Grammatical Framework Tutorial
Revised version

Aarne Ranta
Xerox Research Centre Europe (Grenoble Laboratory)
Aarne.Ranta@xrce.xerox.com

February 19, 1999

1 Introduction

The grammatical framework (GF) is a framework for writing and using grammars. GF grammars have a special format that comprises not only syntax, in the usual linguistic sense, but also semantics, in a logical sense. Thus GF extends linguistic grammar formalisms by semantic notions, and logical formalisms by syntactic notions.

The purpose of GF is to enable a syntactically and semantically accurate description of languages. The functionalities that GF defines, in a generic way for any grammar of appropriate form, include parsing, generation, semantic verification by type checking, paraphrasing, stepwise editing, and meaning-preserving translation.

The name of GF, as well as many of its central ideas, are derived from **logical frameworks**. A logical framework is a generic system for writing and using mathematical theories, including logical calculi. It must comprise, not only the syntax of these calculi (the notion of well-formed formula in them), but also their semantics (the notion of valid judgement). In a way, GF is just a logical framework enriched with wider possibilities of concrete syntax.

Logical frameworks belong to research in logic and theoretical computer science, whereas GF also addresses the problems of computational linguistics: it aims at an accurate description of natural languages. Being a generic framework for grammars, GF can be compared with some other generic grammar formalisms, such as DCG (Definite Clause Grammar, closely corresponging to the programming language Prolog) and XLE (Xerox Linguistic Environment, implementing LFG, i.e. Lexical-Functional Grammar).

In this report, we will make very little comparison between GF and other grammar formalisms. It should be pointed out that the intended practical applications are very different. Both DCG and XLE have evolved in the linguistic tradition of looking for more and more comprehensive and elegant descriptions of entire natural languages, whereas GF has been designed to make it easy to define **sublanguages**, typically very limited fragments of entire languages used in specific technical areas. Thus the existing resources developed within DCG and XLE are formalizations of general linguistic theories, aimed to be applicable to almost any linguistic material, whereas the (still few) examples of GF grammars are typically natural-language systems corresponding to mathematical theories, such as arithmetic and geometry.

Comparing the size of GF itself and its applications with the size of the other systems mentioned, GF may look like a tiny boat in comparison to an ocean ship. We do not know how well GF scales up to huge applications for which XLE, for instance, is routinely used—to grammars with thousands of rules intended to apply to all kinds of texts. But this does not seem to be so important at this stage: we believe that there are situations in which a tiny boat is more useful than an ocean ship. Although small, GF is not meant to be a toy but a precision instrument.

GF stands both for a formalism and for its computer implementation. The implementation has been done in the functional programming language Haskell, a language widely used for the implementation of logical frameworks but seldom in linguistics. The kernel of GF consists of a few hundred lines of code, divided into a few modules, essentially defining a Haskell data type of grammars and generic parsing, type checking, and generation algorithms for that data type. The idea is to treat grammars as first-class Haskell data objects, which can be used and reused for different purposes possibly by doing some auxiliary Haskell programming. However, in addition to the kernel, we provide a line-based user interface that consists of a dialogue in which grammars can be experimented with. There is also a graphical interface designed for syntax editing on the basis of GF grammars. All of these sources (about 4500 lines of Haskell code at present, in version 0.4) are meant to be open for inspection and modification, while users of GF may naturally retain all rights for the further applications and for the grammars that they write.

However, the formalism of GF could very well be implemented in some other way. The linguistic information contained in GF grammars should not be lost even if the current implementation becomes obsolete. The document "Grammatical Framework Syntax and Semantics" presents the formalism from this perspective, specifying GF as a special-purpose programming language.

2 Related material

This document is aimed to be self-contained, so that all concepts and notations are explained here. No other material should be necessary for one who wants to write GF grammars. But we will usually not argue theoretically for the ideas implemented in GF.

Most of the linguistic ideas eventually leading to the development of GF are explained in Ranta (1994), while technical issues of the implementation are discussed in Ranta (1998). The underlying logical theory, **constructive type theory**, is explained in Martin-Löf (1984) and Nordström *et al.* (1990). Most of the central problems of logical frameworks are treated in Magnusson (1994). Carlsson & Hallgren (1998) explain Haskell's graphical system Fudgets, which we have used for our graphical interface, and discuss the interaction of Haskell programs with their environment in more general terms. All these works give further references on the topics discussed.

3 Overview

In logical frameworks, grammatical analysis is usually divided into context-free parsing and type checking. It is the former that is called the syntax of the language, while the latter is seen more as a matter of semantics. With the use of dependent types, type checking is, indeed, a very powerful tool, and computationally by far more demanding than context-free parsing. While context-free parsing has been able to employ well-established techniques for a long time, the problem of type checking dependent types is still a field of active research.

It is possible to use GF for generating parsers and generators for context-free grammars. While this usage leaves most of the characteristic features of GF unemployed, it provides a step of introduction to GF. Moreover, all GF grammars can be seen as having a context-free "skeleton", forming a level at which certain computational tasks are most conveniently performed. Section 4 is devoted to context-free grammars, of which, however, a mild extension will be given.

Context-free grammars work well for artificial languages, which have a simple context-free skeleton by design. In the description of natural languages, the context-free format does not work so well. One reason for this is the **morphological variation** of expressions, which could in principle be described by context-free means but would leave to an unappealing multiplication of categories and rules. More crucially, the connections between languages via a common abstract syntax would then be lost. For instance, French would have four categories of adjectives (masculine and feminine singular and plural), whereas English would only have one. It seems that there should be a level of abstract syntax that abstracts over such differences. In Section 5, we will present some ways in which GF can describe things like morphology and word order in abstraction from a deeper level of structure, without sacrificing the accuracy of description of concrete languages.

Section 6 explains some uses of dependent types in grammars: semantic well-formedness conditions, selectional restrictions, anaphora, and explicit variables. More generally, it illustrates the ways in which GF completes parsing with type checking. It also introduces the notion of **interactive editing**, which is the principal mode of working with the current GF interfaces. But the details of the editing process are only described in the document on GF Implementation and Interfaces.

Section 7 discusses the semantics of GF grammars. Denotational semantics, familiar to linguists from the work of Montague, can be given within GF by defining the interpretation function as generation to a formal language. An alternative is internal semantics, familiar to logicians from Martin-Löf's type theory. It requires yet another extension of the notation of GF, by definitional equalities.

Section 8 shows an application of GF to a formal language, a fragment of the programming language ML.

Section 9 discusses some ways in which GF grammars can be used in application programs.

The best way of reading this document is perhaps at a computer terminal, with GF installed, giving the possibility of trying out the examples and writing one's own grammars. In order to make the document as tutorial as possible, we will not specify the syntax of GF notation formally in the main text. The syntax and semantics of GF are specified systematically in a separate document, "Grammatical Framework Syntax and Semantics".

4 Context-free grammars in GF

4.1 An example English context-free grammar

Let us formalize the following context-free grammar in GF:

```
S  -> NP is A1
S  -> if S then S
A1 -> A1 or A1
NP -> every CN
CN -> A1 CN
CN -> number
A1 -> even
A1 -> odd
A1 -> prime
NP -> zero
```

This grammar gives an analysis of infinitely many sentences of arithmetic, including simple ones, such as

zero is even

and more complicated ones, such as

if every even or odd number is prime then every number is prime.

4.2 Category symbols

The rules of a context-free grammar are written in terms of certain **category symbols** and **syncategorematic strings**, also called **nonterminals** and **terminals**, respectively. What has to be known to understand the grammar thus includes, at least, what the category symbols are; the rest can be taken as terminals. In a GF file, correspondingly, there is a field to list the categories of the grammar, preceding the field listing the rules. For the grammar of the previous section, we need the category field

```
Categories S ; NP ; A1 ; CN ;
```

There may be other fields, and the category field may give additional information on the categories. We return to these possibilities later. The line above is already a valid piece of GF code.

Any set of distinct alphanumeric identifiers can be declared in the `Categories` field. We will try to use category names that are both systematic and traditional, if possible. Thus the above categories are sentence, noun phrase, one-place adjective, and common noun, respectively.

4.3 The GF presentation of context-free rules

In a `Rules` field, which comes after the `Categories` field, we then list the rules of the grammar. We will here deviate from context-free notation to achieve something more modular and more powerful: in each rule, we distinguish between **functional structure** and **linear form**. The functional structure of a rule is the "skeleton" consisting of its category symbols alone: the category on the left of the arrow is the **value type** of the rule, whereas the categories on the right are its **argument types**. A limiting case is a function with no argument types—in mathematical terms, a **constant**, in grammatical terms, a **lexical item**.

A GF rule is identified by three pieces of information: a **functional type**, a **linear pattern**, and a **rule label**, also to be called a **combinator**—in mathematical terms, the rule label is just a function symbol. These parts are organized as follows (the typewritten characters :, -, and ; belong to GF):

combinator : type - pattern ;

All these parts can be found in the GF version of the first context-free rule, the rule of predication with a one-place adjective:

```
PredA1 : (Q:NP)(F:A1)S - Q "is" F ;
```

The mnemonic name `PredA1` has been chosen as the rule label. The functional type is that of two arguments, of types `NP` and `A1`, with the value type `S`. The arguments are identified by variable symbols, `Q` and `F`, to which the linear pattern then refers. It says that the linear form of the predication is one where the linear form of the proper name is prefixed to the linear form of the adjective, with the string "is" in between.

Note that the strings occurring in linear patterns are enclosed in double quotes. This has proved to simplify the notation; we first experimented with a notation where everything in a linear pattern was a string unless otherwise interpretable, but this lead into a very complicated system of escape characters.

Lexical rules—in other words, rules for constants—can only have strings in their linear parts, because there is no argument variable to refer to:

```
Even : A1 - "even" ;
Zero : NP - "zero" ;
```

But there is no formal distinction between lexical and other rules in GF.

4.4 Abstract syntax trees

If we omit the linear pattern of a GF rule, we obtain a **typing judgement** in the sense of constructive type theory:

```
PredA1 : (Q:NP)(F:A1)S
```

By the rule of **functional application**, this judgement licences the formation of type-theoretical terms of the form

```
PredA1(Q,F)
```

of type `S`, provided `Q` is a term of type `NP` and `F` is a term of type `A1`. This term is the **abstract syntax tree** corresponding to the string formed in accordance with the linear pattern. Thus one could view GF rules as rules for building abstract syntax trees, together with information about how these trees relate to concrete strings. From this information, GF derives a **parser** and a **generator**, which are functions going from strings to trees and vice-versa.

For instance, the rules stated so far establish a relation between the string *zero is even*

and the tree

```
PredA1(Zero,Even).
```

The tree is a **parse tree** of the string, whereas the string is the **linearization** of the tree. In general, strings do not have unique parse trees, but trees always have unique linearizations (up to morphological variation).

4.5 The example English grammar in GF

Having to repeat some of the code already presented, let us show the complete GF grammar corresponding to the context-free example:

```
Categories S ; NP ; A1 ; CN ;

Rules

PredA1  : (Q:NP)(F:A1)S  - Q "is" F ;
CondS   : (A:S)(B:S)S    - "if" A "then" B ;
DisjA1  : (F:A1)(G:A1)A1 - F "or" G ;
Every   : (A:CN)NP       - "every" A ;
ModA1   : (A:CN)(F:A1)CN - F A ;

Number  : CN - "number" ;
Even    : A1 - "even" ;
Odd     : A1 - "odd" ;
Prime   : A1 - "prime" ;
Zero    : NP - "zero" ;
```

The reader who has an implementation of GF available, can experiment with this grammar by typing it into a file, which can then be loaded by the `import` command. The reader may also find the grammar in the GF distribution.

4.6 The corresponding French grammar

One of the consequences of the neat distinction between functional structure and linear form is that the linear form can be varied while the functional structure is kept constant. Two grammars related in this way define a very strong notion of equivalence between two sets of strings. They form a **bilingual grammar**, which is obviously just a special case of **multilingual grammars**. By a multilingual grammar, one can parse a string in accordance with one set of linear patterns and regenerate a string in accordance with another, thus obtaining a structure-preserving **translation** from one language to another. One can also combine pieces of documents written in different languages into one document, which can then be reproduced in each of the languages.

No substantial multilingual grammars can be defined in a linguistically realistic way until we introduce morphological parametres (in Section 5). But we can already illustrate the idea by giving a grammar of French for the same arithmetic fragment as the English grammar above:

```
Categories S ; NP ; A1 ; CN ;

Rules

PredA1  : (Q:NP)(F:A1)S  - Q "est" F ;
CondS   : (A:S)(B:S)S    - "si" A "alors" B ;
DisjA1  : (F:A1)(G:A1)A1 - F "ou" G ;
Every   : (A:CN)NP       - "tout" A ;
ModA1   : (A:CN)(F:A1)CN - A F ;

Number  : CN - "nombre" ;
Even    : A1 - "pair" ;
Odd     : A1 - "impair" ;
Prime   : A1 - "premier" ;
Zero    : NP - "z\\'ero" ;
```

In this very simple case, the French version of the English grammar is almost obtained simply by replacing English words by French words (it did require some planning to choose a fragment for which this is possible!). Now we can parse the English string

if every even or odd number is prime then every number is prime,

obtain the syntax tree

```
CondS(PredA1(Every(ModA1(Number,DisjA1(Even,Odd))),Prime),
      PredA1(Every(Number),Prime))
```

and generate the French string

si tout nombre pair ou impair est premier alors tout nombre est premier.

4.7 Permutation, reduplication, and suppression of constituents

The simple word-to-word correspondence between the English and French grammars above already has one exception: the order of constituents in the rule `ModA1` of adjectival modification. In French, the adjective comes after the noun, in English, before. (The reason why the French order has been chosen as the order of arguments in abstract syntax will get clear in Section 6: in a semantically more accurate description, the type of adjectives depends on a common noun argument.)

What the example of adjectival modification illustrates is that the order of constituents in the linear pattern is independent of the order of arguments in the function type. Even if we manage to make them match in one language, we may fail to do so in another language. In addition, as we will see, there are languages in which the order of constituents of a phrase may vary as a function of the larger structure of which the phrase is itself a constituent. (The German sentence is one example of this.)

Not only the order in which constituents appear can be chosen freely by the grammar writer, but also the number of occurrences. Thus there are **reduplications**, that is, repetitions of one and the same constituent, as well as **suppressions**. Reduplications are perhaps not very common, but the possibility to describe them shows that what we are calling the context-free fragment of GF is more powerful than context-free grammar. The following Finnish version of the first example grammar is not context-free:

```
Categories S ; NP ; A1 ; CN ;

Rules

PredA1  : (Q:NP)(F:A1)S  - Q "on" F ;
CondS   : (A:S)(B:S)S    - "jos" A "niin" B ;
DisjA1  : (F:A1)(G:A1)A1 - F "tai" G ;
Every   : (A:CN)NP       - A "kuin" A ;
ModA1   : (A:CN)(F:A1)CN - F A ;

Number  : CN - "luku" ;
Even    : A1 - "parillinen" ;
Odd     : A1 - "pariton" ;
Prime   : A1 - "jaoton" ;
Zero    : NP - "nolla" ;
```

The rule for `Every`, where we have chosen a somewhat colloquial construction *A kuin A* repeating the common noun *A*, together with the rule `ModA1` that generates infinitely many common nouns, makes the grammar more than context-free. The context-free rule

```
NP -> CN kuin CN
```

would permit phrases like

pariton luku kuin luku

which our grammar rules out.

The possibility of reduplication is just a consequence of the principle that all constituents may freely occur in linear patterns. It is perhaps a curiosity. But the suppression of arguments will play a central role in many of the grammars that we will discuss, especially in various cases of **anaphora**. For instance, the English pronominalization rule

```
Pron : (A:CN)(a:PN(A))PN(A) - "it" ;
```

which is sufficient for fragments where only inanimate beings are considered, forms a proper name of an individual of any domain `A` on the basis of some given proper name `a`, to which it refers. But the rule suppresses all information about both `a` and its type `A`. From the point of view of semantics, the suppressed argument is of course essential, and it may also be important in translation, since the French rule is

```
Pron : (A:CN)(a:PN(A))PN(A) - pron(Gen(a)) ;
```

yielding *il* or *elle* depending on `a`.

4.8 The context-free rule format

In the examples above, it was shown how the general notation of GF is used for encoding context-free grammars. For rules that are absolutely context-free (in particular, that do not permute, reduplicate, or suppress constituents), a special format can be used that is familiar from other grammar formalisms. For instance, the French grammar of Section 4.6 could be written

```
Categories S ; NP ; A1 ; CN ;

Rules

PredA1. S -> NP est A1 ;
CondS.  S -> si S alors S ;
DisjA1. A1 -> A1 ou A1 ;
Every.  NP -> tout CN ;
ModA1.  CN -> CN A1 ;

Number. CN -> nombre ;
Even.   A1 -> pair ;
Odd.    A1 -> impair ;
Prime.  A1 -> premier ;
Zero.   NP -> z\'ero ;
```

Having this format available in GF may save from some typing. It can also make it easier to reuse old context-free grammars. Of course, trying to use this format will also give demonstrations of how limited it is.

5 Refinements of natural-language syntax

This section is devoted to questions like morphology, word order, and discontinuity. It might not be interesting for a reader who just wants to use GF to describe formal languages. It might also be skipped by a linguist who is familiar with some treatment of these phenomena (in unification grammar, for instance) and is more interested in type checking, which is the main novelty of GF in the analysis of natural languages. But it should be stressed that the present way of formalizing morphology, word order, and discontinuity is motivated by the need of separating these matters from the abstract syntax and thus to make multilingual grammars possible.

5.1 Parametre types

An expression in a given language may have several different forms, depending on **morphological parametres**, also called **features**. Thus an English common noun has both a singular and a plural form, and so do verbs, but the forms of the latter also depend on person, tense, and mode. In a GF grammar, a list of **parametre types** can be declared in a `Parametres` field listing the types and their parametres, as in the following:

```
Parametres Num(sg,pl) ; Gen(masc,fem) ; Mod(ind,subj) ;
```

This list of parametres is adequate for a fragment of French or Italian, declaring the numbers singular and plural, the genders masculine and feminine, and the modes indicative and subjunctive (although the Italian grammar would rather say conjunctive instead of subjunctive). What parametres there are depends on language: parametres do not appear in the abstract syntax.

5.2 Categories depending on parametres

What parametres make sense for a given expression depends on the category of the expression. There are two kinds of dependencies. An expression may have **variation** with respect to a parametre. For instance, English and French common nouns vary in number. But they may also have **inherent features**. For instance, French common nouns are either masculine or feminine inherently.

Variant and inherent parametres are specified in the `Categories` field of a GF grammar, according to the pattern

category - variant - inherent

Only the category field is obligatory, as illustrated by all the context-free grammars of the previous section. If the "inherent" field is empty, it can be omitted, but the "variant" field cannot be omitted if the "inherent" field is inhabited. (Otherwise it could be ambiguous whether the remaining field is for variant or inherent features.)

For instance, an enriched version of our first example French grammar could have the category field

```
Categories S - Mod ; NP - - Gen,Num ; A1 - Gen,Num ; CN - Num - Gen ;
```

which says that sentences vary in mode and have nothing inherent, noun phrases do not vary but have inherent number and gender, and so on.

5.3 Operations

In the `Operations` field of a grammar, one can define functions on parametres, like the following definition of regular noun endings in English and French:

```
Operations nomReg(Num) = case (sg) -> "",
                              (pl) -> "s" ;
```

An operation can have arbitrarily many parametres; the following definition of the French copula ("to be") has $2 \times 2 = 4$ cases:

```
etre(Num,Mod)   = case (sg,ind)  -> "est",
                       (sg,subj) -> "soit",
                       (pl,ind)  -> "sont",
                       (sg,subj) -> "soient" ;
```

A definition may also refer to operations previously defined, which may save us some writing. The following operation yields the regular adjective endings in French:

```
adjReg(Gen,Num) = case (masc,n) -> nomReg(n),
                       (fem,n)  -> "e" + nomReg(n) ;
```

Note the use of the **pattern variable** `n`, defining the two cases uniformly for singular and plural numbers. Also note the **junction symbol** `+` showing that the string "e" is to be written together with the regular noun ending. Indeed, when an ending is used in a later rule, it is typically conjoined by means of the symbol `+` as well; for instance,

```
Even : A1 - case (g,n) -> "pair" + adjReg(g,n) ;
```

which generates the four forms *pair, pairs, paire, paires* as a function of gender and number. Cases could be taken here as well, but it is usually more convenient to define appropriate operations in the `Operations` field and only use pattern variables in the `Rules` part of the grammar.

5.4 Linear patterns with morphology

Morphological variation and inheritance concern not only words but arbitrarily complex phrases. The following revised version of the French rule of adjectival predication shows an **agreement** of the copula and the adjective to the number and gender of the subject. In addition, the whole construction has the mode parametre of the value type `S`. The mode parametre is inherited by the copula:

```
PredA1 : (Q:NP)(F:A1)S  -
                    case (m) -> Q etre(Num(Q),m) F(Gen(Q),Num(Q)) ;
```

Another example, adjectival modification, uses the field for inherent features, separated by a hyphen `-` from the linear pattern.

```
ModA1  : (A:CN)(F:A1)CN - case (n) -> A(n) F(Gen(A),n) - Gen(A) ;
```

Three kinds of expressions are used for morphological features: constants (like `sg`), pattern variables (like `n`), and parametre type identifiers applied to constituent variables (like `Gen(Q)`). The last-mentioned is GF's conventional way of referring to inherent features. (In Section 5.7 below, a fourth kind of feature expressions is introduced: applications of feature-valued operations.)

5.5 The consistency of a grammar

Morphological parametres generate lots of mutual dependencies between the categories, operations, and rules of a grammar. Thus there is a notion of a grammar's being **consistent**, in the sense that all of these dependencies are correctly respected. In type-theoretical terms, this amounts to the type checking of morphological and syntactic functions.

The user interface of GF provides a **consistency checker**, by which one can check one's GF grammars. As long as the consistency checker is not used, inconsistencies in grammars may cause Haskell runtime errors (or, alternatively, default values whose grammaticality cannot be guaranteed), typically arising in linearization. For instance, if we had written

```
PredA1 : (Q:NP)(F:A1)S  - case (m) -> Q etre(Num(Q),m) F(Gen(Q)) ;
```

in the previous section, no form could be produced for the adjective `F`, because its number parametre is not specified.

5.6 Variants depending on subsequent strings

All morphological operations considered above are ones that take morphological features as arguments and give strings as values. Now, the choice of a string can have further variations, typically depending on the string that follows. For instance, the English indefinite article is either *a* or *an*, depending on whether the subsequent word begins with a vowel. We could approximate this rule by the following definition:

```
indef = "a", "an" ("a", "e", "i", "o") ;
```

The first string gives the default form, after which the exceptions are listed with the triggering contexts in parentheses. This operation can then be used in the syntactic rule for indefinite noun phrases:

```
Indef : (A:CN)NP - indef A(sg) - sg ;
```

English, however, poses the problem that the orthographic form of a word does not determine its vocalicity. It is only safe to say that words beginning with *a* and *i* are vocalic; *e* and *o* can be consonants (*euphemism, one*), *u* is as often a consonant as a vowel, and *h* is sometimes vocalic (*hour*). One low-level solution to this problem is to use a special symbol (say, `@^`) to mark vocalic words and to restore the symbol in the **tokenization** process that precedes parsing; we will discuss this solution below in Section 5.12 on tokenization.

In French, the orthography reflects much better the pronunciation, in the sense that the latter can almost always be derived from the former. There is a rather general operation of **elision**, definable as follows:

```
elision = "e", "'" ("a", "e", "i", "o", "u", "y", "\\") ;
```

The backslash `\\` (doubled because it is the escape character of GF) marks, in LaTeX code, accented vowels: `\'e` gives *é*, etc. This leaves open the problen that *h*, too, is sometimes vocalic. The symbol combination `@^` can then be used in the same way as in English.

The following example shows how the French definite article, together with its possible fusions with prepositions, can be defined by a joint use of morphological features and variants such as elision. Prepositions, including the absence of one (`nul`), are treated as a morphological parametre type.

```
Prep(nul,aa,de) ;

elision  = "e", "'" ("a", "e", "i", "o", "u", "y", "\\") ;
pr(Prep) = case (nul) -> "",
                (aa)  -> "\\`a",
                (de)  -> "d" + elision ;
def(Gen,Num,Prep) =
 case
  (masc,sg,nul) -> "l" + elision,
  (masc,sg,de)  -> "du", "de l'" ("a", "e", "i", "o", "u", "y","\\"),
  (masc,sg,aa)  -> "au", "\\`a l'" ("a", "e", "i", "o","u","y","\\"),
  (fem, sg,c)   -> pr(c) ("la","l'"("a","e","i","o","u","y","\\")),
  (g,   pl,nul) -> "les",
  (g,   pl,de)  -> "des",
  (g,   pl,aa)  -> "aux" ;
```

5.7 Feature-valued operations

Sometimes it is necessary to define operations on morphological features that have features as values as well. For instance, the conjunction of two noun phrases in French has a gender that is feminine if and only if both conjuncts are feminine. The number of the noun phrase is always plural. This rule can be written

```
ConjNP : (Q:NP)(R:NP)NP - Q "et" R - collGen(Gen(Q),Gen(R)), pl ;
```

which uses the operation

```
collGen(Gen,Gen) = case (masc,masc) -> masc, (masc,fem)  -> masc,
                        (fem,masc)  -> masc, (fem,fem)   -> fem
                 : Gen ;
```

Notice that the parametre type is specified, to guarantee unique type checking.

5.8 Alternative forms of operations

Any operation can be defined either in the `Operations` field or locally in the linear patterns; the latter may be convenient if an operation is only needed once in the rules. Thus to define the French common noun forms *cheval, chevaux*, there are two possibilities: using a predefined operation,

```
nomAl(Num) = case (sg) -> "l", (pl) -> "ux" ;

Cheval : CN - case (n) -> "cheva" + nomAl(n) - masc ;
```

or defining the operation in the rule itself,

```
Cheval : CN -
  case (n) -> "cheva" + (case (sg) -> "l", (pl) -> "ux")(n) - masc ;
```

In this case, where the only dependence in the operation is on the number parametre of the whole construction, one could also define the linear pattern itself by cases:

```
Cheval : CN - case (sg) -> "cheval", (pl) -> "chevaux" - masc ;
```

The syntax of local definitions of operations is the same as the syntax of definitions in the `Operations` field, except that the whole definition is written in parentheses, without a concluding semicolon. Here is another example, defining the variants of the French word *si* ("if") in the rule for forming conditionals:

```
CondS : (A:S)(B:S)S -
   case (m) -> ("si", "s'" ("il", "ils")) A(ind) "alors" B(m) ;
```

5.9 Parametric word order

In German, the order of constituents in a sentence depends on whether the sentence is used as a main clause (direct order), after a modifier (indirect order), or as a subordinate clause. This phenomenon can be described by letting the German sentence have a parametre type of word order, and by taking cases in rules that construct sentences:

```
Parametres Ord(dir,indir,sub) ;

PredA1 : (Q:NP)(F:A1)S - case (dir)   -> Q "ist" F,
                              (indir) -> "ist" Q F,
                              (sub)   -> Q F "ist" ;
```

(Parametres other than order are ignored here.) The parametrization is put into use in rules constructing composite sentences, such as conditionals:

```
CondS : (A:S)(B:S)S - case (dir)   -> "wenn" A(sub) "dann" B(indir),
                           (indir) -> B(indir) "wenn" A(sub),
                           (sub)   -> B(sub) "wenn" A(sub) ;
```

(One could argue about the last two cases, which switch the order of the subclauses to make the parametre of the main clause variable.) Given these rules, plus the rules for *zero* and *even*, the English sentence

if zero is even then zero is even

is translated into the German (direct order) sentence

wenn Null gerade ist dann ist Null gerade.

5.10 Discontinuous constituents

In all cases considered by now, syntax trees are linearized into strings, such that the linearizations of the subtrees of a complex are substrings of the linearization of the whole tree. There are cases, however, where a constituent is divided into separate parts by intervening other constituents. The simplest example are perhaps two-part conjunctions such as *both–and*. The constituents of the expression

both bread and wine

are naturally seen to be *bread*, *wine*, and *both–and*. To capture this intuition, we declare a category `C2` of two-part conjunctions as one whose expressions consist of two parts, as follows:

```
Categories C2 - - - 2 ;
```

Thus the number of parts is the last field in a category declaration; the intervening fields are obligatory, even if they are empty. If the number is the default 1, the last field (and the other fields if they are all empty) can be omitted. The following rules are enough for building the example above (`MN` is the category of mass nouns):

```
ConjMN : (C:C2)(A:MN)(B:MN)MN - C.1 A C.2 B ;
conj   : C2 - ("both","and") ;
bread  : MN - "bread" ;
wine   : MN - "wine" ;
```

Notice the tuple notation `(x,y)` in the rule `conj`, which produces a two-part conjunction, and the projection notation `x.n` in the rule `ConjMN`, which uses one.

The corresponding French rules can be formulated without discontinuity, because it is the same element *et* that is repeated twice:

```
ConjMN : (C:C2)(A:MN)(B:MN)MN - C A C B ;
conj   : C2 - "et" ;
bread  : MN - "du pain" ;
wine   : MN - "du vin" ;
```

The number of parts attached to a category is thus not a property of the category in the abstract syntax, but, like morphological parametres, belongs to the grammars of individual languages.

The property of GF that subtrees are reflected as substrings in the linearization of a complex tree could be called the principle of **compositionality of linearization**. With discontinuous constituents, the principle generalizes in an obvious way to tuples of strings: every syntax tree c linearizes into a tuple of strings that can be used in the linearizations of all larger trees of which c is a constituent.

The introduction of two-part conjunctions as a discontinuous category in English could be avoided by producing the same effect by the rule

```
ConjMN : (A:MN)(B:MN)MN - "both" A "and" B ;
```

The crucial motivation of discontinuity is, again, in multilingual correspondences. The following section will show a more substantial example of this.

5.11 Example: verb phrases in English, French, and German

In English, there is a traditional analysis of sentences into noun phrases and verb phrases, the latter ones being further analysed into transitive verbs, complements, copulas, negations, etc. The following grammar describes a part of this structure.

```
Parametres
Case(nom,acc,about,to) ; Num(sg,pl) ;

Categories
S ; NP - Case - Num ; VP - Num ; V2 - Num - Case ; A1 ; A2 - - Case ;

Rules
PredVP     : (Q:NP)(F:VP)S  - Q(nom) F(Num(Q)) ;
ComplPosV2 : (F:V2)(Q:NP)VP - case (n) -> F(n) Q(Case(F)) ;
ComplNegV2 : (F:V2)(Q:NP)VP - case (n) -> do(n) "not" F(pl) Q(Case(F));
UsePosA1   : (F:A1)VP       - case (n) -> be(n) F ;
UseNegA1   : (F:A1)VP       - case (n) -> be(n) "not" F ;
ComplA2    : (F:A2)(Q:NP)A1 - F Q(Case(F)) ;
```

The parametre type `Case` is used for representing the ways in which noun phrases can occur in sentences. It includes both cases in the usual sense and prepositions. Notice, in particular, that two-place verbs and adjectives have an inherent feature of complement case.

In German, the category of verb phrases is problematic because of parametric word order, but this phenomenon can be explained by making VP discontinuous. The two parts could be described as the verbal element and the complement. The latter part may be empty.

```
Parametres
Ord(dir,indir,unt) ; Case(nom,acc,dat) ; Prep(null,von,zu) ; Num(sg,pl) ;

Categories
S - Ord - ; NP - Case - ; VP - - - 2 ; V2 - Num - Prep,Case - 2 ; A1 ;
A2 - - Prep,Case ;

Rules
PredVP     : (Q:NP)(F:VP)S  -
                case (dir)   -> Q(nom)      F.1(Num(Q)) F.2(Num(Q)),
                     (indir) -> F.1(Num(Q)) Q(nom)      F.2(Num(Q)),
                     (unt)   -> Q(nom)      F.2(Num(Q)) F.1(Num(Q)) ;
ComplPosV2 : (F:V2)(Q:NP)VP -
     case (n) -> (F.1(n), prep(Prep(F)) Q(Case(F)) F.2(n)) ;
ComplNegV2 : (F:V2)(Q:NP)VP -
     case (n) -> (F.1(n), prep(Prep(F)) Q(Case(F)) "nicht" F.2(n)) ;
UsePosA1   : (F:A1)VP       - case (n) -> (sein(n), F) ;
UsePosA1   : (F:A1)VP       - case (n) -> (sein(n), "nicht" F) ;
ComplA2    : (F:A2)(Q:NP)A1 - F Q(Case(F)) ;
```

Note that two-place verbs are also discontinuous, because they may have, as detached from the complement, a prefix such as *an* or *zu*. For instance, the three variants of the German equivalent to the sentence *John smiles to Mary* are

Johann lächelt Maria zu,

lächelt Johann Maria zu,

Johann Maria zulächelt.

The rule for the verb *zulächeln* is

```
Smile  : V2 - ("l\\\"achelt","zu") - null, dat ;
```

In French, we have a discontinuity problem created by the negation words *ne–pas*, where *pas* is placed between the verb and its complement. This can be solved by treating VP as a category with two parts. But a more difficult question is to find the proper places of **clitics**, the object forms of pronouns. In English and German, the object forms of pronouns are used in the same way as other noun phrases, but in French, they are (under certain conditions) placed before the verb. Thus the English sentence

the line contains it

is translated

la ligne le contient.

This does not extend to all complements: the sentence

1998 *is divisible by it*

is translated

1998 *est divisible par lui.*

The following grammar solves the problem of French clitics by making noun phrases discontinuous: there is what might be called a pre-verbal part and a post-verbal part. One-place adjectives have two parts just like verbs, the first part including the possible clitic complement. In comparison to the grammars above, we have added two noun phrases to illustrate the way in which their two parts are formed.

```
Parametres
Case(nom,acc,aa,par) ; Gen(masc,fem) ; Num(masc,fem) ;

Categories
S ; NP - Case - Gen,Num - 2 ; VP - Gen,Num - - 2 ;
V2 - Num - Case ; A1 - Gen,Num - 2 ; A2 - Gen,Num - Case ;

Rules
PredVP     : (Q:NP)(F:VP)S  -
   Q.1(nom) Q.2(nom) F.1(Gen(Q),Num(Q)) F.2(Gen(Q),Num(Q)) ;
ComplPosV2 : (F:V2)(Q:NP)VP -
   case (g,n) -> (Q.1(Case(F)), F(n) Q.2(Case(F))) ;
ComplNegV2 : (F:V2)(Q:NP)VP -
   case (g,n) -> (ne Q.1(Case(F)), F(n) "pas" Q.2(Case(F))) ;
UsePosA1   : (F:A1)VP -
   case (g,n) -> ("", F.1(g,n) etre(n) F.2(g,n)) ;
UseNegA1   : (F:A1)VP -
   case (g,n) -> (ne, F.1(g,n) etre(n) F.2(g,n)) ;
ComplA2    : (F:A2)(Q:NP)A1 -
   case (g,n) -> (Q.1(Case(F)), F(g,n) Q.2(Case(F))) ;

It   : NP - case (nom) -> ("il",""),
                 (acc) -> ("l" + elision,""),
                 (aa)  -> ("lui",""),
                 (par) -> ("","par lui") - masc ;
Zero : NP - case (c)   -> ("", prep(c) "z\\'ero") - masc ;
```

If ditransitive verbs are included in the fragment, the possibility of two clitics arises, which leads to further complications in the presence of the first and the second persons. This combinatorial problem can be solved by the introduction of more discontinuos parts; we leave this as an exercise to readers familiar with French clitics.

5.12 Tokenization

What is read by the parser and generated in linearization are not strings but lists of strings, called **tokens**. In most of the examples up to now, tokens can be found simply by taking the substrings separated by spaces, and token lists can be turned into strings by adding spaces between words. This procedure is, so to say, the minimal tokenizer, and it is recognized in GF by the name `mini`. Each of the grammars above could have the line

```
Tokenizer mini ;
```

as their first line, but this is not necessary because GF chooses `mini` by default.

The orthographies of most languages, however, require deviations from minimal tokenization. There are different rules for spacing before and after commas, colons, and full stops, as well as conventions about capital letters. GF currently provides the predefined tokenizers `text`, `latex`, and `code` (the last one treating all characters but letters as tokens). To modify a tokenizer or to build a new one requires some Haskell programming and a recompilation of GF, except for the possibility of declaring strings that should be regarded as tokens. Thus an English grammar can start with the lines

```
Tokenizer text
Tokens "John", "@^uncle", "European" ;
```

indicating that *John* and *European* are always written capitalized (the tokenizer would otherwise uncapitalize them in the beginning of sentences) and that *uncle* is a vocalic word (if the default is that words beginning by *u* are consonantic). The predifined tokenizers have been built in such a way that they restore `@^`'s when tokenizing strings and remove them when untokenizing.

It would certainly be possible to perform much of morphological analysis in tokenization, by using finite-state transducers, for instance. In this report, however, we try to describe as much as possible inside grammars.

On the other hand, it would certainly be better if tokenizers could be entirely defined inside GF grammars, why not simultaneously with the grammatical structures proper. The relation between grammars and tokenizers remains to be clearly defined.

6 Dependent types

6.1 Simple types vs. dependent types

This report does not aim to give an introduction to type theory, for which there are other sources. But some remarks on dependent types must be made to render the subsequent discussion possible to follow.

The types that we have until now given to syntactic combinators are **simple types**, which are either **basic types** (like `S`, `NP`, `VP`) or **function types** of the form

$$(x_1 : C_1) \cdots (x_n : C_n)C$$

where $C_1, \ldots, C_n, C$ are basic types. The semantics of the function type is given by the rule of application,

$$\frac{f : (x_1 : C_1) \cdots (x_n : C_n)C \quad c_1 : C_1 \quad \ldots \quad c_n : C_n}{f(c_1, \ldots, c_n) : C}$$

In fact, this format of function types is just a special case of simple type theory, since value and argument types are restricted to basic types. But it is the one that corresponds directly to context-free rules of the form

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

The application rule corresponds to the construction of syntax trees from subtrees.

Dependent types are obtained by generalizing the basic types by **families of types**, that is, types depending on terms—or, as one would say in a grammatical context, on syntax trees. The general form of a family of types is

$$C(x_1, \ldots, x_n)$$

where each of the variables x_i ranges over some given type. The function type construction can be generalized to

$$(x_1 : C_1) \cdots (x_n : C_n)C$$

which looks the same as in simple type theory, but each of the variables x_i may now occur free in the subsequent types. The application rule becomes

$$\frac{f : (x_1 : C_1) \cdots (x_n : C_n)C \quad c_1 : C_1 \quad \ldots \quad c_n : C_n(x_1 = c_1, \ldots, x_{n-1} = c_{n-1})}{f(c_1, \ldots, c_n) : C(x_1 = c_1, \ldots, x_n = c_n)}$$

where the equations $x_i = c_i$ designate **substitutions** of terms for variables.

6.2 Semantic control of well-formedness

In GF grammars, dependent type give a powerful technique of semantic control. Let us start with an example. A (simplified) grammar of street addresses would represent an address as a triple of a street, a city, and a country:

```
Categories

Address ; Country ; City ; Street ;

Rules

address : (x:Country)(y:City)(z:Street)Address - z y x ;

Finland    : Country - "Finland" ;
France     : Country - "France" ;
Helsinki   : City    - "Helsinki" ;
Tampere    : City    - "Tampere" ;
Hameentie  : Street  - "Hameentie" ;
Hameenkatu : Street  - "Hameenkatu" ;
```

Thus it is possible to form both correct addresses, like

```
address(Finland,Tampere,Hameenkatu) : Address
```

and incorrect (that is, nonexisting) ones,

```
address(France,Tampere,Hameentie) : Address
```

An additional definition is needed of the concept of address that is not just syntactically but also semantically well-formed.

With dependent types, it is possible to define well-formed addresses directly, by formalizing the intuition that a city is not just a city but a city of some country, and that a street is a street of some city of some country. We modify the grammar thus:

```
Categories

Address ; Country ; City (x:Country) ; Street (x:Country, y:City(x)) ;

Rules

address : (x:Country)(y:City(x))(z:Street(x,y))Address - z y x ;

Finland    : Country                    - "Finland" ;
France     : Country                    - "France" ;
Helsinki   : City(Finland)              - "Helsinki" ;
Tampere    : City(Finland)              - "Tampere" ;
Hameentie  : Street(Finland,Helsinki) - "Hameentie" ;
Hameenkatu : Street(Finland,Tampere)  - "Hameenkatu" ;
```

Now, type-checking the syntax tree

```
address(Finland,Tampere,Hameenkatu) : Address
```

succeeds, but type-checking the syntax tree

```
address(France,Tampere,Hameentie) : Address
```

returns two **unsolvable constraints**,

```
Helsinki = Tampere, Finland = France.
```

Section 6.4 gives more details about type checking.

6.3 Conditions and their proofs

This section is more directed mathematically than most of this Tutorial; a linguistically oriented reader can safely skip it.

Dependent types permit the encoding of logic in grammars by using the so-called **propositions-as-types principle**, also known as the **Curry-Howard isomorphism**. Let us introduce this notion by an example grammar, definining Peano style natural numbers, their $<$ predicate, and spans of the form $[x, y]$ where $x < y$ is forced by a semantic condition.

```
Tokenizer code Tokens "=>" ;

Categories

Nat ; Span ; Prop ; Proof (x:Prop) ;

Rules

0 : Nat        - "0" ;
s : (n:Nat)Nat - n "'" ;

Lt : (m:Nat)(n:Nat)Prop - m "<" n ;

Lt0 : (n:Nat)Proof(Lt(0,s(n))) - "0 <" n "'" ;
Lts : (m:Nat)(n:Nat)(c:Proof(Lt(m,n)))Proof(Lt(s(m),s(n))) -
  c "=>" m "' <" n "'" ;

span : (m:Nat)(n:Nat)(c:Proof(Lt(m,n)))Span - "[" m "," n "]" ;
```

Note the way in which the predicate `Lt` ($<$) is defined in terms of proofs. These definitions correspond to the two axioms

$$0 < n'$$
$$m < n \Rightarrow m' < n'$$

For instance, the proof that $3 < 5$ is represented as the object

```
Lts(s(s(0)),s(s(s(s(0)))),
    Lts(s(0),s(s(s(0))),
        Lts(0,s(s(0)),
            Lt0(s(0))))) : Proof(Lt(s(s(s(0))),s(s(s(s(s(0)))))))
```

which linearizes into

```
0 < 0'' => 0' < 0''' => 0'' < 0'''' => 0''' < 0'''''
```

The combinator `span` defines well-formed spans, from which linearization suppresses the proof of the well-formedness condition. Context-free parsing must thus return an open problem, a **metavariable** for the proof. For instance, parsing

```
[0''',0''''']
```

returns an **incomplete syntax tree** and a problem `[Proof]`,

```
span(s(s(s(0))),s(s(s(s(s(0))))),[Proof]) : Span
[Proof] : Proof(Lt(s(s(s(0))),s(s(s(s(s(0)))))))
```

In this case, a **refinement** of the metavariable by the proof shown in the previous paragraph solves the problem and completes the syntax tree. In any case, resolving open problems requires either **interaction** or **automatic proof search** or a combination of the two. (In Section 7.4 below, we will show a way in which this proces can be greatly simplified if the condition is decidable.)

6.4 How type checking completes context-free parsing

If dependent types are present in a grammar, successful context-free parsing does not guarantee well-formedness in the full sense of the grammar. The resulting syntax tree must be **type-checked**. As in the ALF system of Magnusson (1994), the outcome of type checking is not just "true" or "false", but a set of **open problems** and **constraints**. The result "true" corresponds to an empty set of problems and constraints. The result "false" corresponds to the presence of unsolvable constraints. But there are intermediate cases, permitting the solution of the open problems by interaction or by an automatic tactic.

The interactive process is demonstrated by an example GF session in the document "Grammatical Framework Implementation and Interfaces". In general, it is possible to start interaction by sending an incomplete syntax tree to the editor. For instance, in the grammar of Section 6.2, the syntax tree

```
address(Finland,[City],[Street]) : Address
```

leads to an interaction where the open problems

```
[City] : City(Finland), [Street] : Street(Finland,[City])
```

must be solved next.

As an example of a tactic, GF, like ALF, uses **unification** to find necessary solutions. With dependent types, this is sometimes a surprisingly powerful tactic. Thus the incomplete syntax tree

```
address([Country],[City],Hameenkatu) : Address
```

is immediately completed into

```
address(Finland,Tampere,Hameenkatu) : Address
```

because the type of `Hameenkatu` does not permit any other solutions. This example shows that semantic control does not always complicate editing but it may also greatly simplify it.

6.5 Selectional restrictions

Perhaps the most characteristic use of dependent types in logic is to formalize **type restrictions** of functions. For instance, the propositional function Prime is defined for natural numbers but not for triangles; Equilateral is defined for triangles but not for numbers. Thus one first defines numbers and triangles as sets,

$$\mathrm{Nat} : \mathrm{set}, \ \mathrm{Triangle} : \mathrm{set}$$

and then the two properties as functions over these sets,

$$\mathrm{Prime} : (x : \mathrm{Nat})\mathrm{prop}, \ \mathrm{Equilateral} : (x : \mathrm{Triangle})\mathrm{prop}.$$

Predicate calculus, from the type-theoretical point of view, has just one set, a domain of individuals

$$D : \mathrm{set}$$

The above notions are all defined as predicates, that is, as propositional functions over the domain D:

$$\mathrm{Nat}, \mathrm{Triangle}, \mathrm{Prime}, \mathrm{Equilateral} : (x : D)\mathrm{prop}.$$

Thus it is meaningful to say that, for instance, the number 3 is equilateral.

Quantifiers in predicate calculus are functions from propositional functions to propositions. For instance,

$$\exists : (B : (x : D)\mathrm{prop})\mathrm{prop}.$$

Thus the proposition

$$\exists((x)(\mathrm{Triangle}(x) \& \mathrm{Prime}(x)))$$

says that some triangle is prime.

In type theory, where predicates are domain-restricted, quantifiers must also be. Thus the existential quantifier is

$$\exists : (A : \mathrm{set})(B : (x : A)\mathrm{prop})\mathrm{prop}.$$

Given a set and a propositional function over the set, for instance,

$$\mathrm{Nat} : \mathrm{set},\ \mathrm{Prime} : (x : \mathrm{Nat})\mathrm{prop},$$

we can apply the quantifier to get

$$\exists(\mathrm{Nat}, \mathrm{Prime}) : \mathrm{prop}.$$

But it is not possible to get

$$\exists(\mathrm{Triangle}, \mathrm{Prime}) : \mathrm{prop},$$

because the propositional function Prime is not defined for triangles.

Facts like the one that it is possible to say

some number is prime

but not

some triangle is prime

are known as **selectional restrictions** in linguistics. The quantifier example of the previous section shows how selectional restrictions can be formalized by using dependent types. What is more, the use of dependent types in grammar gives a systematic way of specifying how selectional restrictions are inherited and how they get modified in grammatical constructions.

In a GF grammar, the `Categories` field may assign to each category (except the first one) a context on which the category depends. The context can usually be abbreviated as a list of categories. For example, the declarations

```
Categories S ; CN ; PN(CN) ; A1(CN) ; A2(CN,CN) ;
```

tell that S and CN are constant (in other words, non-dependent) categories, while PN and A1 depend on an argument of type CN, and A2 depends on two such arguments. Intuitively, the category CN corresponds to the type of sets in mathematics, where each set is a **domain of individuals**. The adjective categories A1 and A2 correspond to the types of one- and two-place **propositional functions**, respectively, defined for some specific domains of arguments. The category PN corresponds to the type of the **elements of a set**. For instance, we may have

```
Nat  : CN          - "number" ;
Tr   : CN          - "triangle" ;
Pr   : A1(Nat)     - "prime" ;
Eql  : A1(Tr)      - "equilateral" ;
Div  : A2(Nat,Nat) - "divisible" ;
Zero : PN(Nat)     - "zero" ;
Trt  : PN(Tr)      - "the triangle t" ;
```

In predication, it must be required that the types of the predicate and its arguments match. This effect could be achieved by specific rules, such as

```
PredA1Nat : (a:PN(Nat))(F:A1(Nat))S - a "is" F ;
PredA1Tr  : (a:PN(Tr))(F:A1(Tr))S   - a "is" F ;
```

but the general solution is to use **domain parametres**, that is, introducing arguments of type CN.

```
PredA1 : (A:CN)(a:PN(A))(F:A1(A))S  - a "is" F ;
```

6.6 Type checking and selectional restrictions

While the domain-specific predication rules are in fact just context-free in power, the general rule surpasses context-free grammars in two ways. First, if there is an infinite number of CN's, no finite set of context-free categories and rules can replace the general rule. (A context-free grammar is, by definition, a finite set of rules over a finite set of categories.) Second, the domain parametre of the general rule is suppressed in linearization. What context-free parsing can do is just return a **metavariable** to mark the unknown constituent. Thus the context-free parse tree for the string

zero is prime

is a tree with an unknown constituent [CN>,

PredA1([CN],Zero,Pr)

and it is only by subsequent **type inference** that the complete syntax tree

PredA1(Nat,Zero,Pr)

is obtained.

Now, type inference may fail, and the type checker (of which type inference is a subordinate part) then returns an incomplete tree with some constraints on the metavariables. The string *the triangle t is prime*

results in

```
PredA1([CN],Zero,Pr), [CN] = Nat, Nat = Tr
```

In this case, the constraints are in fact unsolvable: there is no way to complete the tree. In some other cases, the constraints can be solved by interaction with the type checker.

In many logical formalisms, metavariables are represented uniformly by **question marks** `?`. The syntax of these languages guarantees that expressions with metavariables are uniquely parsed. In the beginning, question marks were used in GF, too, but they proved to lead to annoyingly many parses in many grammars. Using different metavariables for different categories makes it possible to parse them in a unique way.

6.7 Variable bindings

In the language of mathematics, as well as in many formal languages, there are **variables** that are first introduced by **variable bindings** and then used like ordinary expressions. For instance, the sentence

there exists a number x such that x is both even and prime

first binds the variable x and then uses it as a name of a number. In type-theoretical terms, variable bindings create a **context** in which the variables are meaningful expressions.

In most "everyday" fragments of natural language, there are no explicit variables, but we will see, in the following section, that many grammatical structures can be understood in terms of **implicit variables**. On the level of abstract syntax, they are just like explicit variables, but they are suppressed in linearization.

To use explicit variables in languages defined in GF, one has to declare a list of variable symbols in an optional `Variables` field between the `Categories` and `Operations` fields. For instance,

```
Variables k, m, n (PN) ; A, B (CN, S) ;
```

This makes the symbols k, n, and m recognizable by the GF parser as expressions of the category PN, and A and B as expressions of the categories CN and S. In addition, these letters followed by primes (`'`) and by numerals are recognized (this is the way we obtain infinitely many variable symbols).

For instance, the rule of explicit universal quantification could be written as follows in English:

```
ForAll : (A:CN)(B:(x:PN(A))S)S - "for all" A(pl) x@B "," B ;
```

The notation `x@B` is used for referring to the bound variable `x` of the argument `B`. The part `@B` can actually be suppressed, by the convention that a bound variable with no argument indicated belongs to the first argument that has that variable bound.

Now, the sentence

for all numbers n, n is even or odd

is recognized and given the syntax tree

```
ForAll(Nat,(n)PredA1(Nat,n,DisjA1(Nat,Ev,Od)))
```

Notice the places in which the variable `n` occurs in the syntax tree. The first occurrence `(n)` tells that the variable is bound in the subsequent expression. The second occurrence is at an ordinary argument place for proper names.

The general form of a syntax tree with a variable binding is,

$$(x)b$$

called an **abstract**. An expression of this form has a type of the form

$$(x : A)B,$$

where B is a type under the assumption that $x : A$.

The syntax of variable-binding GF rules requires some explanation. In the rule above, the symbols `B` and `x` are both used in the linear pattern, although only `B` is declared as an argument of `ForAll`. What is more, `B` is treated as an expression of the category `S`, and not of the type `(x:PN(A))S`, which would be expected from the type of `ForAll`.

In the implementation of GF, the operation of (preliminary) parsing is only defined for "flat" function types, that is, types that only have simple categories as their argument types. Flat types correspond to context-free rules in the way explained in Section 6.1 above. This is very convenient for operations on concrete syntax, but requires a way of translating between real types and flat types. What this translation does is treat bound variables as ordinary arguments and, moreover, suppress all category dependencies. Thus the flat-type rule of `ForAll` is (something like)

```
ForAll : (A:CN)(x:VarPN)(B:S)S - "for all" A(pl) x "," B ;
```

In this rule, the linear pattern can be understood in the usual way. In analogy with the flattening of types, also syntax trees are flattened so that, for instance,

```
ForAll(A,(x)B)
```

becomes

```
ForAll(A,x,B)
```

The trick of treating bound variables as ordinary syntactic constituents is standard in implementations of logic. For instance, the abstract syntax of a lambda abstract $(\lambda x)b$ is something like `Lambda x b`, which does not differ from the representation of, say, a pair (a, b), which is something like `Pair a b`.

6.8 Hidden constituents in anaphoric expressions

A simple English structure with a variable binding is the following version of universal quantification:

```
Whatever : (A:CN)(B:(x:PN(A))S)S - "whatever" A "you choose" B ;
```

Notice that the variable `x` is implicit: it is not shown in the linear pattern. But the syntax tree of the expression `B` may contain occurrences of this variable—occurrences that are suppressed in linearization as well.

The following rules of **anaphora** are typical examples of constructions that may rely on implicit variables:

```
Pron : (A:CN)(a:PN(A))PN(A) - "it" ;
Def  : (A:CN)(a:PN(A))PN(A) - "the" A ;
```

While pronominalization (in mathematical English) suppresses all information, the formation of definite noun phrases preserves domain information.

The domain argument of `Pron` can often be found by type inference. But there is no general method of finding the `PN` argument, except some heuristics looking into the context of the expression. The concept of context here is, again, the technical concept familiar from type theory: a context is a sequence of declarations of variables. In a syntactic construction, contexts are created by bindings of variables.

To give an example, the GF parser (including type checking), when applied to the string

whatever number you choose it is even or odd

returns the incomplete syntax tree

```
Whatever(Nat,(h)PredA1(Nat,Pron(Nat,[PN]),DisjA1(Nat,Ev,Od)))
```

with the metavariable

```
[PN] : PN(Nat) (h : PN(Nat))
```

The context contains just one variable, which is directly of the desired type, so that it is no problem for a simple **anaphora resolution** tactic to complete the tree into

```
Whatever(Nat,(h)PredA1(Nat,Pron(Nat,h),DisjA1(Nat,Ev,Od)))
```

But such a tactic must be kept separate from type checking proper, since the preferred interpretation of the pronoun may be something that comes from outside the local context.

6.9 Type checking and translation

In mathematical language, the only (third person singular) pronoun in English is *it*. In French, there is variation in gender:

```
Pron : (A:CN)(a:PN(A))PN(A) - case (c) -> pron(c,Gen(A)) - Gen(A) ;
```

where `pron` is a suitably defined morphological operation. For instance, the English sentence

whatever line you choose, there exists a point such that it is incident with it

cannot be given the correct translation

quelle que soit la ligne, il existe un point tel qu'il soit incident avec elle without knowledge of the domain arguments of the English pronouns. In the opposite direction, however, a translation can be given on the basis of context-free parsing alone.

In general, the linearization of a syntax tree in one language may make use of any piece of information that is suppressed in another language. Because of this, a joint use of parsing and type checking (including both type inference, tactics, and user interaction) is essential in translation.

7 Semantic definitions

7.1 Denotational semantics

The best-known way of giving meanings to syntax trees is by **denotational semantics**, where one defines an **interpretation function** that assigns some semantic object to each syntax tree. This kind of semantics is familiar to the linguist from the work of Montague. In GF, denotational semantics can be given in two ways. One can write some Haskell code to define an interpretation function on syntax trees. But one can also define the interpretation function as a translation to a logical formalism, which is then conceived just as another language. The **compositionality** of interpretation, so important for Montague, follows from the compositionality of linearization (cf. Section 5.10 above).

For instance, the context-free fragment of Section 4.5 is interpreted in a higher-level logic by providing the following (entirely context-free!) grammar for the latter:

```
Tokenizer code
Tokens "\\rightarrow", "\\vee'", "\\Pi", "\\Sigma" ;

Categories S ; NP ; A1 ; CN ;

Rules
PredA1. S  -> NP ( A1 ) ;
CondS.  S  -> S \rightarrow  S ;
DisjA1. A1 -> A1 \vee' A1 ;
Every.  NP -> \Pi ( CN ) ;
ModA1.  CN -> \Sigma ( CN , A1 ) ;

Number. CN -> N ;
Even.   A1 -> Ev ;
Odd.    A1 -> Od ;
Prime.  A1 -> Pr ;
Zero.   NP -> 0 ;
```

For instance, the sentence

if every even or odd number is prime then every number is prime

is translated to the formula

$$\Pi(\Sigma(N, Ev \vee' Od))(Pr) \rightarrow \Pi(N)(Pr)$$

(which would look better had we enclosed multicharacter identifiers in LaTeX mboxes). The symbol $\vee'$ is used for the disjunction of one-place propositional functions. If a notation in a more familiar form is needed, it can be obtained by adding definitions, such as the rule of β conversion, in the way explained in the following section.

Another way of giving denotational semantics, and a more familiar one, would be to translate GF trees not into the strings of a formal language, but into the trees of one. This is only possible in a separate module, as explained in Section 9 below.

7.2 Internal semantics

The translation of natural language to a logical formalism leaves open the question of the semantics of the latter. One of the insights of the type-theoretical analysis of logic is that denotational semantics, if performed with precision, is just translation to another language, even if this other language were called a model instead of a language. (This is very clear if the semantics is implemented on a computer, which requires everything to be formulated explicitly by using some language.) Type theory has thus developed a way of formulating semantics internally. This is done by means of **definitional equalities**, a notion that is not found in predicate calculus.

Definitional equalities can be included in GF grammars in the last field, entitled `Definitions`. The following examples show, first, an **explicit definition** of the constant `two`, and, second, a **recursive definition** of the function `sum`.

```
two = succ(succ(zero)) : PN ;

sum(x,zero)    = x : PN,
sum(x,succ(y)) = succ(sum(x,y)) : PN ;
```

These definitions are in terms of the **primitive expressions** `zero` and `succ`; that is, expressions with no definitions.

```
zero : PN       - "zero" ;
succ : (n:PN)PN - "the successor of" n ;
```

In virtue of these definitions, the expression

the sum of two and two

which has the syntax tree

`sum(two,two)`

can be **computed to a normal form**

the successor of the successor of the successor of the successor of zero

by first computing the syntax tree into

`succ(succ(succ(succ(zero))))`

and then linearizing the tree.

The meanings of primitive expressions have just to be accepted as understood. They are, typically, the **canonical forms**, also called the **constructors**, of a set. GF has a special format of definition that lists constructors, shown in the following example:

```
Nat = {zero,succ} : CN ;
```

This definition must be thought of as completed by the rules for `zero` and `succ`.

7.3 Paraphrases

Definitional equalities generate a **paraphrase** relation among syntax trees and thus, derivatively, among the strings of a concrete language. Thus it is possible to generate parahrases for an expression. For instance, the grammatical rules

```
DeclCN  : (A:CN)(B:(x:PN)S)S - "if" x "is" indef A "then" B ;
ForAll  : (A:CN)(B:(x:PN)S)S - "for all" A(pl) x "," x B ;
Div     : A2 - "divisible by" ;
Ev      : A1 - "even" ;

ForAll(A,B) = DeclCN(A,B) : S ;
Ev          = ComplA2(Div,two) : A1 ;
```

together establish the following sentences as paraphrases:

for all numbers n, n is even,

for all numbers n, n is divisible by two,

if n is a number then n is even,

if n is a number then n is divisible by two.

Genarating paraphrases gives a possibility to create stylistic variation. But it may also be used to avoid ambiguities (by looking for unambiguous paraphrases) and to repair "holes" in multilingual grammars. It may happen, for instance, that a construction is missing from one language. Translation can then be given by means of a paraphrase.

7.4 Decidable semantic conditions

In Section 6.3, we considered a grammar defining spans $[x, y]$ as pairs of numbers x and y such that $x < y$. Parsing a string of the form `[x,y]` led to a search of a proof of the condition.

The interactive construction of proof objects, however, can be a nuisance. In the present case, like in many other cases, the proof is trivial and does not give any useful information. As the $<$ predicate between integers is decidable, a proof can in principle be constructed automatically.

Type-theoretically, decidable propositions can be represented as Boolean values and decidable predicates as Boolean-valued functions. The grammar must then tell how propositions are computed into a canonical Boolean value, either `True` or `False`. A corresponding version of the grammar of Section 6.3 is

```
Tokenizer code ;

Categories Nat ; Span ; Prop ; IsTrue (x:Prop) ;

Rules

0 : Nat        - "0" ;
s : (n:Nat)Nat - n "'" ;
```

```
True  : Prop - "True" ;
False : Prop - "False" ;

yes : IsTrue(True) - "yes" ;

Lt : (m:Nat)(n:Nat)Prop - m "<" n ;

span : (m:Nat)(n:Nat)(c:IsTrue(Lt(m,n)))Span - "[" m "," n "]" ;

Definitions

Lt(0,   n)    = True    : Prop ;
Lt(s(n),0)    = False   : Prop ;
Lt(s(m),s(n)) = Lt(m,n) : Prop ;
```

Notice how the dependent type `IsTrue(Lt(m,n))` encodes the semantic condition, which now has only one possible proof: `yes`.

When time comes to find a proof of the decidable condition, one refinement step suffices, the refinement with `yes`. When performing the refinement, GF type checks whether

```
yes : IsTrue(Lt(m,n))
```

which amounts to the same as checking whether

```
Lt(m,n) = True : Prop
```

which is of course decidable by pure computation, if `m` and `n` are constant numbers. For the same reason, the refinement fails with an unsolvable constraint if the condition is false.

8 Grammars of formal languages

As indicated by the example of translation into logical formalism in Section 7.1 above, formal languages present no radical differences. Their morphology is usually simpler (or nonexisting), and they need a different tokenization. On the other hand, there exist so many tools for dealing with formal languages that it is perhaps seldom useful to define them in GF. But there is one thing that can be done easily: to define a concrete notation simultaneously with type checking and operational semantics (the definitions of noncanonical constants).

8.1 Example: mini ML

The following grammar is a fragment of a simple functional programming language, a "mini ML", which has a polymorphic type system and a set of computation rules that can be experimented with in GF. Precedences are treated as morphological features, controlling the use of parentheses. A bigger mini ML is included in the GF distribution.

```
Tokenizer code Tokens "->" ;

Parametres Prec(one,two,three) ;

Categories Type - - Prec ; Exp(Type) - - Prec ; Dig ; Pos ;

Variables x, y, z, X, Y, Z (Exp) ;

Operations
par2(Prec) =
 case (one) -> ("(",")"), (two) -> ("",""),   (three) -> ("","") ;
par3(Prec) =
 case (one) -> ("(",")"), (two) -> ("(",")"), (three) -> ("","") ;

Rules

Dig1.    Dig -> 1 ;  (* decimal integers *)
Dig2.    Dig -> 2 ;
Dig3.    Dig -> 3 ;
Dig4.    Dig -> 4 ;
Dig5.    Dig -> 5 ;
Dig6.    Dig -> 6 ;
Dig7.    Dig -> 7 ;
Dig8.    Dig -> 8 ;
Dig9.    Dig -> 9 ;
UseDig.  Pos -> Dig ;
AddDig.  Pos -> Pos Dig ;
AddZero. Pos -> Pos 0 ;

Int    : Type            - "Int" - three ;   (* integers *)
Zero   : Exp(Int)        - "0"   - three ;
UsePos : (p:Pos)Exp(Int) - p     - three ;

Fun  : (A:Type)(B:Type)Type -                (* function type *)
  par2.1(Prec(A)) A par2.2(Prec(A)) "->" B - one ;
Abs   : (A:Type)(B:Type)(b:(x:Exp(A))Exp(B))Exp(Fun(A,B)) -
  "\\" x "->" b - one ;
Ap    : (A:Type)(B:Type)(c:Exp(Fun(A,B)))(a:Exp(A))Exp(B) -
  par2.1(Prec(c)) c par2.2(Prec(c))
    par3.1(Prec(a)) a par3.2(Prec(a)) - two ;

Definitions

Int  = {Zero,UsePos} : Type ;
Ap(A,B,Abs(A,B,b),a)  = b(a) : Exp(B) ;
```

9 What one can use a grammar for

9.1 GF operations on strings and trees

There is a constant ambiguity when speaking about expressions, whether they are strings (of characters in written language, of phonemes in spoken language) or trees (articulated grammatical objects). We can resolve this ambiguity by distinguishing between language in two senses—**phenomenal** and **structural**. In the phenomenal sense, a language is a collection of strings. In the structural sense, a language is a system of syntax trees. A GF grammar defines a language in both of these senses, and establishes a relation between them. Applications of GF grammars may then define operations on both levels and exploit the grammar-defined relations to relate these levels.

Phenomenal language is what one typically wants to read and write, character by character. But structural language is what one uses to define more complex operations on, such as translation and semantic interpretation. Even if one just wants to transform strings into other strings, for instance, the string $2+2$ into the string 4, one often goes via the structural level, in accordance with the schema

$$\text{string} \longrightarrow_p \text{tree} \longrightarrow_s \text{tree} \longrightarrow_l \text{string}$$

The first arrow, labelled p for parsing, and the last arrow, labelled l for linearization, are generically defined for all grammars in the GF format, and can be taken for granted in applications. Thus an application can be identified by how it defines the middle arrow, labelled s for semantic operations. The semantic operation can be, for instance,

computation into a normal form,

verification in the sense of type checking or consistency checking,

translation into another system of trees,

optimization into a more useful tree.

Some such operations are already defined in grammars. For instance, the simplest model of translation is one in which s is the identity operation, and one just takes the functions l and p from two different grammars. Type checking is also an operation provided by GF, and computation rules can be optionally provided by the grammar writer.

Thus an implementation of GF permits translations, computations, type checking, and paraphrasing without any other programming than writing GF grammars. A synthesis of all these operations is stepwise editing, which is the main mode of interaction in the graphical interface FGF. While implicitly editing an abstract syntax tree, the user has a simultaneous view of texts developing in all string languages that he has imported, provided that they have a common tree language.

9.2 User-defined operations

While computation rules in grammars can be used for defining recursive functions, and GF can thus be used as a functional programming language, computations easily become inefficient to perform and/or tedious to define. Thus one might consider defining semantic operations in a real programming language and interface them with GF syntax trees. If Haskell is used, it should be possible to make this interfacing in a modular way.

To give a simple example, the Peano-style natural numbers and their basic operations defined in Section 7.2 above can be interfaced with Haskell integers and arithmetic operations by a module interpreting and producing GF syntax trees (in their internal Haskell representation).

A particular class of applications leading outside GF are systems for **data base queries**. Such a system typically consists of a query language, a huge data base, and a search mechanism that transforms questions into answers. Natural-language interfaces for such systems should be easy to write using GF, by a set of parallel grammars (for each of the languages one wants to include) and by functions translating between GF syntax trees and expressions of the query language. In some cases, the query language could even be definable in one of the parallel GF grammars. But it would hardly make sense to redefine the data base itself in GF.

9.3 Predefined categories and combinators

The Grammatical Framework is just a framework, in which all content (categories and syntax trees) are user-defined in grammars. In functional programming languages, such as Haskell, it would similarly be be possible to define all objects from scratch, e.g. integers as Peano style numbers or as decimals like in Section 8.1 above. But most programming languages have **predefined** data types and data objects, such as integers or strings. Predefined objects not only save user's time but also permit more powerful (low-level) computation methods than would be possible inside the language.

The document on GF Implementation and Interfaces tells how users can customize GF by their own predefined categories and combinators. Currently in use (in Version 0.4) are **integers**, **strings**, and **Booleans**, with some well-known operations defined in terms of corresponding Haskell constructs.

The user who wants to add more predefined constants has to provide information about their parsing and linearization, and may also provide computation rules. All this is can be done by editing a data base in one module.

References

Magnus Carlsson & Thomas Hallgren, *Fudgets—Purely Functional Processes with applications to Graphical User Interfaces*, PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 1998.

Lena Magnusson, *The implementation of ALF—a proof editor based on Martin-Löf's monomorphic type theory with explicit substitution*, PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, Gothenburg, 1994.

Per Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.

Bengt Nordström, Kent Petersson & Jan Smith, *Programming in Martin-Löf's Type Theory. An Introduction*, Clarendon Press, Oxford, 1990.

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

Aarne Ranta, "A functional framework for grammars", manuscript, 1998.

Aarne Ranta, “Grammatical Framework Syntax and Semantics”, manuscript, 1999.

Aarne Ranta, “Grammatical Framework Implementation and Interfaces”, manuscript, 1999.