Modular Grammar Engineering in GF

Aarne Ranta
Department of Computing Science
Chalmers University of Technology and Göteborg University
aarne@cs.chalmers.se

Abstract. *The Grammatical Framework GF is a grammar formalism designed for multilingual grammars. A multilingual grammar has a shared representation, called abstract syntax, and a set of concrete syntaxes that map the abstract syntax to different languages. A GF grammar consists of modules, which can share code through inheritance, but which can also hide information to achieve division of labour between grammarians working on different modules. The goal is to make it possible for linguistically untrained programmers to write linguistically correct application grammars encoding the semantics of special domains. Such programmers can rely on resource grammars, written by linguists, which play the rôle of standard libraries. Application grammarians use resource grammars through abstract interfaces, and the type system of GF guarantees that grammaticality is preserved. The ongoing GF resource grammar project provides resource grammars for ten languages. In addition to their use as libraries, resource grammars serve as an experiment showing how much grammar code can be shared between different languages.*

1. Introduction

1.1. Multilingual grammars

By a *multilingual grammar*, we mean a grammar that describes multiple languages sharing a common representation. The term "multilingual grammar engineering" is ambiguous: it can mean engineering of multilingual grammars, but also just multilingual engineering of grammars, which are formally unrelated even though developed in parallel.

Multilingual grammars are useful in applications like translation between languages, and *localization*, which means porting natural-language applications to new languages. The shared representation can be used to reduce the work needed to add a language. Of course, developing formally unrelated grammars in parallel can also save work, since experience gained from one language can be useful in another one, and code can be shared by the copy-and-paste method. In this paper, however, we will focus on multilingual grammars with shared representations.

Our discussion will use the concepts and notation of GF, *Grammatical Framework* (Ranta, 2004b). The development of GF started in the Multilingual Document Authoring project at Xerox (Dymetman et al., 2000). It was from the beginning designed as a formalism for multilingual grammars: a monolingual grammar is just a special case of a GF grammar. Formally, a multilingual grammar in GF is a pair

$$\mathcal{G} = < \mathcal{A}, \{\mathcal{C}_1, \ldots, \mathcal{C}_n\} >$$

where $\mathcal{A}$ is an *abstract syntax* and $\mathcal{C}_i$ are *concrete syntaxes* for $\mathcal{A}$. The abstract syntax is a collection of *categories* and *functions*. It defines a *tree language*, which is a common representation of the *string languages* defined by the concrete syntaxes.

A concrete syntax is given by a method of *linearization*, which translates abstract syntax trees into strings (more generally, into records of strings and features). A multilingual grammar thereby defines a system of *multilingual generation* from the shared abstract syntax.

Generation is the primary direction of grammatical description in GF, and this gives GF grammars a special flavour that differs from many other grammar formalisms. However, GF is designed in such a way that linearization is always invertible to a *parser*, which maps strings back into abstract syntax trees.

1.2. Interlingua-based translation

It is easy to see what *translation* means in the context of a multilingual grammar: it is parsing from one language followed by linearization into another language. The abstract syntax works as an *interlingua*, and there is no need to define *transfer* in the sense of translation functions from one language to another.

But how do we manage to construct an interlingua for translation? The main point is that GF does not claim to have *one* interlingua. Instead, GF is a *framework* for defining interlinguas (i.e. abstract syntaxes) and their mappings into different languages (i.e. concrete syntaxes). In a typical application of GF, the interlingua is a precise semantical description of a limited application domain, which enables meaning-preserving translation in this domain.

1.3. Application grammars and resource grammars

Domain-specific multilingual grammars can give higher translation quality than general-purpose grammars. But it can be expensive to write them. Building a grammar of, say, aircraft maintenance manuals in Russian requires both *domain expertise*—knowledge of aircraft maintenance—and *linguistic expertise*—theoretical knowledge of Russian language.

A grammar for geometric proof systems in Russian requires a very different domain expertise but a similar linguistic expertise. Aren't such combinations of domain and linguistic expertise rare?

One goal of this paper is to show how a division of labour can be achieved in grammar engineering between domain experts and linguistic experts. This division is based on a distinction between *application grammars* and *resource grammars*.

An application grammar has an abstract syntax expressing the semantics of an application domain. A resource grammar has an abstract syntax expressing linguistic structures. The concrete syntax of an application grammar can be defined as a mapping to the abstract syntax of the resource grammar: it tells what *structures* are used for expressing semantic object, instead of telling what strings are used.

For example, consider an application grammar for mathematical proofs, defining a predicate saying that a natural number is even. The concrete syntax for French will say that the predicate is expressed by using the adjective *pair*. This is part of the application grammarian's domain knowledge; she needs this knowledge to rule out other dictionary equivalents of *even*, such as *lisse*. The resource grammar then defines how adjectival predication works in terms of word order and agreement, and how the adjective *pair* is inflected. This knowledge is expected from the linguist but not from the domain expert.

1.4. The resource grammar library

An application grammarian has access only to the abstract syntax of the resource grammar. Its concrete syntax is hidden from her, and the resource grammarian has all responsibility for it. The resource grammarian can, furthermore, make changes that are automatically inherited by all application grammars, and will never force to any manual changes in the applications.

To use software engineering terminology, resource grammars in GF play the role of *standard libraries*. The abstract syntax of a resource grammar is its API, *Application Programmer's Interface*.

Like everywhere in software production, standard libraries save work and improve quality by allocating different subtasks of grammar writing to different grammarians, and making it possible to reuse code. Recent efforts in developing GF have therefore centered around resource grammars. The ongoing GF Resource Grammar Project (Ranta, 2002) covers ten languages.

1.5. Shared representations in resource grammars

In an application grammar, the abstract syntax expressing a semantic model is the shared representation of different languages. However, it is even possible to use a common abstract syntax in a multilingual resource library. What is needed is a category and rule system that describes *pure constituency*, and abstracts away from details such as word order and agreement.

The ongoing GF Resource Grammar Project uses a common API for the ten languages that it covers. This API contains categories and functions for syntactic combinations (sentences, verb phrases, etc) and for some structural words. What is not covered by the common API is morphology and the major part of the lexicon.

Shared representations of the resource library are very useful for application grammarians. They make it easy to port applications from one language to another. At its simplest, a multilingual application grammar can be implemented as a *parametrized module*, where the parameters are the categories and functions of the common API. This is possible if all target languages use the same syntactic structures to express the same semantic contents.

Shared representations can also be used to make generalizations over similar (e.g. genetically related) languages. In the GF Resource Grammar Project, 75% of the code for French, Italian, and Spanish is shared in this way.

1.6. The GF language and its module system

The GF grammar formalism is a typed functional programming language, in many respects similar to ML (Milner et al., 1990) and Haskell (Peyton Jones, 2003). A more remote ancestor is LISP (McCarthy, 1960), which is functional but not typed. Like ML and Haskell, GF permits a powerful programming style by functions and user-defined data structures, controlled by static type checking. A detailed description of the GF formalism is given in (Ranta, 2004b) and a tutorial introduction in (Ranta, 2004c). What is not covered in those papers is the module system, which is a new feature of GF, and the main technical content of this paper.

A module system is useful both for software engineering and because of *separate compilation*. In a typical software development process, only some modules are changed at a time, and most modules do not need recompilation. In GF, this means in particular that resource grammar libraries can be precompiled so that application grammarians only need to compile modules that they are building themselves.

1.7. The structure of this paper

Section 2 explains how grammars are written in GF, using the module system of GF. Section 3 introduces the notions of application grammar, resource grammar, and grammar composition. Section 4 summarizes the ongoing GF Resource Grammar Project. Section 5 gives examples of applications that use the resource grammar library. Section 6 discusses practical issues of grammar engineering. Section 7 gives comparisons with some related work.

2. Introduction to GF and its module system

GF has three main module types: *abstract*, *concrete*, and *resource*. Abstract and concrete modules are *top-level*, in the sense that they appear in grammars that are used at runtime for parsing and generation. They can be organized into *inheritance hierarchies* in the same way as object-oriented programs, which enables code sharing between top-level grammars. Resource modules are a means of sharing code in cross-cutting ways, independently of top-level hierarchies.

2.1. Abstract syntax

An abstract syntax module defines a grammar of abstract syntax trees, which are the main device of representation of grammatical analysis. Depending on how the abstract syntax is defined, this representation can be seen as semantic or syntactic. We start with an example using a semantic representation, based on propositional logic.

An abstract syntax has two kinds of rules:

`cat` rules declaring *categories*

`fun` rules declaring tree-forming *functions*

The abstract module `Logic` introduces one category, propositions, as well as two functions, conjunction and implication.

```
abstract Logic = {
  cat Prop ;
  fun Conj, Impl : Prop -> Prop -> Prop ;
}
```

As usual in functional programming languages, the colon `:` is used to indicate typing and the arrow `->` to denote function types. Two-place functions (such as `Conj` and `Impl`) are written as functions whose values are one-place functions.

As an example of a syntax tree, the one corresponding to the formula $A\&B \supset B\&A$ is written

```
Impl (Conj A B) (Conj B A)
```

Thus function application is written as juxtaposition, and parentheses are used for grouping.

2.2. Concrete syntax

To define the relation between trees in an abstract syntax and strings in a target language, *concrete syntax modules* are used. A concrete syntax is always *of* some abstract syntax, which is shown by its module header. It must contain the following rules:

- for each `cat` in the abstract syntax, a `lincat` definition giving the linearization type of that category,
- for each `fun` in the abstract syntax, a `lin` definition giving the linearization function for that function.

The following is a concrete syntax of `Logic`:

```
concrete LogicEng of Logic = {
  lincat Prop = {s : Str} ;
  lin Conj A B = {s = A.s ++ "and" ++ B.s} ;
  lin Impl A B = {s = "if" ++ A.s ++ "then" ++ B.s} ;
}
```

Linearization types in GF are *record types*. The record type `{s : Str}`, has just one field, with the label `s` and the type `Str` of strings (more precisely: token lists). Labels are used for *projection* from a record using the dot (`.`) operator, so that e.g.

```
{s = "even"}.s
```

computes to the string `"even"`. The concatenation operator `++` puts token lists together.

Following the concrete syntax `LogicEng`, we can now linearize the syntax tree

```
Impl (Conj A B) (Conj B A)
```

into a record, whose `s` field contains the string

if A and B then B and A

2.3. Linguistically motivated abstract syntax

Abstract syntax trees that represent semantic structures are usually the most interesting choice if the goal is to translate between languages.

However, an abstract syntax can also be built in a more traditional linguistic way. An example is the module `Phrase`, which introduces the categories of sentence, noun phrase, verb phrase, transitive verb, common noun, and determiner, and three functions that put together trees of these types: verb phrase predication, transitive verb complementization, and common noun determination.

```
abstract Phrase = {
  cat S ; NP ; VP ; TV ; CN ; Det ;
  fun PredVP : NP -> VP -> S ;
  fun ComplTV : TV -> NP -> VP ;
  fun DetCN : Det -> CN -> NP ;
}
```

The function `PredVP` corresponds to the famous phrase structure rule

$$\mathrm{S} \rightarrow \mathrm{NP\ VP}$$

This correspondence is exact if the following linearization rule is given:

```
lin PredVP vp np = {s = np.s ++ vp.s}
```

Different languages typically have more complex versions of this rule, adding agreement (Section 2.14) and/or word order variation (Section 2.12).

An abstract syntax such as `Phrase` is usually too close to linguistic structure to form a good basis for translation, but it can be an excellent way to structure a resource grammar library; `Phrase` is actually a subset of the GF Resource Library API (Section 4).

2.4. Multilingual grammars

A multilingual grammar is a system of modules where one abstract syntax is paired with many concrete syntaxes. For instance, if we give another concrete syntax,

```
concrete LogicFin of Logic = {
  lincat Prop = {s : Str} ;
  lin Conj A B = {s = A.s ++ "ja" ++ B.s} ;
  lin Impl A B = {s = "jos" ++ A.s ++ "niin" ++ B.s} ;
}
```

producing Finnish instead of English, then the three modules `Logic`, `LogicEng`, and `LogicFin` form a bilingual grammar. The structure of this grammar is given in Figure 1. A bilingual grammar can obviously be extended by new languages simply by adding new concrete syntaxes.

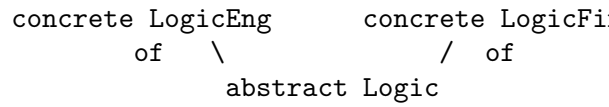

Figure 1. The module structure of a simple bilingual grammar.

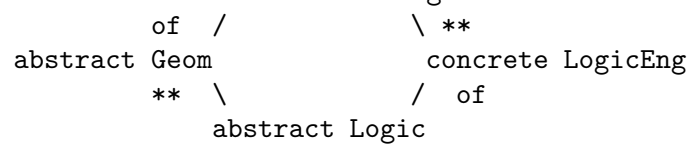

Figure 2. The module structure of a simple extended grammar.

However, even for the simple example of `Logic`, adding a language like German would pose the problem of word order, and adding French would pose the problem of mood. We will return to these problems in Section 2.9.

2.5. Inheritance

The module `Logic` alone does not generate any propositions (i.e. trees of type `Prop`), since it has no functions to form atomic propositions. The following module, `Geom`, is an *extension* of `Logic`. It *inherits* all categories and functions of `Logic`, and adds some new ones.

```
abstract Geom = Logic ** {
  cat Line, Point ;
  fun Parallel, Intersect : Line -> Line -> Prop ;
  fun Incident : Point -> Line -> Prop ;
}
```

Parallel to the extension of the abstract syntax `Logic` to `Geom`, the concrete syntax `LogicEng` can be extended to `GeomEng`:

```
concrete GeomEng of Geom = LogicEng ** {
  lin Intersect x y = {s = x.s ++ "intersects" ++ y.s} ;
  -- etc
}
```

The structure of the resulting grammar is given in Figure 2.

2.6. Hierarchies of semantic domains

The abstract syntax `Logic` can be used as a basis of many other extensions than `Geom`. For instance,

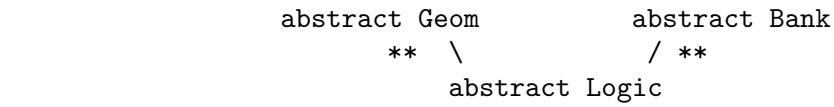

Figure 3. The module structure of an abstract syntax with extensions.

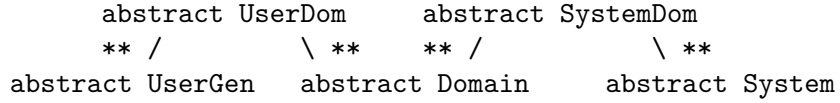

Figure 4. Multiple inheritance in a dialogue system.

```
abstract Bank = Logic ** {
  cat Customer ;
  fun HasLoan : Customer -> Prop ;
  -- etc
}
```

can be used as a description language for banks, customers, loans, etc, sharing the same basic logic as `Geom` but extending it with a different vocabulary. The structure of an abstract syntax with different extensions is given in Figure 3. An abstract syntax covers what is often called a *domain ontology* or a *domain model*. Its concrete syntax defines the terminology and syntax used for expressing facts in the domain. Sharing modules by inheritance reduces the amount of work needed to build such systems for new domains.

The module system also permits *multiple inheritance*. A typical example is a dialogue system, with separate grammars for user input and system replies. Both grammars inherit from a domain semantics, but also from generic dialogue move grammars that are independent of domain but different for the user and the system (Figure 4).

2.7. Semantic representation

As a semantic representation formalism, the "zeroth-order" language shown by the examples above is not very powerful. In full GF, the abstract syntax has the power of a *logical framework* with variable bindings and dependent types (Ranta, 2004b). Thus it is possible to formalize the syntax and semantics of Montague's PTQ fragment of (Montague, 1974) and extend the fragment with anaphoric expressions treated via the Curry-Howard isomorphism (Ranta, 1994; Ranta, 2004a). For instance, the progressive implication needed for analysing sentences of the type *if a man owns a donkey he beats it* can be defined by the abstract syntax rule

```
fun Impl : (A : Prop) -> (Proof A -> Prop) -> Prop ;
```

In this paper, however, we focus on the issues of modularity and therefore keep the semantic structure as simple as possible.

2.8. Resource modules

A resource module is "pure concrete syntax", in the sense that it defines linguistic objects without any reference to abstract trees representing them. There are two judgement forms occurring in resource modules: one for *parameters* and another for *auxiliary operations*.

2.8.1. *Parameters*

A *parameter type* is a finite datatype of values such as number (singular/plural) and gender (masculine/feminine/neuter). The values can be complex, i.e. formed by *constructors* that take other values as arguments. The GF notation for parameter types is shown in the following resource module which gives the complete definition of the non-composite verb forms of French according to *Bescherelle* (1997):

```
resource TypesFre = {
  param Number   = Sg | Pl ;
  param Person   = P1 | P2 | P3 ;
  param Gender   = Masc | Fem ;
  param Tense    = Pres | Imperf | Passe | Futur ;
  param TSubj    = SPres | SImperf ;
  param NPImper  = SgP2 | PlP1 | PlP2 ;
  param TPart    = PPres | PPasse Gender Number ;
  param VForm    = Infinitive
                 | Indicative Tense Number Person
                 | Conditional Number Person
                 | Subjunctive TSubj Number Person
                 | Imperative NPImper
                 | Participle TPart ;
}
```

The first six parameter types are simple enumerations of values. The last type `VForm` is the most complex one: it defines all of the actually existing verb forms, whose total count is $1 + 24 + 6 + 12 + 3 + 5 = 51$ different forms.

Parameter types are similar to *algebraic datatypes* in functional programming languages. Without algebraic datatypes, complex verb features of French would have to be defined in terms of 51 enumerated forms, which destroys all structure, or as a cross product of the atomic

features number, person, mood, tense, etc. The cross product would give something like $6 \times 4 \times 2 \times 3 \times 2 = 288$ forms, where most of the combinations would never be realized.

In contrast to algebraic datatypes in ML and Haskell, parameter types in GF may not be recursive or mutually recursive. This implies that their value sets are finite. This property is crucial for the compilation of GF grammars to efficient parsing and linearization algorithms.

2.8.2. *Tables*
The main usage of parameters is in *tables*, which are functions over parameter types. Since parameter types are finite, these functions can always be written explicitly as lists of argument-value pairs. Tables are, of course, a formalization of the traditional idea of *inflection tables*. A table is an object of a *table type*: for instance,

```
Number => Str
```

is the type of tables that assign a string to a number. The table

```
table {Sg => "mouse" ; Pl => "mice"}
```

is an example object of this type. The operator `!` is used for *selection*, i.e. application of a table. For instance,

```
table {Sg => "mouse" ; Pl => "mice"} ! Pl
```

is an object of type `Str`, which computes to `"mice"`.

Variables and wild cards can be used instead of constructors as left-hand sides of table branches. For instance, the table

```
table {NSg Nom => "Kommunist" ; c => "Kommunisten"}
```

defines the inflection of the German noun *Kommunist* by using one branch for the nominative singular and a common branch for the seven remaining cases.

2.8.3. *Operation definitions*
It is tedious and error-prone to repeat code with minor variations, for instance, to write tables for English regular nouns one by one. What is more, the grammarian will miss important generalizations if she does so. The means for eliminating such repetition in GF is *operation definitions*. An operation definition introduces a new constant with its type and definition. For instance, an operation forming regular nouns in English (or French) is defined as follows:

```
oper regNoun : Str -> Number => Str = \noun ->
  table {Sg => noun ; Pl => noun + "s"} ;
```

The operation `regNoun` is defined as a *function* that can be applied to strings to yield tables from numbers to strings. As in abstract syntax, the notation `A -> B` is used for function types, and function application is expressed by juxtaposition. The notation `\x -> t` (with a backslash before the variable) is used for lambda abstraction. (Also notice the gluing operator `+` which combines two tokens into one.)

In GF as in all functional programming, functions are the main vehicle of generalization, abstraction, and reuse. Since `oper` definitions can be of any type, including higher-order types (i.e. functions on functions), it is almost always possible to capture a common feature in two pieces of code by defining a function. As a rule of thumb, whenever a functional programmer is tempted to program by copy and paste, she should try to define a function instead.

In GF, `oper` definitions belong to resource modules, and can be used in concrete syntax as auxiliaries (see Section 2.9). The grammar compiler eliminates them by *inlining*, and they are hence not present in runtime grammars. In order for this to be possible, `oper` definitions cannot be recursive or mutually recursive. This, however, is not a restriction of expressive power, since resource grammars have no recursive datatypes. (An exception is the type of strings: for them, GF has a standard library providing some built-in functions.)

As operations are eliminated from runtime grammars, they are a bit like *macros*. There is an important difference, however: macros are expanded in the code by textual replacement, whereas operations enjoy *separate compilation*. Macros are, for instance, not type checked separately, but any type errors in them are reported as type errors in their use sites. Operations are type checked in the place where they are defined. Moreover, optimizations can be applied to operations, and are thereby done once and for all, covering all applications.

2.8.4. *Opening resource modules*

Resource modules have their own inheritance hierarchies, which are independent of abstract and concrete syntax hierarchies. However, the ultimate purpose of resource modules is use in concrete syntax modules. This is done by *opening* them. For instance, a French concrete syntax for `Logic` may open a resource called `TypesFre`, which defines parameter types, and `MorphoFre`, which defines inflectional patterns.

```
concrete LogicFre of Logic = open TypesFre, MorphoFre in
  {...}
```

Opening means that the parameters and operations defined in the resources can be used in the body of `LogicFre`. It differs from extension (`**`) in an important way: the contents of an opened module

are not inherited and hence not exported further. This is essential for information hiding.

If many modules are opened at the same time, there is a risk of name clashes. This problem is solved by *qualified* opening, which means that imported names have prefixes telling what modules they come from, e.g. `MorphoFre.regAdj`.

2.9. Concrete syntax revisited

2.9.1. *The Word and Paradigm model of morphology*

The Word and Paradigm model is one of the three main approaches to morphology distinguished by Hockett (1954). Its idea is close to traditional grammars, in particular those of classical languages. The model has a straightforward implementation in GF: a *paradigm* is a function f from Str to a table, which depends on one or more parameters:

$$\texttt{oper}f\colon \mathrm{Str} \;\rightarrow\; P_1 \;\Rightarrow\; \cdots \;\Rightarrow\; P_n \;\Rightarrow\; \mathrm{Str} = \ldots$$

A *word* has two interpretations: either it is the string given as argument to a paradigm; or, as we prefer, a zero-place abstract syntax function. The latter interpretation is similar to a unique index that identifies a dictionary lemma. If only one entry corresponds to a word, traditional dictionaries do not distinguish between the lemma as a string and the lemma as an identifier. However, if a word occurs with several paradigms, the corresponding lemmas are often made unique by the use of a device such as a subindex: for instance, *lie*$_1$ (n., pl. *lies*) vs. *lie*$_2$ (v., past *lay*) vs. *lie*$_3$ (v., past *lied*).

A *resource morphology* is a complete set of paradigms for a given language; for instance, the resource morphology of French in the GF Resource Grammar Project (Section 4) consists of the 88 Bescherelle verb conjugations expressed as operations, plus the (much fewer) different paradigms for nouns and adjectives.

A *resource lexicon* is a set of words paired with paradigms. The lexicon can hardly be complete for any language, since new words are being added all the time, and sometimes old words start to get inflected with new paradigms.

2.9.2. *Typed lexicon*

A *lexical entry* is the definition of the grammatical properties of a word. In general, it does not consist of just a word and a paradigm, but it must also give whatever *inherent features* the word has. For instance, common nouns in French have a gender as their inherent feature: the gender is not a part of the paradigm of the noun, but a constant property of the noun.

Lexical entries can be represented as records. An essential part of a resource morphology is its type system, telling what types of records are available. Each such type corresponds to a *part of speech*, as defined in purely morphological terms. For instance, here are the definitions of French nouns, adjectives, and verbs.

```
Noun      = {s : Number => Str ; g : Gender} ;
Adjective = {s : Gender => Number => Str} ;
Verb      = {s : VForm => Str} ;
```

Actually to build a resource lexicon is to define zero-place `oper` functions of lexical types. To do this in a controlled way, it is advisable to see the lexical types as *abstract data types*: their definitions are hidden from the lexicographer, who thus never writes records and tables explicitly, but uses operations defined in the resource morphology.

Here is a fragment of a French resource module, `MorphoFre`, with some operations that a lexicographer can use. We do not show the actual definitions of the operations; the user of the module is not supposed to see them, either. The module `MorphoFre` uses the parameter type definitions from the module `TypesFre` (Section 2.8.1 above), but it does not actually inherit them since its purpose is to hide them. Hence, `TypesFre` is just `open`ed (and not inherited) in `MorphoFre`.

```
resource MorphoFre = open TypesFre in {
  oper Noun, Adjective, Verb : Type ;
  oper nReg, nCheval : Str -> Gender -> Noun ;
  oper aReg, aMoral : Str -> Adjective ;
  oper vAimer, vVenir : Str -> Verb ;
  oper masculine, feminine : Gender ;
  }
```

Here is a fragment of a lexicon built by using the resource morphology:

```
resource LexFre = open MorphoFre in {
  oper Contenir : V = vVenir "contenir" ;
  oper Egal     : A = aMoral "égal" ;
  oper Point    : N = nReg "point" masculine ;
  }
```

One who builds a lexicon in this way has to know some French: to recognize which paradigms to use for which words, e.g. that *contenir* is "inflected in the same way" as *venir*. But she need not know the full details of the paradigms. In addition to the ease of use, hiding the definitions of parameter types and lexical entry types gives a guarantee against errors due to manipulating these objects on a low level. The lexicographer does not get the responsibilities of the morphologist.

2.10. The word-and-paradigm model generalized to syntax

We have spent a long time discussing morphology, lexicon, and the word-and-paradigm model. But what about syntax? The main idea is simple: syntax in GF is just a generalization of the word-and-paradigm model from words to phrases. The generalization goes (1) from lexical categories to phrasal categories, and (2) from zero-place functions to any-place functions. In the same way as every abstract lexical object (= zero-place function) is assigned a record with a paradigm and inherent features, so every abstract syntax tree (= arbitrarily complex function application) is assigned a record with a paradigm and inherent features.

For example: French common nouns and adjectival phrases, now of whatever complexity, have the linearization types

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

The adjectival modification rule

```
fun ModAdj : CN -> AP -> CN ;
```

forms complex common nouns: e.g. `ModAdj Maison Blanc` is linearized

```
{s = table {
   Sg => "maison" ++ "blanche" ;
   Pl => "maisons" ++ "blanches"
   } ;
 g = Fem
 }
```

Since the `ModAdj` rule is recursive, it is not possible to enumerate the linearizations of all such phrases. Instead, a linearization rule is a function from the linearization types of the constituents to the linearization type of the value. Here is the adjectival modification rule for French, with gender agreement (from noun to adjective) and inheritance (from the noun "head" to the whole construction), as the interesting features:

```
lin ModAdj N A = {
  s = table {n => N.s ! n ++ A.s ! N.g ! n} ;
  g = N.g
  }
```

2.11. The lexicon-syntax boundary

GF makes no formal distinction between lexicon and syntax. Grammar engineers may want to restore some of this distinction by placing "lexical" rules in different modules than "syntactic" rules. However, especially in grammars based on a semantical abstract syntax, it is not guaranteed that what is lexical in abstract syntax (i.e. zero-place) is also lexical in concrete syntax (i.e. expressed by one word). In a multilingual grammar, what is one word in one language may be many words, or none at all, in another language. Of course, having a single source for both lexicon and syntax also has the advantage of eliminating the problem of synchronizing two different components.

In both compilers and NLP systems, lexical analysis is usually separated from parsing. The main advantage is efficiency: parsing speed depends on the grammar size, whereas lexing speed is independent of the lexicon size. The GF grammar compiler achieves this advantage by removing lexical rules from the runtime parser, and replacing them with a *trie*—a non-cyclic finite-state automaton that analyses words in the way familiar from finite-state morphology (Huet, 2002).

The GF grammar compiler automatically specializes the lexical analyser to the grammar. Hence lexical analysis does not recognize words that the grammar cannot make use of. Nor does it generate lexical ambiguity which is irrelevant for the grammar. For instance, in geometry it may be irrelevant that the noun *point* can also be analysed as a verb.

2.12. Discontinuous constituents

Records with several string fields can be used for representing *discontinuous constituents*. For instance, the *topological model* is a traditional account of German grammar, where a clause has five parts, called Vorfeld, Left Bracket, Mittelfeld, Right Bracket, and Nachfeld (Müller, 1999). The record type representing such clauses is

```
{vf, lbr, mf, rbr, nf : Str}
```

In the GF Resource Grammar Project (Section 4), we have used a simplified model, in which a verb phrase has a verb part `s` and a complement part `s2`:

```
lincat VP = {s : VForm => Str ; s2 : Number => Str} ;
```

Sentences have a 3-valued parameter for word order, with values for verb-second, verb-initial, verb-final orders (typically used in main clauses, questions, and subordinate clauses, respectively). The predication rule comes out as follows:

```
lin PredVP np vp =
  let subj  = np.s  ! NPCase Nom ;
      verb  = vp.s  ! VInd np.n np.p ;
      compl = vp.s2 ! np.n
  in {s = table {
        V2 => subj ++ verb ++ compl ;
        VI => verb ++ subj ++ compl ;
        VF => subj ++ compl ++ verb
        }
     } ;
```

Notice the use of local `let` definitions to avoid multiple computation and to make the code clearer.

The above predication rule suffices for most generation purposes, since it generates the "unmarked" word order. For recognition, however, one might want to have alternative orders as free variants. For instance, in the verb-second branch, the complement instead of the subject may come before the verb. This can be expressed in GF by the `variants` construction, which introduces free variation between elements. The `V2` branch then becomes

```
V2 => variants {
  subj ++ verb ++ compl ;
  compl ++ verb ++ subj
  }
```

2.13. Interfaces and their instances

Sometimes different languages have concrete-syntax rules that have the same structure but operate on different parameter types. One example is subject-verb agreement: a common rule is that the noun phrase has some *agreement features* as inherent features, and the predicate verb phrase uses these features to instantiate some of its parameters. The noun phrase itself is inflected in a designated subject case. The following *interface* module declares the types of agreement features and cases, and the subject case, without actually defining them:

```
interface Agreement = {
  param Agr ;
  param Case ;
  oper subject : Case ;
}
```

Interface modules are thus similar to resource modules, but without giving definitions to the constants that they declare.

An *instance* of an interface looks like an ordinary resource module, except for the header that indicates the interface whose instance it is:

```
instance AgreementEng of Agreement = {
  param Agr = Ag Number Person ;
  param Case = Nom | Acc | Gen ;
  oper subject : Case = Nom ;
}
```

Notice the analogy between `abstract` and `concrete` on the top level and `interface` and `instance` on the resource level. As we will see in Section 3.2, one way to produce interfaces and their instances is by reuse of top-level grammars.

2.14. Incomplete modules and their completions

An interface module can be opened by another module, which must then be declared as *incomplete*, since it uses undefined types and operations. Incomplete modules are similar to *parametrized modules* in ML, where they are also known as *functors*. To give an example, the following concrete syntax is incomplete since it opens the interface `Agreement`:

```
incomplete concrete PhraseI of Phrase =
  open Agreement in {
    lincat NP = {s : Case => Str ; a : Agr} ;
    lincat VP = {s : Agr => Str} ;
    lin PredVP n v = {s = n.s ! subject ++ v.s ! n.a}
}
```

Complete modules are created by providing instances to interfaces:

```
concrete PhraseEng = PhraseI with
  (Agreement = AgreementEng)
```

The code contained in `PhraseI` can thus be shared between concrete syntaxes, by different instantiations of the interface `Agreement`. It works for any language in which the VP receives some agreement features from the subject and which has a rigid SVO or SOV word order. A further parametrization can be made to abstract away from word order:

```
order : Str -> Str -> Str -> Str ; -- interface
order S V O = S ++ V ++ O ;        -- SVO instance
order S V O = V ++ S ++ O ;        -- VSO instance
```

2.15. Compilation and complexity

A top-level GF grammar is a system of syntax-tree constructors (the abstract syntax) and their mappings to linearization functions (the concrete syntax). A functional programming language is available for writing grammars at a high level of abstraction. But isn't such a formalism too powerful to be tractable or even decidable? The solution to this problem is a compiler that takes full GF grammars into a format that is much simpler than a functional language. This format is called *Canonical GF*, GFC (see (Ranta, 2004b) for details).

Runtime applications of GF work with GFC grammars. The time complexity of linearization in GFC is linear in the size of the tree. The parsing problem of GFC has recently been reduced to the parsing problem for *Multiple Context-Free Grammars* (Seki et al., 1991), by Ljunglöf 2004. The parsing time is therefore polynomial in the size of the input, but the exponent depends on the grammar—in particular, on the number of discontinuous constituent parts.

The grammar compiler performs *type-driven partial evaluation*, which means that a canonical data structure is created for every linearization rule. An important consequence of this is that discontinuous constituents are only present in the compiled grammar if there are top-level linearization types that require them; resource modules can freely use discontinuous constituents without infecting the top-level grammar.

3. Application grammars and resource grammars

3.1. Semantic vs. syntactic structure

Most of the example grammars presented above have abstract syntaxes that encode semantic models. An abstract syntax encoding syntactic structure was given in Section 2.3.

The purpose of a semantical abstract syntax is to define the *meaningful* expressions of the application domain: in geometry, for instance, one cannot say that a point intersects a line, and this is guaranteed by the type of the function,

```
fun Intersect : Line -> Line -> Prop ;
```

Concrete syntax maps semantic structures into natural language, e.g.

```
lin Intersect x y = {s = x.s ++ "intersects" ++ y.s} ;
```

Conversely, the syntax tree that the parser finds for the sentence *x intersects y* has the form

```
Intersect x y
```

However, another analysis of the sentence arises if a syntactically motivated grammar is used. If we extend the module `Phrase` (Section 2.3) with the transitive verb *intersect*, we get the tree

```
PredVP x (ComplTV vIntersect y)
```

The former tree has more direct semantics than the latter. It is moreover less sensitive to differences among languages. Assume, for the sake of argument, that the only way to express this proposition in French is by using the collective-reflexive construction

x et y se coupent

It is easy to formulate the linearization rule on the semantically built abstract syntax:

```
lin Intersect x y =
  {s = x.s ++ "et" ++ y.s ++ "se" ++ "coupent"} ;
```

On the phrase structure level, however, the translation from English to French would require transfer to another abstract syntax tree, to something like

```
PredVP (ConjNP x y) (ReflTV vCouper)
```

3.2. Grammar composition

The possibility of doing translation without transfer is a strong argument in favour of a semantically built abstract syntax. Yet it is not appealing to write linearization rules directly on such a syntax, because the rules become linguistically ad hoc and repetitive, since linguistic generalizations are not factored out. They are also hard to write, since both domain-semantical and linguistic knowledge is needed. The solution to all these problems is to use grammars of different levels:

application grammars, taking care of semantics;

resource grammars, taking care of linguistic details.

The concrete syntax of an application grammar can then be defined by mapping into the abstract syntax of the resource grammar. The categories of the application grammar are given categories of the resource grammar as linearization types. For instance, geometrical propositions are linearized to sentences, and points and lines into noun phrases:

```
lincat Prop = S ; Point, Line = NP ;
```

The `Intersect` predicate is linearized into English by using VP predication and the transitive verb *intersect*:

```
lin Intersect x y = PredVP x (ComplTV vIntersect y) ;
```

In French, a reciprocal (reflexive) construction is used:

```
lin Intersect x y = PredVP (ConjNP x y) (ReflTV vCouper) ;
```

As a rule of thumb,

> An application grammar should only use applications of resource grammar functions, never any records, tables, or literal strings.

In other words, the resource grammar's linearization types should be abstract data types, and the only access to them should be via the interface provided by the abstract syntax of the resource grammar.

3.3. Compiling grammar composition

To make it formally correct to use the abstract syntax of one grammar as the concrete syntax of another one, we define a translation from pairs of top-level grammar rules into operation (`oper`) definitions:

$$\begin{cases} \texttt{cat}\ C \\ \texttt{lincat}\ C\ =\ T \end{cases} \implies \texttt{oper}\ C\ :\ \texttt{Type}\ \ =\ T$$

$$\begin{cases} \texttt{fun}\ f\ :\ A \\ \texttt{lin}\ f\ =\ t \end{cases} \implies \texttt{oper}\ f\ :\ A\ =\ t$$

Categories are thus translated to type synonyms for their linearization types, and functions to operations defined in terms of their linearization rules. It is easy to verify that type correctness is preserved in the derived `oper` definitions.

When resource grammar tree constructors are used in an application grammar, they are computed as `oper` constants and thereby eliminated. This is done at compile time by using partial evaluation. One effect of compilation is that the runtime grammar is minimized to match just the needs of the application grammar: the whole resource is not carried around, but just a part of it. For instance, it may be that the verb `vCouper` imported from the French resource lexicon to express the predicate `Intersect` is only used in the present indicative third person plural form `"coupent"`. Then the remaining 50 forms are not present in the runtime grammar. For another example, consider the German version of the `Intersect` predicate. The linearization rule can be

```
lin Intersect x y = PredVP x (ComplTV vSchneiden y) ;
```

The runtime linearization rule compiled from this is

```
lin Intersect x y = {s = table {
   V2 => x.s ! Nom ++ "schneidet" ++ y.s ! Acc ;
   VI => "schneidet" ++ x.s ! Nom ++ y.s ! Acc ;
   VF => x.s ! Nom ++ y.s ! Acc ++ "schneidet"
   }
}
```

Even though the resource grammar category `VP` is discontinuous, verb phrases are not constituents in the application grammar, and no discontinuous constituents are therefore present at runtime. This means that the parser has a lower complexity in the compiled application grammar than in the full resource grammar (Section 2.15).

3.4. Supplementary levels of grammars

We have said that application grammars define semantic structure and resource grammar define linguistic structure. But there is no fixed set of structural levels: grammar composition is associative, and any number of composed grammars can be compiled into one runtime grammar. Thus grammar composition of GF can be used for multi-phase natural language generation, where the transition from semantics to text goes through several levels. Two supplementary levels of representation have actually been found useful in GF-based projects: *language-independent API:s* and *derived resources*.

The idea of language-independent API:s is that, even though languages are "put up" in different ways, it is often possible to "see them as" having the same structure after all. In the multilingual resource grammar library, it is very useful to have a common API for different languages, so that the library is easier to learn and to use, and it becomes possible to write parametrized modules depending on it. At the same time, the "native" API of each language can be more fine-grained and give access to structures specific to that language,

Derived resources abstract away from the "linguistic way of thinking" of the the core resource grammar API:s. An example is a predication library, which is a parametrized module defining functions for applying n-place verbs and adjectives simultaneously to n arguments. For instance, such a library has the function

```
fun predTV : TV -> NP -> NP -> S ;
lin predTV x y = PredVP x (ComplTV vIntersect y) ;
```

which gives a concise way to linearize two-place predicates, e.g.

```
lin Intersect = predTV vIntersect ;
```

The application grammarian does not need to think in terms of subjects, verb phrases, and complements, but just in terms of logical argument places.

4. The GF Resource Grammar Project

The GF Resource Grammar Project has as its goal to create standard libraries for GF applications (Ranta, 2002). The latest release covers seven languages: English, Finnish, French, German, Italian, Russian, and Swedish. Later work has extended the coverage and added Danish, Norwegian, and Spanish. Of these languages, Finnish is Fenno-Ugric and the others are Indo-European. The library aims to give

a complete set of morphological paradigms for each language,

a set of syntactic categories and structures sufficient for generation of text and speech on different application domains,

a common API for the syntax of all languages involved.

The middle requirement is, of course, vague. In practice, it means that the library is developed in an application-testing-extension cycle. The coverage seems to converge towards a syntax similar to the Core Language Engine CLE (Alshawi, 1992).

The main goal of the Resource Grammar Project is practical: to enable programmers not trained in linguistics to produce linguistically correct application grammars. At the same time, resource grammars are a test case for GF from the linguistic point of view, since they are sizable general-purpose grammars. Writing a resource grammar is similar to grammar writing in the traditional sense of linguistics.

4.1. The coverage of the resource library

The current version of the language-independent syntax API has 61 categories and 136 functions. To give a rough idea of what this means in more familiar terms, the English version expands to 66,2569 context-free rules, and the Swedish version expands to 86,116 ones. These context-free projections, however, are not accurate descriptions of the grammars: the correct number of context-free rules can be infinite, since GF is not a context-free formalism.

The coverage of morphology modules varies from almost complete (French, Spanish, Swedish) to fragmentary (Norwegian). The most comprehensive esource lexica have been built for Swedish (50,000 lemmas) and Spanish (12,000).

4.2. Parametrized modules and language families

Languages inside a given family, such as Germanic or Romance, have similarities that reach from vocabularies related by sound laws to parameter systems and agreement rules. In the GF resource project, these similarities have not (yet) been exploited for Germanic languages, but an experiment has been made for Romance languages. When the French version had been written, we wanted to see how much of the code could be reused when writing the Italian version. It was not copy-and-paste sharing that was interesting (this had been used between German and Swedish), but inheritance and parametricity. Thus we identified the reusable parts of the French type system and syntax, and moved them to separate Romance interface modules. The end result was a grammar where 75 % of code lines (excluding blanks and comments) in the Italian and French syntax and type systems belong to the common Romance modules. No effort was made to share code in morphology and lexicon, but this could be a sensible project for someone interested in Romance sound laws.

At a later stage, Spanish was added to the group of Romance languages, with almost no new parametrization needed. Most recently, the Swedish grammar was generalized to Scandinavian, through addition of Danish and Norwegian. In this family, 90 % of syntax code is shared.

5. Application projects

5.1. Multilingual authoring

As an interactive generalization of interlingua-based translation, GF grammars support *multilingual authoring*, that is, interactive construction of abstract syntax trees, from which concrete texts are simultaneously obtained in different languages by linearizations (Khegai et al., 2003). The user of a multilingual authoring system can construct a document in a language that she does not know while seeing it evolve in a language she does know. Each authoring system has a limited domain of application. Pilot projects for such systems cover topics ranging from mathematical proofs to health-related phrases usable at a visit to a Doctor. Such a system typically involves 3–7 languages.

5.2. Dialogue systems

A dialogue system is a program that permits human-computer interaction in spoken language (but may also involve other input modes such as mouse clicks). Dialogue systems need two kinds of grammars: those

used in dialogue management and those used in speech recognition. Dialogue management grammars are used for parsing the user's utterances to dialogue moves. Speech recognition grammars help the speech recognizer to work faster and with less ambiguity when producing text input to dialogue management grammars.

The GF compiler derives speech recognition grammars in the Nuance format (Nuance Communications, 2002) automatically from GF grammars (Bringert et al., 2004). Since Nuance grammars are context-free, these grammars are not completely accurate, but they give conservative approximations. Moreover, if dialogue management is seen in the same way as interaction in a multilingual authoring system (Ranta and Cooper, 2004), it is possible to generate at each state of the dialogue an optimally restricted Nuance grammar that covers just those utterances that are meaningful in that state.

5.3. Object-oriented software specifications

Formalized software specifications are becoming more and more important in large-scale program development. One of the languages that is widely used is OCL (Object Constraint Language) (Warmer and Kleppe, 1999). However, even if programmers use formal specifications, it is natural language that management wants to see and that contracts with customers are written in. Formal specifications thus come with an additional cost of keeping informal specifications in synchrony with them. To bridge this gap, the software development system KeY (Ahrendt et al., 2004) is equipped with a GF-based natural-language authoring tool (Hähnle et al., 2002). The tool provides automatic translation of formal specifications into natural language, and also multilingual authoring for simultaneous construction of the two.

An interesting feature of the KeY grammar is that it is special-purpose but not closed. It has a core grammar that covers all OCL structures, but each application also has its own expressions for classes and methods: it may be bank accounts and withdrawals, or engines and brakes, etc. The application-specific expressions need not be just words from a lexicon, but they may be multi-word phrases with agreement and word-order phenomena. Since OCL grammar extensions are built by programmers whose training and focus is on formal specifications and not on natural language, there is urgent need to support grammar engineering through high-level API:s.

Building a German core grammar for KeY was a substantial project where a programmer without linguistic training wrote an application grammar by using the resource grammar library (unpublished "Studienarbeit" by Hans-Joachim Daniels). The results were encouraging:

the grammar produces correct even though sometimes clumsy German, is complete for OCL, and completely defined in terms of the resource grammar API.

6. Practical issues

6.1. Library browsing and syntax editing

Software engineering tools that support library browsing are obviously useful in resource-based grammar engineering. Syntax editing, which is the technique used in multilingual authoring applications, can be seen as a generalization of plain library browsing. The author of an application grammar can use syntax editing with a resource grammar to construct linearization rules. In each editing state, a menu is dynamically created with just those resource functions that are type-correct.

6.2. Grammar writing by parsing

As a step towards automatic application grammar construction, one can use a resource grammar as a parser. For instance, the linearization rule of the function

```
fun Intersect : Line -> Line -> Prop ;
```

can be built by sending an example sentence to the parser:

```
x schneidet y
```

The resource grammar parser builds from this the tree

```
PredVP x (ComplTV vSchneiden y)
```

which produces the linearization rule by grammar composition. (If the parse is ambiguous, the user is prompted to disambiguate.) Thus the application grammarian need not even look up functions from the library API, but just send target language strings to the parser! This procedure of course presupposes that the resource grammar has enough coverage to parse all input. The lexicon, in particular, may prove insufficient, since application grammars may use very special vocabulary. In such cases, it is important that the parser can localize unknown words and that the library module with lexical paradigms gives good support for adding new words to the lexicon.

6.3. Compilation of grammars into program modules

If grammars are only usable through an interpreter of the grammar formalism, their usefulness as software components is limited. Therefore, the GF grammar compiler is being extended by new back-ends that produce modules in main-stream programming languages. These modules must have high-level API:s giving access to grammar-based functionalities: parsing, generation, and translation. Currently GF can generate Java programs that perform generation, parsing, translation, and syntax editing (Bringert, 2005).

7. Related work

7.1. Records vs. feature structures

Like many other formalisms, e.g. PATR (Shieber, 1986) and HPSG (Pollard and Sag, 1994), GF uses records to represent linguistic information. In the other formalisms, records are often called *feature structures*. Let us compare a GF record with a PATR feature structure, representing the French noun *cheval*.

```
{s = table {                    {s = "cheval" ;
       Sg => "cheval" ;          n = Sg ;
       Pl => "chevaux"} ;
 g = Masc}                       g = Masc}
```

The GF record gives the inflection table and the inherent gender of the noun. The PATR record shows just the singular form, and indicates the number singular in the same way as the gender. The difference is explained by the fact that, in GF, records are obtained as *linearizations of trees*, whereas in PATR, as *parses of strings*. In GF, you start from the abstract object `Cheval`, and get both the singular and the plural forms. In PATR, you start from the string `"cheval"`, and thus you have only the singular form. The PATR record can be obtained as an *instance* of the GF record—by selecting the `Sg` field of the table.

7.2. Modularity in grammar engineering

Since the influential paper of Parnas (1972), information hiding has been considered the key aspect of modularity in software engineering. In grammar engineering, information hiding is not a wide-spread idea. It has even been said that information hiding is the very antithesis of productivity in grammar writing (Copestake and Flickinger, 2000).

For instance, a feature used in morphology may surprisingly turn out to be useful in semantics (*ibid.*). Blocking the view of other modules hence prevents the working grammarian from finding generalizations. As a more general argument, Copestake and Flickinger point out that grammar engineering is still more like research than engineering, and that normal software engineering ideas do not therefore apply.

The modularity to which (Copestake and Flickinger, 2000) refer is the traditional division of a grammar to morphology, syntax, and semantics. In GF, this kind of modularity applies mostly to resource grammars. It may well be useful to know morphological features in semantics, but this is no argument against information-hiding between resource grammars and application grammars. While resource grammar writing may still involve linguistic research, application grammar writing can be made more like software engineering.

7.3. Grammar specialization

Writing special-purpose grammars is costly, because it requires specific combinations of competence. On the other hand, special-purpose grammars are computationally more tractable than general-purpose grammars, particularly in speech recognition. One way out of this dilemma has been offered by Rayner *et al.* in the CLE project (2000): to extract fragments from large grammars by *explanation-based learning*. In brief, this means intersecting the grammar with a corpus. The GF approach where application grammars are extracted by grammar composition has the same goal, but uses a different technique.

However, a combination of grammar composition and explanation-based learning is already implicit in the technique of grammar writing by parsing (Section 6.2 above). In this method, the "corpus" consists of all linearization rules of an application grammar given as example strings. One example per rule is enough, and the corpus is automatically extended by using linearization with the resource grammar. For instance, if the target language is German and the example is *x schneidet y*, also the variants *schneidet x y* and *x y schneidet* are recognized, when occurring in positions that require these word orders. Thus the corpus that is needed is smaller than in ordinary explanation-based learning.

7.4. Resource grammar libraries

The coverage of the GF Resource Grammar Library is comparable to the CLE fragment (Alshawi, 1992; Rayner et al., 2000). By this we mean coverage in the sense of what syntactic structures are included. Coverage in the sense of parsing corpora has not been compared.

The set of languages covered by the GF resource library is larger than any other comparable library known to us. The idea of accessing grammars of different languages through a common API does not seem to have been considered before.

8. Conclusion

We have shown how the module system of the GF grammar formalism makes it possible to share code between grammars and to obtain a division of labour in grammar engineering. Grammars can share code in various ways: the most important way is an abstract syntax defining a common semantic representation in a multilingual grammar, but it is also possible to share large parts of concrete syntax, using inheritance and parametrized modules.

The practical goal of introducing modularity in grammar engineering is to make grammars accessible to software engineers and thereby to improve the quality of natural-language input and output in computer programs when linguists are not available. In addition to resource grammar libraries and grammar development tools, it is important that grammars can be compiled into program modules in main-stream programming languages such as Java.

Acknowledgements

The author is grateful to Robin Cooper and Elisabet Engdahl for stimulating discussions, and to anonymous referees for a multitude of useful questions. The work was financed from grant 2002-4879, *Records, Types and Computational Dialogue Semantics*, from Vetenskapsrådet.

References

Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt: 2004, 'The KeY Tool'. *Software and System Modeling*. Online First issue, to appear in print.

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

Bescherelle: 1997, *La conjugaison pour tous*. Hatier.

Bringert, B.: 2005, 'Embedded Grammars'. Master's Thesis, Department of Computer Science, Chalmers University of Technology.

Bringert, B., R. Cooper, P. Ljunglöf, and A. Ranta: 2004, 'Development of multimodal and multilingual grammars: viability and motivation'. Deliverable D1.2a, TALK Project, IST-507802.

Copestake, A. and D. Flickinger: 2000, 'An open-source grammar development environment and broad-coverage English grammar using HPSG'. *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*.

Dymetman, M., V. Lux, and A. Ranta: 2000, 'XML and Multilingual Document Authoring: Convergent Trends'. In: *COLING, Saarbrücken*. pp. 243–249.

Hockett, C. F.: 1954, 'Two Models of Grammatical Description'. *Word* **10**, 210–233.

Huet, G.: 2002, 'The Zen Computational Linguistics Toolkit'. `http://pauillac.inria.fr/~huet/`

Hähnle, R., K. Johannisson, and A. Ranta: 2002, 'An Authoring Tool for Informal and Formal Requirements Specifications'. In: R.-D. Kutsche and H. Weber (eds.): *Fundamental Approaches to Software Engineering*, Vol. 2306 of *LNCS*. pp. 233–248, Springer.

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*. pp. 453–464, Springer-Verlag.

Ljunglöf, P.: 2004, 'Grammatical Framework and Multiple Context-Free Grammars'. In: G. Jaeger, P. Monachesi, G. Penn, and S. Wintner (eds.): *Proceedings of Formal Grammar, Nancy, August 2004*. pp. 77–90.

McCarthy, J.: 1960, 'Recursive Functions of Symbolic Expressions and their Computation by Machine, part I'. *Communications of the ACM* **3**, 184–195.

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 R. Thomason.

Müller, S.: 1999, *Deutsche Syntax Deklarativ*. Max Niemeyer Verlag.

Nuance Communications: 2002, 'Nuance'. `http://www.nuance.com`.

Parnas, D.: 1972, 'On the Criteria To Be Used in Decomposing Systems into Modules'. *Communications of the ACM* **15**, 1053–1058.

Peyton Jones, S. (ed.): 2003, *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

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

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

Ranta, A.: 2002, 'GF Homepage'. `www.cs.chalmers.se/~aarne/GF/`.

Ranta, A.: 2004a, 'Computational semantics in type theory'. *Mathematics and Social Sciences* **165**, 31–57.

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

Ranta, A.: 2004c, 'Grammatical Framework Tutorial'. In: A. Beckmann and N. Preining (eds.): *ESSLLI 2003 Course Material I*, Vol. V of *Collegium Logicum*. pp. 1–86, Kurt Gödel Society.

Ranta, A. and R. Cooper: 2004, 'Dialogue Systems as Proof Editors'. *Journal of Logic, Language and Information*.

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

Seki, H., T. Matsumura, M. Fujii, and T. Kasami: 1991, 'On Multiple Context-Free Grammars'. *Theoretical Computer Science* **88**, 191–229.

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

Warmer, J. and A. Kleppe: 1999, *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley.