Temp`	temporal and aspectual features	past anterior
`Tense`	tense	present, past, future
`Text`	text consisting of several phrases	*He is here. Why?*
`Utt`	utterance: sentence, question...	*be quiet*
`VP`	verb phrase	*is very warm*
`VPSlash`	verb phrase missing complement	*give to John*
`Voc`	vocative	*my darling*
`[`*C*`]`	list of category *C*	*X, Y, Z*

1.2 Open lexical categories

`AdA`	adjective-modifying adverb	*very*
`AdN`	numeral-modifying adverb	*more than*
`AdV`	adverb directly attached to verb	*always*
`Adv`	verb-phrase-modifying adverb	*in the house*
`A`	one-place adjective	*warm*
`A2`	two-place adjective	*divisible*
`AP`	adjectival phrase	*very warm*
`N`	common noun	*house*
`N2`	relational noun	*son*
`N3`	three-place relational noun	*connection*
`PN`	proper name	*Paris*
`V`	one-place verb	*sleep*
`V2`	verb with an NP complement	*love*

`VA`	verb with an AP complement	*look*
`VQ`	verb with a QS complement	*wonder*
`VS`	verb with an S complement	*claim*
`VV`	verb with a VP complement	*want*
`V2A`	verb with NP and AP complement	*paint*
`V2Q`	verb with NP and QS complement	*ask*
`V2S`	verb with NP and S complement	*tell*
`V2V`	verb with NP and VP complement	*cause*
`V3`	verb with two NP complements	*show*

2 Suprasentential level: texts and utterances

`Text`	::=	ϵ	(empty text)
	|	`Phr Punct Text`	*Who walks? John.*
`Punct`	::=	`<fullstopPunct>`	.
	|	`<questionPunct>`	*?*
	|	`<exclamationPunct>`	*!*
`Phr`	::=	`PConj Utt Voc`	*but walk, my friend*
`PConj`	::=	`PConj`	*and*
`Voc`	::=	`NP`	*my friend*
`Utt`	::=	`S`	*John walks*
	::=	`QS`	*who walks*
	|	`Imp`	*don't walk*
	|	`NP`	*this man*
	|	`Adv`	*here*
	|	`VP`	*to sleep*
	|	`IP`	*who*
	|	`IAdv`	*why*

3 Sentential level: polarity, tense, and mood

`S`	::=	`Tense Pol Cl`	*John wouldn't have walked*
`QS`	::=	`Tense Pol QCl`	*wouldn't John have walked*
`RS`	::=	`Tense Pol RCl`	*that wouldn't have walked*
`Pol`	::=	`<positivePol>`	(as in *he walks*)
	|	`<negativePol>`	(as in *he doesnät walk*)
`Tense`	::=	`Temp Ant`	(the two components of a tense)
`Temp`	::=	`<presentTemp>`	(as in *he walks*)
	|	`<pastTemp>`	(as in *he walked*)
	|	`<futureTemp>`	(as in *he will walk*)
	|	`<conditionalTemp>`	(as in *he would walk*)
`Ant`	::=	`<simultaneousAnt>`	(as in *he walks*)
	|	`<anteriorAnt>`	(as in *he has walked*)
`Imp`	::=	`ImpForm Pol VP`	*don't forget yourselves*
`ImpForm`	::=	`<famImpForm>`	(as in *help yourself*)
	|	`<polImpForm>`	(as in *help yourself, sir*)
	|	`<pluralImpForm>`	(as in *help yourselves*)

4 Clause level: predication and complementation

Cl	::=	NP VP	*John walks*
	|	SC VP	*that he walks is good*
VP	::=	V	*sleep*
	|	VV VP	*want to run*
	|	VS S	*know that she runs*
	|	VQ QS	*wonder if she runs*
	|	VA AP	*look red*
	|	VPSlash NP VP	*use it*
	|	VPSlash	*love himself*
	|	VPSlash	*be loved*
	|	Comp	*be small*
	|	VP Adv	*sleep here*
	|	VP AdV	*always sleep*
Comp	::=	AP	*(be) small*
	|	NP	*(be) a man*
	|	Adv	*(be) here*
SC	::=	S	*that she goes*
	|	QS	*who goes*
	|	VP	*to go*

5 Questions and relatives

5.1 Clause formation and extraction

SSlash	::=	Tense Pol ClSlash	*he used (it)*
QCl	::=	Cl	*does John walk*
	|	IP VP	*who walks*
	|	IP ClSlash	*whom does John love*
	|	IAdv Cl	*why does John walk*
	|	IComp NP	*where is John*
RCl	::=	RP VP	*who walks*
	|	RP ClSlash	*whom John loves*
ClSlash	::=	NP VPSlash	*he uses (it)*
	|	Cl AdvSlash	*he walks with (her)*
	|	ClSlash Adv	*he uses (it) today*
	|	NP VS SSlash	*she says that he uses (it)*
VPSlash	::=	V2	*use (it)*
	|	V3 NP	*send it (to her)*
	|	V3 NP	*send (it) to her*
	|	V2V VP	*force (her) to run*
	|	V2S S	*tell (us) that she runs*
	|	V2Q QS	*ask (us) if she runs*
	|	V2A AP	*paint (it) red*
	|	VV VPSlash	*want to use (it)*
	|	V2V NP VPSlash	*force me to use (it)*

5.2 Interrogative and relative pronouns

`IP`	::=	`IDet CN`	*which five songs*
	|	`IDet`	*which five*
	|	`IP Adv`	*who in Paris*
`IDet`	::=	`IQuant Num IDet`	*which five*
`IAdv`	::=	`Prep IP`	*with whom*
`IComp`	::=	`IAdv`	*where (is it)*
	|	`IP`	*who (is it)*
`RP`	::=	`<idRP>`	*which*
	|	`Prep NP RP`	*only one of which*

6 Noun phrases and determiners

`NP`	::=	`Det CN`	*the man*
	|	`PN`	*John*
	|	`Pron`	*he*
	|	`Predet NP`	*only the man*
	|	`NP V2`	*the man seen*
	|	`NP Adv`	*Paris today*
	|	`NP RS`	*Paris, which is here*
	|	`Det`	*these five*
	|	`CN`	*beer*
`Det`	::=	`Quant Num Ord`	*these five best*
	|	`Quant Num Det`	*these five*
`Quant`	::=	`Pron Quant`	*my*
	|	`<defArt>`	*the*
	|	`<indefArt>`	*a*
`Num`	::=	`Card`	*fifty-six*
	|	`<sgNum>`	(as in *this*)
	|	`<plNum>`	(as in *these*)
`Card`	::=	`Numeral`	*fifty-six*
	|	`Digits`	*56*
`Ord`	::=	`Numeral`	*fifty-sixth*
	|	`Digits`	*56th*
	|	`A`	*best*

6.1 The numeral system

`Digits`	::=	`Dig`	*5*
	|	`Dig Digits`	*528*
`Dig`	::=	`<0_Dig>...<9_Dig>`	*0,1,2,3,4,5,6,7,8,9*
`Numeral`	::=	`Sub1000000`	coerce *1..999999*
`Sub1000000`	::=	`Sub1000`	*m * 1000*
	|	`Sub1000 Sub1000`	*m * 1000 + n*
	|	`Sub1000`	coerce *1..999*

`Sub1000`	::=	`Sub10`	*m * 100*
	|	`Sub10 Sub100`	*m * 100 + n*
	|	`Sub100`	coerce *1..99*
`Sub100`	::=	`Sub10`	*m * 10*
	|	`Sub10 Sub10`	*m * 10 + n*
	|	`Sub10`	coerce *1..9*
`Sub10`	::=	`<one_Sub10>`,...,`<nine_Sub10>`	*one,...,nine*

7 Common nouns, adjectives, and adverbs

`CN`	::=	`N`	*man*
	|	`CN AP`	*old man*
	|	`CN RS`	*man who walks*
	|	`N2 NP`	*mother (of x)*
	|	`N3 NP NP`	*distance (from x) (to y)*
	|	`CN SC`	*question where she sleeps*
`AP`	::=	`A`	*warm*
	|	`A NP`	*warmer than Paris*
	|	`A2 NP`	*married to her*
	|	`A2`	*married to himself*
	|	`A2`	*married*
	|	`AP SC`	*good that she sleeps*
	|	`AdA AP`	*very good*
`AdN`	::=	`CAdv`	*more (than five)*
`AdA`	::=	`A`	*extremely*
`Adv`	::=	`A`	*quickly*
	|	`Prep NP`	*in the house*
	|	`CAdv A NP`	*less quickly than John*
	|	`CAdv A S`	*less quickly than I ran*
	|	`AdA Adv`	*very quickly*
	|	`Subj S`	*when he arrives*

8 Coordination

For *C* = `Adv`, `AP`, `NP`, `RS`, `S`:

C	::=	`Conj [`*C*`]`	*X, Y and Z*
`[`*C*`]`	::=	*C C*	*Y, Z*
	|	*C* `[`*C*`]`	*X, Y, Z*

9 Idiomatic constructions

`QCl`	::=	`IP`	*which houses are there*
`Cl`	::=	`NP`	*there is a house*
	|	`VP`	*it is hot* (impersonal)
	|	`VP`	*one sleeps* (generic)
	|	`NP RS`	*it is I who did it*
	|	`Adv S`	*it is here she slept*

9.1 Structural words

`AdA`	::=	*almost, quite, so, too, very*
`AdV`	::=	*always*
`Adv`	::=	*everywhere, here, from-here, to-here, somewhere, there*
	|	*from-there, to-there*
`CAdv`	::=	*less, more*
`Conj`	::=	*and, both-and, either-or, or*
`Det`	::=	*every, few, many, much, some(Pl), some(Sg)*
`IAdv`	::=	*how, when, where, why*
`IDet`	::=	*how-many*
`IP`	::=	*what(Pl), what(Sg), who(Pl), who(Sg)*
`IQuant`	::=	*which*
`NP`	::=	*everybody, everything, somebody, something*
`PConj`	::=	*but, otherwise, therefore*
`Predet`	::=	*all, most, only*
`Prep`	::=	*above, after, before, behind, between, by(agent), by-means*
	|	*during, for, from, in-front, in, on, of(partitive)*
	|	*of(possessive), through, to, under, with, without*
`Pron`	::=	*I, you(Sg), you(Polite), he, she, it, we, you(Pl), they*
`Quant`	::=	*that, this*
`Subj`	::=	*although, because, if, when*
`Utt`	::=	*no, yes*
`VV`	::=	*can, can(know-how), must, want*