D2.1 GF Grammar Compiler API

March 17, 2011

Contract No.:	FP7-ICT-247914
Project full title:	MOLTO - Multilingual Online Translation
Deliverable:	D2.1. GF Grammar Compiler API
Security (distribution level):	Public
Contractual date of delivery:	M13
Actual date of delivery:	March 2010
Type:	Prototype
Status & version:	Draft (evolving document)
Author(s):	A. Ranta, T. Hallgren, et al.
Task responsible:	UGOT
Other contributors:	

The GF Grammar Compiler API

Aarne Ranta, Thomas Hallgren, et al.

MOLTO Deliverable 2.1

Contents

1 Introduction

GF, Grammatical Framework, is a programming language for multilingual grammars. GF is used in the MOLTO project to build translation systems. How to write GF grammars is specified in the numerous tutorials and manuals available via http://grammaticalframework.org. This report explains the compiler of the GF language:

- how GF source code is compiled to various formats usable in runtime systems
- how the developers can test their grammars
- how other formats (such as lexica and example sentences) can be converted to GF code
- how to call the compiler in various ways:
  - GF shell commands and scripts
  - Haskell programs
  - web applications

2 The compiler functionalities

2.1 The source and target languages

GF follows the traditional design of programming languages: there is a **source language** in which the programs (i.e. grammars) are written, and a **machine language**, which is used at run-time. The compiler converts the source language into the target language. The source language is called GF (Grammatical Framework) and the target language PGF (Portable Grammar Format). As a point of reference, GF can be compared with Java and PGF with JVM (Java Virtual Machine). The differences between GF and PGF on one hand, and Java and JVM on the other, are similar and represent general differences between **high-level** and **low-level** languages.

The following table compares some characteristics of GF and PGF. The comparison concerns the concrete syntax, except for the module system, where also the abstract syntax has a modular structure in GF but not in PGF.

Feature	**GF**	**PGF**
type system	rich	simple
user-defined types	yes, algebraic	no, just integers
data structures	records, tables	tuples
computation model	functional programming	PMCFG
module system	hierarchic, parametrized	flat
purpose	abstraction, discipline	efficiency
primary format	textual	binary
implemented in	Haskell	Haskell, Java, JavaScript

The differences in the type and module system indicate that GF makes more distinctions than PGF. These distinctions enable compile-time sanity checking of grammars. Some examples are shown in the section below.

The difference in "primary format" is not essential. It means that `.pgf` files, used for storing PGF, are binary, whereas `.gf` files, written by GF programmers, are textual. This distinction is not a hard one, because there is also a textual debugging format for PGF and a binary format for GF object files (`.gfo` files), which is used for storing separately compiled GF modules before linking them into a PGF file.

The implementation language is not essential either. But the only available GF compiler is at the time of writing implemented in Haskell. However, there are PGF interpreters written in Haskell, Java, and JavaScript, and interpreters in more languages (C and C#) are under construction. This divergence is natural. Run-time PGF interpreters are useful when embedding GF applications in different host languages. Since PGF is a simple language, writing a new interpreter is not a huge effort (for instance, for Java it was done in a few weeks). But a GF compiler is a much more substantial program. Fortunately, it is needed only when the grammars are developed: run-time applications need only PGF and can forget about GF. Therefore it is no problem that the GF compiler is implemented in another language than the run-time applications.

2.1.1 Compilation examples

PGF is a combination of two things:

- dependent type theory, used for abstract syntax
- PMCFG (Parallel Multiple Context-Free Grammars), used for concrete syntaxes

A PGF file always contains one abstract syntax and, for each targeted language, one concrete syntax. It is a single, indivisible file, thereby similar to a Java `.jar` file or to a statically linked executable binary.

The following examples shows how GF code gets reduced in compilation to PGF. Let us first look at a set of algebraic data types that model the forms for German adjectives, nouns, and determiners. The hierarchic type `AForm` shows that only the attributive forms make distinctions in gender, number, and case, not the predicative forms. The hierarchic type `GNumber` captures the fact that gender distinctions are made only in the singular. The comments on the right show how in PGF all these types are converted to initial segments of integers:

```
param
  Number     = Sg | Pl ;               -- {0,1}
  Gender     = Masc | Fem | Neut ;     -- {0,1,2}
  GNumber    = GSg Gender | GPl ;      -- {0,1,2,3}
  Declension = Strong | Weak ;         -- {0,1}
  Case       = Nom | Acc | Dat | Gen ; -- {0,1,2,3}
  AForm      = APred | AAttr Declension GNumber Case ; -- {0,1,2,...,32}
```

The simplification improves run-time efficiency, since it replaces pattern matching by positional lookup. But it also means a loss of type distinctions: if it was applied at compile time, a programmer would not notice when confusing the neuter gender and the dative case, because both receive number 2.

Here is a GF definition of the regular adjective declension. It uses regular expression pattern matching to capture linguistic generalizations and to define the 33 adjective forms by 13 cases. Even more of the cases could be merged, maybe with a loss of clarity.

```
oper
  regA : Str -> {s : AForm => Str} = \s -> {
    s = \\a => s + case a of {
      APred => [] ;
      AAttr Strong (GSg Masc) Nom => "er" ;
      AAttr Strong (GSg Masc) Acc => "en" ;
      AAttr Strong (GSg (Masc | Neut)) Dat => "em" ;
      AAttr Strong (GSg (Masc | Neut)) Gen => "es" ;
      AAttr Strong (GSg Neut) (Nom | Acc)  => "es" ;
      AAttr Strong (GSg Fem | GPl) (Nom | Acc) => "e" ;
      AAttr Strong (GSg Fem) (Dat | Gen)=> "er" ;
      AAttr Strong GPl  Dat => "en" ;
      AAttr Strong GPl  Gen => "er" ;
      AAttr Weak   (GSg _) Nom => "e" ;
      AAttr Weak   (GSg (Fem | Neut)) Acc => "e" ;
      _ => "en"
      }
    } ;
```

Now, this is an auxiliary operation, which is not shown in PGF at all. But it generates the forms of all those adjectives whose linearization is actually implemented by using it. Hence if we have

```
  fun Klein : A ;  -- "small"
  lin Klein = regA "klein" ;
```

we get the following "inflection table" in PGF:

```
  F9 := (S23,
    S24,S25,S26,S27,S28,S28,S24,S24,S27,S27,S26,S27,S28,S28,S25,S24,
    S28,S25,S25,S25,S28,S28,S25,S25,S28,S28,S25,S25,S25,S25,S25,S25)
  S23 := "klein"
  S24 := "kleiner"
  S25 := "kleinen"
  S26 := "kleinem"
  S27 := "kleines"
  S28 := "kleine"
```

In GF and PGF, inflection tables are generalized from words to complex phrases. The following definitions show the types corresponding to noun phrases, adjectives, and nouns:

```
lincat
  NP = {s : Case => Str} ;
  A  = {s : AForm => Str} ;
  N  = {s : Number => Case => Str ; g : Gender} ;
```

We can now write a rule that forms a definite singular noun phrase from an adjective and a noun. Its concrete syntax is a complex interplay between parameters:

```
fun DefModN : A -> N -> NP ;
lin DefModN adj noun = {
      s = \\c => let ng = GSg noun.g in
          defArt ng c ++ adj.s ! AAttr Weak ng c ++ noun.s ! Sg ! c
      } ;
```

The PGF representation of the concrete syntax has three "rules", one for each gender the noun might have. Each rule has four "branches", one for each case. Each branch has a proper form of the article, and two pairs that indicate what arguments follow (0 for adjective, 1 for noun) and in what forms (as shown for the parameter types above).

```
F3 := (S1,S4,S7,S10) [DefModN]
F4 := (S2,S5,S8,S11) [DefModN]
F5 := (S3,S6,S9,S12) [DefModN]
S1 := "der" <0,17> <1,0>
S2 := "die" <0,21> <1,0>
S3 := "das" <0,25> <1,0>
S4 := "den" <0,18> <1,1>
S5 := "die" <0,22> <1,1>
S6 := "das" <0,26> <1,1>
S7 := "dem" <0,19> <1,2>
S8 := "der" <0,23> <1,2>
S9 := "dem" <0,27> <1,2>
S10 := "des" <0,20> <1,3>
S11 := "der" <0,24> <1,3>
S12 := "des" <0,28> <1,3>
```

2.1.2 Optimized PGF

Now, assume the grammar has only neuter nouns. Then it is possible to optimize the PGF by **dead-code elimination**, which removes unreachable rules and words. The code for `Klein` and `DefModN` then shrinks to

```
F3 := (S1,S1,S2,S2) [Klein]
F5 := (S5,S6,S7,S8) [DefModN]
S1 := "kleine"
S2 := "kleinen"
S5 := "das" <0,0> <1,0>
S6 := "das" <0,1> <1,1>
S7 := "dem" <0,2> <1,2>
S8 := "des" <0,3> <1,3>
```

which is approximately one third of the previous code size. Dead-code elimination is one of the techniques that make it feasible to use rich, general-purpose resource grammars for implementing small, light-weight application grammars.

2.2 Optional file formats

In addition to the standard `.gf` source files, the compiler recognizes the following input formats:

- `file.gfe`: input for example-based grammar writing
- `multifile.gfm`: multiple GF source modules in one file, including simplified lexicon format
- `langfile.cf`: context-free (BNF) source file for one language (abstract + concrete)

- `langfile.ebnf`: Extended BNF source file for one language (abstract + concrete)

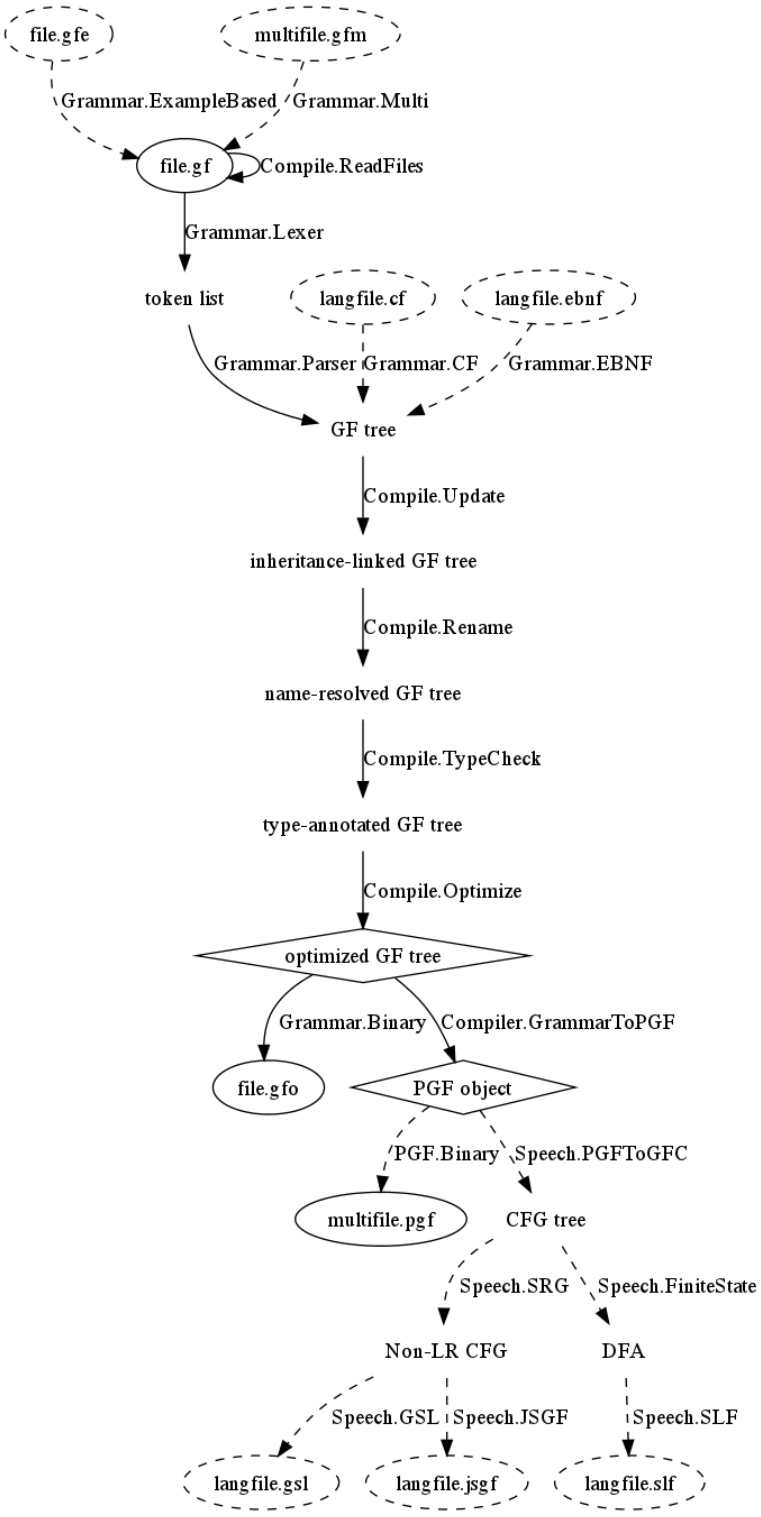
In addition to the standard `.gfo` and `.pgf` output, the compiler can be made to generate code in the following output formats:

- `langfile.gsl`: speech recognition grammar in GSL (Nuance) format
- `langfile.jsgf`: speech recognition grammar in JSGF format
- `langfile.slf`: speech recognition grammar in SLF format
- `langfile.dot`: finite automaton in graphviz format
- `multifile.js`: JavaScript (whole grammar)
- `file.hs`: Haskell (abstract syntax)
- `file.mod`: LambdaProlog (abstract syntax)

3 The compilation process

3.1 The compiler work flow

The following diagram shows the compiler phases: what happens when the command `gf` is invoked on a set of files, or the `import` command is used inside the GF shell.



Explanations for the diagram:

- **ellipse**: a **file** involved.
  - prefix `file`: one GF module
  - prefix `multifile`: complete multilingual grammar with any number of modules and languages
  - prefix `langfile`: abstract+concrete syntax for one language with any number of modules
- **solid ellipse**: major file format, involved by default
  - `file.gf`: GF source module
  - `file.gfo`: GF object file
  - `multifile.pgf`: **Portable Grammar Format** for multilingual grammar at runtime
- **dashed ellipse**: optional file format, such as (see more formats above)
  - `file.gfe`: input for example-based grammar writing
  - `multifile.gfm`: multiple GF source modules in one file, including simplified lexicon format
  - `langfile.cf`: context-free (BNF) source file for one language (abstract + concrete)
  - `langfile.gsl`: speech recognition grammar in GSL (Nuance) format
- **diamond**: run-time datastructure used for processing language
  - **optimized GF tree**: source module with e.g. interpreted `oper`s
  - **PGF object**: the multilingual grammar used for parsing, generation, and translation
- **plain text**: intermediate datastructures with no external output
- **arrow labels**: modules defining each compilation phase

3.2 Compiler modes and flags

The compiler can be invoked in three modes:

- **interactive mode**: from the `gf` shell, the `import` command
- **batch mode**: from the OS shell, the `gf` command
- **server mode**: from a wrapper program, e.g. a web application calling the compiler server

There are three main levels of verbosity:

- **normal** (no flag): show what modules are compiled, and all warnings and error messages
- **silent** (flag `-s`): show fatal error messages only

- **verbose** (flag `-v`): show what files are read or compiled, all warnings and error messages, and each function when optimized and linked

The silent mode is useful in Makefiles aimed for source distribution of code. The verbose mode is useful in grammar development to trace errors that are otherwise difficult to find. Displaying the functions being optimized and linked is useful if the compilation takes a long time: it shows where the hot spots are.

The verbosity flags are available in both the interactive mode and the batch mode. The batch mode has many more flags than this; the actual ones are shown by the flag `-help`,

```
$ gf -help
```

The input formats are recognized from the file suffix. The output formats are generated in two different ways:

- interactive mode: use the command `print_grammar -printer=<FORMAT>`
- batch mode: use the flag `-output-format=<FORMAT>`

For help on the `-output-format` flags, consult

```
> help print_grammar
```

in the shell.

The batch compiler can dump the outcome of many compilation phases. For instance,

```
$ gf --dump-tc              -- type checked source
$ gf --dump-canon           -- optimized source (textual gfo)
```

3.3 The principal phases

As shown by the work-flow diagram, the compilation follows rather standard phases. Each phase can fail with errors characteristic to it.

3.3.1 Dependency analysis

This phase decides what files to read and what to compile. It performs the following steps:

1. Reads the header of the explicitly called `.gf` file.
2. Reads the headers of all files included in the header, recursively.
3. Build a dependency graph.
4. Decide what modules to compile (from `.gf`), what to read from object files (`.gfo`).

Object files are used whenever the `.gf` source file is missing - for instance, when binary distributions of libraries are used.

This phase fails if some of the files in the dependency graph is missing. The files are sought from three places.

- the directory of the explicitly called file
- the directories given in a `-path` flag
- the `-path` flag directories prependend by the value of the environment variable `GF_LIB_PATH`

Consult the GF Reference Manual for more information about `-path` and `GF_LIB_PATH`.

3.3.2 Lexing

This phase performs a lexical analysis of the source code by using a finite-state automaton. Typical reasons of failure are

- unterminated string, e.g. `"foo`
- unterminated comment, i.e. opening {- without a closing -}
- unrecognized special character outside strings and comments, e.g. `&`
- non-ascii letter in an identifier

Consult the GF Reference Manual for a specification of the lexical structure.

3.3.3 Parsing

This phase performs a syntactic analysis of the source code by using an LALR(1) parser. Typical reasons of failure are

- unmatching parentheses
- usage of key words as identifiers (e.g. `def`, `in`)
- missing semicolons after judgements
- judgement of a different form than expected from the keyword

Consult the GF Reference Manual for a specification of the syntactic structure.

3.3.4 Updating

This phase builds links between a new module and the previously compiled ones, e.g. links to inherited constants. Typical failures are

- inheritance of modules of wrong types, e.g. a concrete from an abstract
- inheritance of the same constant from several sources
- incomplete instance of an interface

Consult the GF Reference Manual for the compatibility of the module types and the inheritance rules.

3.3.5 Renaming

This phase resolves the names of all unqualified constants and prepends their module names. It simultaneously tells local variables apart from constants: if an identifier is bound as a variable, it is not interpreted as a constant of the same name. Typical failures are

- constant not found (fatal)
- the same constant found from different opened modules (warning only; can result in type error later)

Consult the GF Reference Manual for the name resolution and the use of qualified names.

3.3.6 Type checking

This phase checks the consistency of the module in terms of type signatures, or tries to infer types if not given. It also performs the resolution of overloaded operations and annotates data structures for the purpose of optimization. Typical failures are

- mismatch between expected and inferred type
- no overload instance found

Consult the GF Reference Manual for the type system of GF.

3.3.7 Optimization

This phase performs partial evaluation of `lin` rules and (optionally) `oper` rules. This includes eta expansion of `lin` terms (functions, records, tables), and the reduction of applications, projections, and selections. Milner's principle, "well-typed programs cannot go wrong" prevents most errors at this phase, but some failures can still appear:

- incomplete patterns in a table
- completely overshadowed patterns in a table (warning only)
- token-internal operations on run-time variables (in particular, gluing (`+`))
- exceptions raised by `Predef.error` (mostly in pattern matching over strings)

The output of this phase for each module is written into a `.gfo` file.
Consult the GF Reference Manual for pattern matching and other evaluation rules.

3.3.8 Linking

This phase links the previously compiled modules together into a multilingual `.pgf` grammar. If the compilation is performed in the GF shell, the grammar is not written into a file, but can be saved later with

```
> print_grammar -pgf
```

If the compiler is run in batch mode, no linking is performed by default, which greatly speeds up the compilation of libraries distributed as `.gfo` only. But the batch command with `-make` option,

```
$ gf -make
```

does perform linking and writes a `.pgf` file, which is by default named after the abstract syntax of the multilingual grammar.

The linking phase is expected not to fail. However, there are two situations where it may:

- bugs in the GF compiler - please report
- running out of memory - increase stack size with e.g.

```
$ gf <OPTIONS> <INPUT-FILES> +RTS -K100M
```

4 Grammar diagnostics in the GF shell

4.1 Inspecting grammars

4.1.1 Getting information about constants in scope

The following examples are obtained in the shell state created with

```
> import alltenses/LangEng.gfo
```

Show all `cat` categories in scope:

```
> print_grammar -cats
A A2 AP AdA AdN AdV Adv Ant CAdv CN Card Cl ClSlash Comp Conj Det
Dig Digit Digits Float IAdv IComp IDet IP IQuant Imp Int Interj
...
```

Show all `fun` functions in scope:

```
> print_grammar -funs
AAnter : Ant ;
ASimul : Ant ;
AdAP : AdA -> AP -> AP ;
AdAdv : AdA -> Adv -> Adv ;
AdNum : AdN -> Card -> Card ;
AdVVP : AdV -> VP -> VP ;
...
```

Show all `fun` functions with value category `Cl`:

```
> abstract_info Cl
cat Cl ;
fun ImpersCl : VP -> Cl ;
fun GenericCl : VP -> Cl ;
fun CleftNP : NP -> RS -> Cl ;
fun CleftAdv : Adv -> S -> Cl ;
fun PredVP : NP -> VP -> Cl ;
...
```

The following examples need a resource module imported with the `-retain` option:

```
> import -retain alltenses/SyntaxEng.gfo
```

Show all operations in scope:

```
> show_operations
Art : Type
ComplV2 : V2 -> NP -> VP
ComplV2A : V2A -> NP -> AP -> VP
ComplV3 : V3 -> NP -> NP -> VP
...
```

Show all operations that have value type `NP`:

```
> show_operations NP
he_NP : NP
i_NP : NP
it_NP : NP
mkNP : Quant -> N -> NP
mkNP : Quant -> CN -> NP
mkNP : Quant -> Num -> CN -> NP
...
```

4.1.2 Inspecting the library

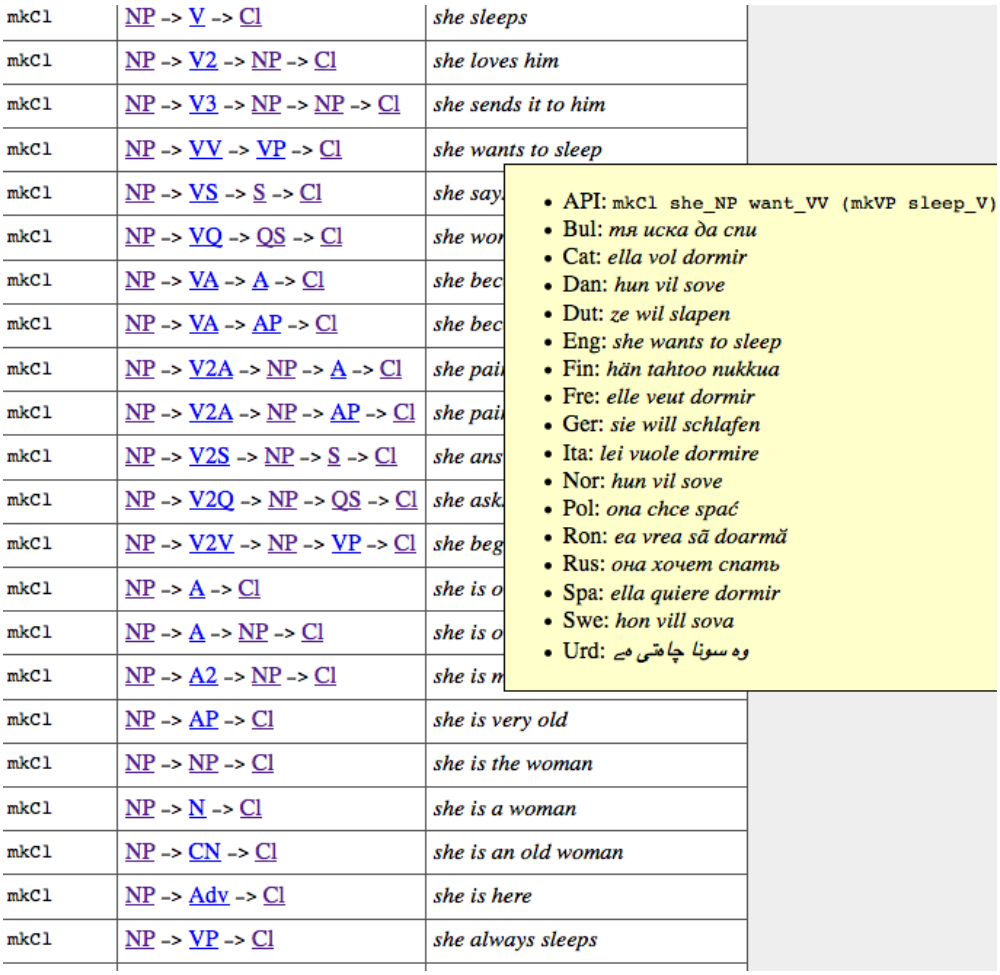
The Resource Grammar Library is an essential part of the GF grammar development system. It can be inspected with all commands shown in the previous section. In particular, `show_operations` can be used for each language with

- `Syntax`L, to see the syntactic constructors and structural words
- `Paradigms`L, to see the morphological paradigms
- `Try`L, to see the both syntax and paradigms

A document with examples and explanations can be found in the Resource Grammar Synopsis,

- http://www.grammaticalframework.org/lib/doc/synopsis.html

The Synopsis has automatically generated examples for all Resource Grammar languages. They are visible when hovering the mouse over the English examples, as shown in the following picture:



mkCl	NP -> V -> Cl	*she sleeps*
mkCl	NP -> V2 -> NP -> Cl	*she loves him*
mkCl	NP -> V3 -> NP -> NP -> Cl	*she sends it to him*
mkCl	NP -> VV -> VP -> Cl	*she wants to sleep*
mkCl	NP -> VS -> S -> Cl	*she say*
mkCl	NP -> VQ -> QS -> Cl	*she wor*
mkCl	NP -> VA -> A -> Cl	*she bec*
mkCl	NP -> VA -> AP -> Cl	*she bec*
mkCl	NP -> V2A -> NP -> A -> Cl	*she pai*
mkCl	NP -> V2A -> NP -> AP -> Cl	*she pai*
mkCl	NP -> V2S -> NP -> S -> Cl	*she ans*
mkCl	NP -> V2Q -> NP -> QS -> Cl	*she ask*
mkCl	NP -> V2V -> NP -> VP -> Cl	*she beg*
mkCl	NP -> A -> Cl	*she is o*
mkCl	NP -> A -> NP -> Cl	*she is o*
mkCl	NP -> A2 -> NP -> Cl	*she is m*
mkCl	NP -> AP -> Cl	*she is very old*
mkCl	NP -> NP -> Cl	*she is the woman*
mkCl	NP -> N -> Cl	*she is a woman*
mkCl	NP -> CN -> Cl	*she is an old woman*
mkCl	NP -> Adv -> Cl	*she is here*
mkCl	NP -> VP -> Cl	*she always sleeps*

- API: `mkCl she_NP want_VV (mkVP sleep_V)`
- Bul: *тя иска да спи*
- Cat: *ella vol dormir*
- Dan: *hun vil sove*
- Dut: *ze wil slapen*
- Eng: *she wants to sleep*
- Fin: *hän tahtoo nukkua*
- Fre: *elle veut dormir*
- Ger: *sie will schlafen*
- Ita: *lei vuole dormire*
- Nor: *hun vil sove*
- Pol: *ona chce spać*
- Ron: *ea vrea să doarmă*
- Rus: *она хочет спать*
- Spa: *ella quiere dormir*
- Swe: *hon vill sova*
- Urd: وہ سونا چاہتی ہے

4.1.3 Visualizing module dependencies

Visualizing module dependencies needs a source grammar and hence the `-retain` option:

```
> import -retain alltenses/LangEng.gfo
```

The dependency graph is generated into a temporary file:

```
> dependency_graph
wrote graph in file _gfdepgraph.dot
```

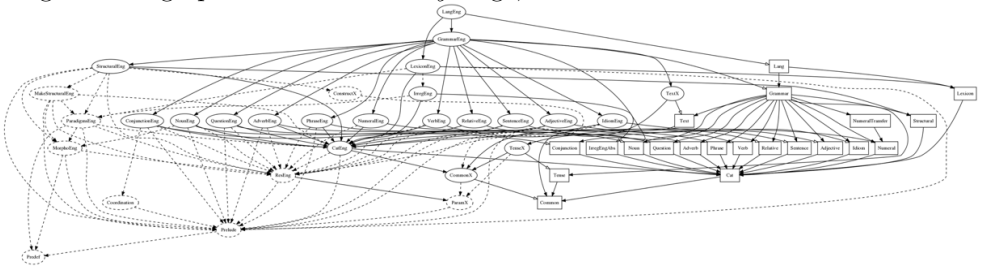
It can be viewed from the shell by first using a shell escape (`!`) to process it,

```
> ! dot -Tpng _gfdepgraph.dot >depLangEng.png
```

and then opening it with a PNG file viewer available in the operating system (e.g. `open` in Mac):

```
> ! open depLangEng.png
```

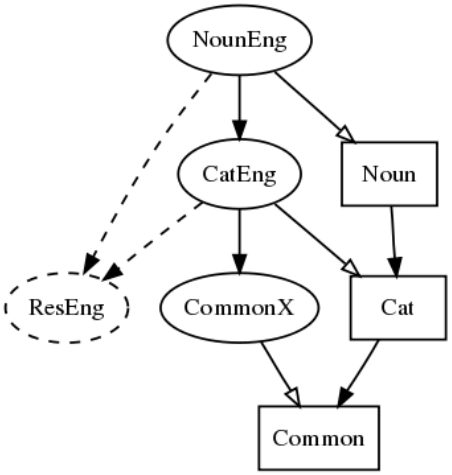
The generated graph in this case if *very* large, but we can still see it:



Smaller graphs can be generated by specifying which modules only to show (or sets of modules by `prefix*`). For instance:

```
> dependency_graph -only=Cat*,Common*,ResEng,Noun*
```

Here is the resulting graph:



4.1.4 Printing different views of the grammar

In addition to generating files, the `print_grammar` command has many options that are not independent file formats, but can give useful information for debugging and other purposes. These commands work again on PGF, so we assume

```
> import alltenses/LangEng.gfo
```

Some example commands are:
Produce a text dump of the PGF:

```
> print_grammar
-- 17033 lines of code
```

Show all words (tokens) of the grammar:

```
> print_grammar -words
! , - . 0 0's 0th 0th's 1 1's 1st 1st's 2 2's 2nd 2nd's 3 3's 3rd 3rd's
4 4's 4th 4th's 5 5's 5th 5th's 6 6's 6th 6th's 7 7's 7th 7th's 8 8's
8th 8th's 9 9's 9th 9th's ? English I John John's Paris Paris's a able
about above add added adding adds after airplane airplane's
-- 1813 words
```

Print a **full-form lexicon** with a complete list of analyses of all word forms:

```
> print_grammar -fullform
airplane
airplane_N : s Sg Nom
...
```

Show the missing linearization rules in each language:

```
> print_grammar -missing
LangEng : AdVVPSlash AdvVPSlash dconcat digits2num dn dn10 dn100 dn1000
dn1000000a dn1000000b dn1000000c nd nd10 nd100 nd1000 nd1000000 num2digits
```

BTW, how did we get the line and word counts? By a **system pipe**:

```
> print_grammar | ? wc
17033   97589 1137167
```

The pipes can also be used for filtering other commands:

```
> pg -fullform | ? grep "air"
been
beer
beer_N : s Sg Nom
...
```

4.1.5 Visualizing grammars as automata

Load a small grammar:

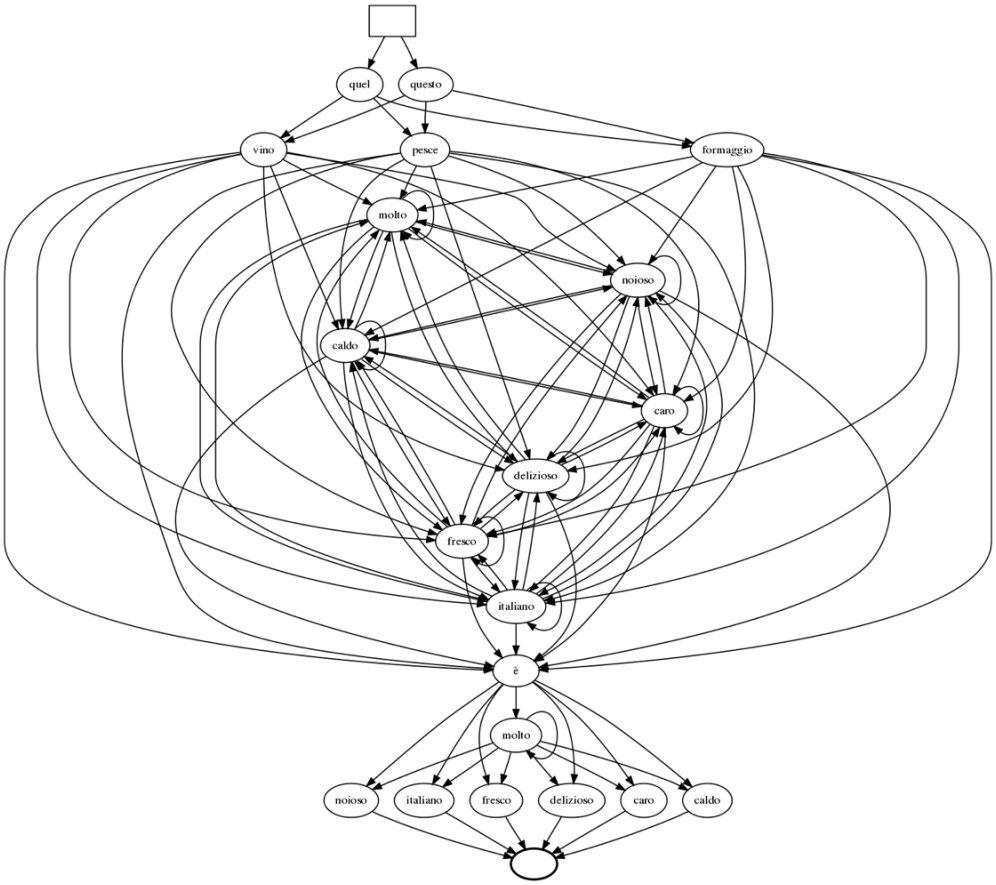
```
> i GF/examples/tutorial/food/FoodIta.gf
```

Print a finite-state automaton into a graphviz file, convert to PNG, and view:

```
> pg -printer=fa | wf -file=faFoodIta.dot

> ! dot -Tpng faFoodIta.dot >faFoodsIta.png

> ! open faFoodsIta.png
```



4.2 Testing grammars

4.2.1 Parsing, linearization, and generation

The main functionalities of GF are also among the main tools for grammar development. Parsing, linearization, and exhaustive and random generation can be combined with numerous options and combined in pipes to form the basis of applications and test scripts. They work on the PGF level, independently of source grammars. Here are some typical examples of their use.

First import a multilingual grammar - either a `.pgf` file or a set of `.gf`/`.gfo` files. We use the top-level resource grammar `Lang`.

```
> import alltenses/LangEng.gfo alltenses/LangSwe.gfo
```

Then **parse** a string in English in the category `Cl` (clause):

```
> parse -lang=Eng -cat=Cl "John sleeps"
PredVP (UsePN john_PN) (UseV sleep_V)
```

Linearize the result into both languages:

```
> linearize PredVP (UsePN john_PN) (UseV sleep_V)
John sleeps
Johan sover
```

Or **translate** by **piping** parsing into linearization:

```
> parse -lang=Eng -cat=Cl "John drinks wine" | linearize -lang=Swe
Johan dricker vin
```

Linearize to the **table** of all tenses and polarities:

```
> parse -lang=Eng -cat=Cl "John drinks wine" | linearize -lang=Eng -table
s Pres Simul CPos ODir : John drinks wine
s Pres Simul CPos OQuest : does John drink wine
s Pres Simul (CNeg True) ODir : John doesn't drink wine
s Pres Simul (CNeg True) OQuest : doesn't John drink wine
s Pres Simul (CNeg False) ODir : John does not drink wine
s Pres Simul (CNeg False) OQuest : does John not drink wine
s Pres Anter CPos ODir : John has drunk wine
...
```

Or just linearize to the **list** of all forms:

```
> linearize -list -lang=Swe drink_V2
dricker, dricks, drack, dracks, drick, dricks, dricka, drickas, druckit,
druckits, drucken, druckens, drucket, druckets, druckna, drucknas, druckna,
drucknas, druckna, drucknas
```

Generate randomly a number of trees in a given category:

```
> generate_random -cat=Cl -number=4
ImpersCl (ComplVA become_VA (AdvAP (PositA hot_A) far_Adv))
CleftAdv there7to_Adv (RelS (UseCl (TTAnt TPast ASimul) PPos ...
PredSCVP (EmbedQS (UseQCl (TTAnt TFut ASimul) PPos (ExistIP whatSg_IP))) ...
PredVP (DetCN few_Det (AdvCN (UseN restaurant_N) somewhere_Adv)) ...
```

Generate exhaustively all trees in a given category up to a given depth:

```
> generate_trees -cat=CN -depth=2 | ? wc
-- 402134 trees
```

Both kinds of generation can be guided by **tree patterns**, which contain **metavariables**:

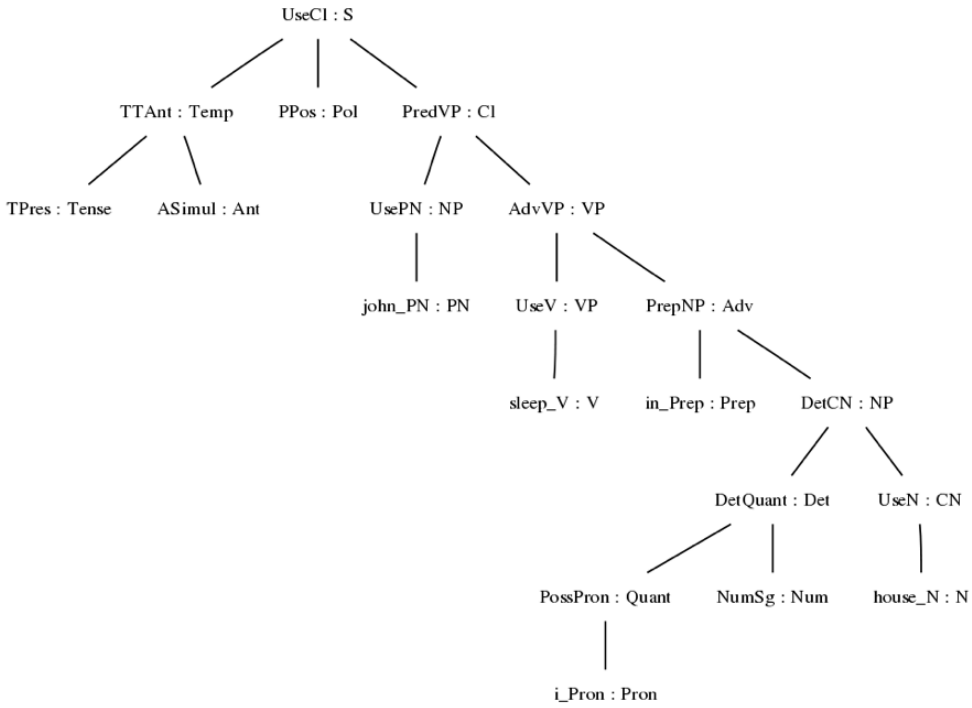
```
> generate_random -cat=Cl -number=8 (PredVP (UsePron ?) (UseV ?)) | l
they lie
you float
you stand
you spit
I swell
she turns
it lives
he swims
```

4.2.2 Visualizing trees and word alignments

The visualization tools are useful when developing and documenting grammars. They all use Graphviz, and can both save `.dot` files and open graphics windows (with `.png` files) directly from the shell. The direct view is produced by the `-view` flag, calling the operating system's PNG viewer.

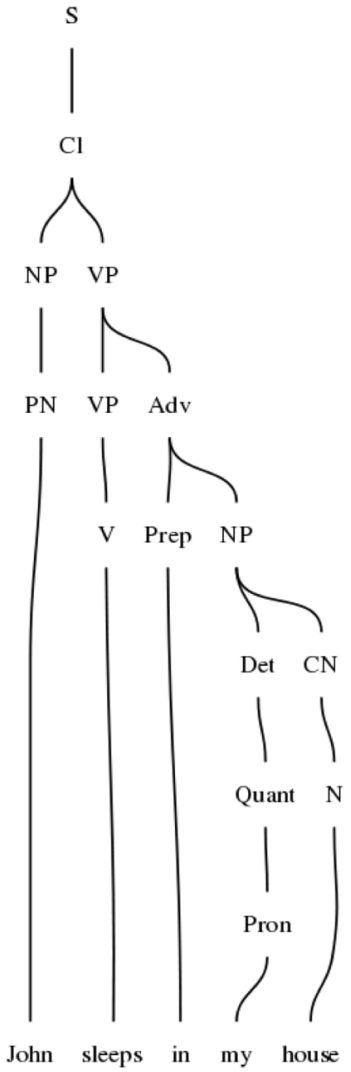
Visualize abstract syntax tree:

```
parse -lang=Eng -cat=S "John sleeps in my house" | visualize_tree -view=open
```



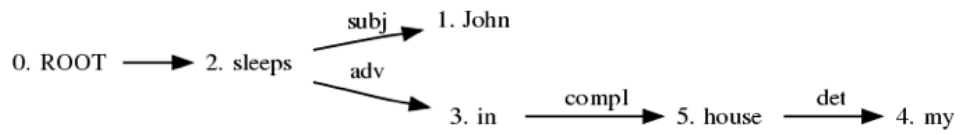
Visualize parse tree ("concrete syntax tree"):

```
parse -lang=Eng -cat=S "John sleeps in my house" | visualize_parse -view=open
```



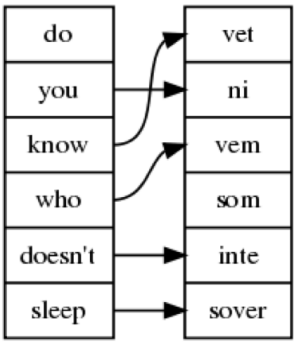
Visualize dependency tree, with labels configured in file `dep.labels`:

```
parse -lang=Eng -cat=S "John sleeps in my house" | visualize_dependency -file=dep.labels -view=open
```



Visualize word aligmnent (actually, phrase alignment):

```
parse -lang=Eng -cat=QS "do you know who doesn't sleep" | align_words -view=open
```



4.2.3 Testing auxiliary operations

The following examples need a resource module imported with the `-retain` option. We import `TryEng`, which contains both syntax and paradigms.

```
> import -retain alltenses/TryEng.gfo
```

The command `compute_concrete` evaluates source GF expressions in an interpreted mode. It can be used for testing morphological paradigms:

```
> compute_concrete mkV "apply"
{s : ResEng.VForm => Str
 = table ResEng.VForm ["apply"; "applies"; "applied"; "applying";
                       "applied"];
 isRefl : Prelude.Bool = Prelude.False; lock_V : {} = <>}
```

A more compact view is obtained by the `-list` option:

```
> compute_concrete -list mkV "apply"
apply, applies, applied, applying, applied, Prelude.False
```

One can also try combinations of syntactic and morphological functions (the flag `-all` is like `-list`, but on separate lines; see `help compute_concrete`):

```
> compute_concrete -all mkCl (mkNP this_Det (mkN "method")) (mkV2
   (mkV "apply") to_Prep) (mkNP every_Det (mkCN (mkA "possible") (mkN "input")))
this method applies to every possible input
does this method apply to every possible input
this method doesn't apply to every possible input
doesn't this method apply to every possible input
...
```

Notice that the morphological operations in the library are sets of **smart paradigms**, where the most regular one has just one argument, and adding arguments makes them applicable to less regular words. Thus,

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

(plus some others). Thus one can define a word by trial and error starting with the simplest paradigm and adding arguments if needed:

```
> compute_concrete -list mkV "write"
write, writes, writed, writing, writed, Prelude.False

> compute_concrete -list mkV "write" "wrote" "written"
write, writes, written, writing, wrote, Prelude.False
```

This can serve as a basis for a tool that uses a form with slots as an interface to smart paradigms.

4.2.4 Ambiguity testing

4.2.5 Treebanks and regression testing

5 Converting other formats to GF

5.1 Example-based grammars

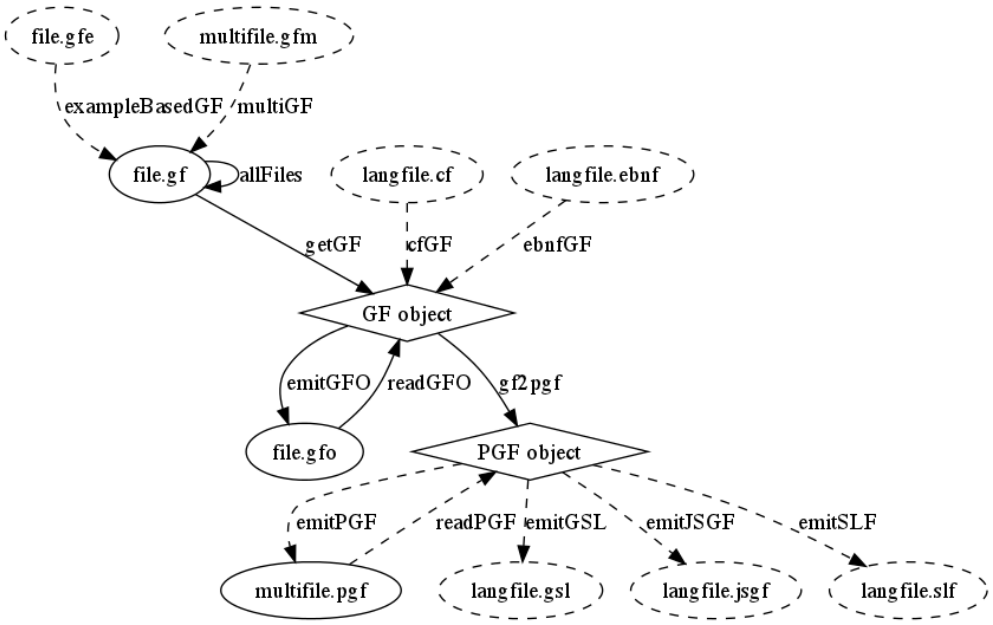
5.2 Lexical resources

5.3 Ontologies

6 The Grammar Compiler Haskell API

The **Grammar Compiler Haskell API** is a module that exposes the most important functions and types used in compiling GF grammars. It makes them available for programmers who want to create alternative accesses to grammar compilation, besides the standard GF shell and batch compiler.

If we eliminate the intermediate datastructures from the compiler workflow figure, we get a simplified figure that shows the functionalities available to the user of the grammar compiler. The module names have now been changed to descriptive names of the functionalities.



The descriptive names (diagram above) are at used as function names in the Haskell API. They operate on the files and datastructures mentioned in the nodes of the diagram. They may also take auxiliary arguments, in particular, for an appropriate compilation environment and for a set of options.

The following types and functions are expected in the Haskell API:

```
module GF.GrammarCompiler where

type GF         -- source grammar
type PGF        -- PGF grammar
type Module     -- any source module
type Judgement  -- any definition
type Term       -- term in concrete syntax
type Type       -- type in concrete syntax
type Name       -- of module, function, category,...

type Check a    -- error/state monad with warnings etc

exBasedGF :: FilePath -> IO GF
multiGF   :: FilePath -> IO GF
getGF     :: FilePath -> IO GF
cfGF      :: FilePath -> IO GF
ebnfGF    :: FilePath -> IO GF
emitGFO   :: GF -> IO ()
readGFO   :: FilePath -> IO GF
gf2pgf    :: GF -> PGF
emitPGF   :: PGF -> IO ()
readPGF   :: FilePath -> IO PGF
emitJSGF  :: PGF -> IO ()
emitSLF   :: PGF -> IO ()

checkTerm      :: GF -> Module -> Term -> Type -> Check Term
checkJudgement :: GF -> Module -> Judgement -> Check Judgement
```

```
evalTerm       :: GF -> Term -> Check Term

showOpers      :: GF -> Type -> [(Name,Type)]
                  -- opers with desired value type

addJudgement   :: GF -> Module -> Judgement -> Check Module
addModule      :: GF -> Module -> Check GF

addConstruct   :: GF -> Judgement -> [Judgement] -> Check GF
                  -- to abstract and all concretes
```

Types and functions working on pure PGF are available in the run-time API, and can be directly used for grammar-testing purposes. Here are some such functions; the full set is described in the PGF API documentation.

```
module PGF where

type PGF
type Tree
type Cat
type Fun
type Language

parse     :: PGF -> Language -> Cat -> String -> [Tree]
linearize :: PGF -> Language -> Tree -> String
genRandom :: PGF -> Cat -> Tree
genAll    :: PGF -> Cat -> [Tree]

languages  :: PGF -> [Language]
categories :: PGF -> [Category]
startCat   :: PGF -> Category
```

7 A web-based grammar IDE

Traditionally, GF grammars are created in a text editor and tested in the GF shell. This is a simple and effective environment for the experienced grammar developer. To better support less experienced grammar developers, one of the goals of the MOLTO project is to create an IDE (Integrated Development Environment) for GF grammar development, and an appropriate GF compiler API to support the IDE.

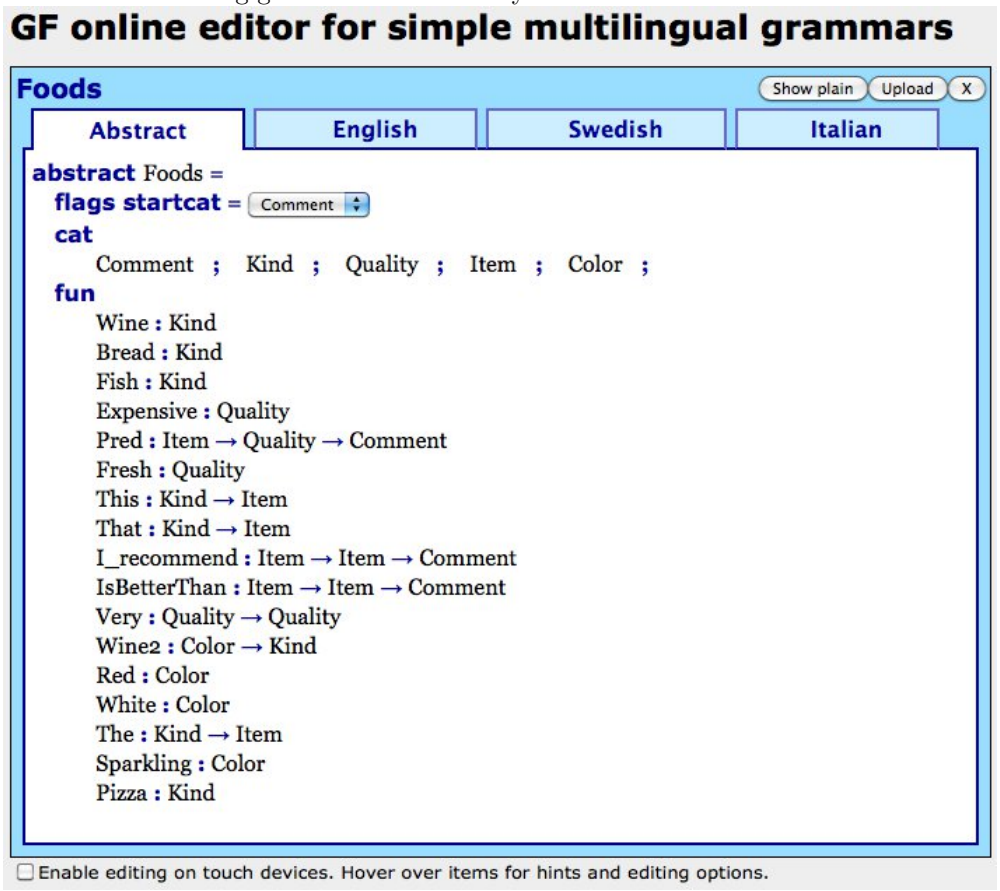
7.1 A Prototype grammar IDE

To guide the development of a suitable compiler API to support grammar IDEs, we have created a prototype web-based grammar editor: the *GF online editor for simple multilingual grammars*. It is available online, and is currently usable for creating small multilingual grammars. Since it doesn't require any downloads or use of command shells, it makes it easier for novice grammar developers to get started. All that is needed is a reasonably modern web browser. Even Android and iOS devices can be used.

The editor guides the grammar author by showing a skeleton grammar file and hinting how the parts should be filled in. When a new part is added to the grammar, it is immediately checked for errors.

Editing operations are accessed by clicking on editing symbols embedded in the grammar display: + = Add an item, = Delete an item, % = Edit an item. These are revealed when hovering over items. On touch devices, hovering is in some cases simulated by tapping, but there is also a button at the bottom of the display, "Enable editing on touch devices", that reveals all editing controls.

The editor runs entirely in the web browser, so once you have opened the web page, you can continue editing grammars even while you are offline.



7.1.1 Limitations

At the moment, the editor supports only a small subset of the GF grammar notation. Proper error checking is done for abstract syntax, but not (yet) for concrete syntax.

The grammars created with this editor always consists of one file for the abstract syntax, and one file for each concrete syntax.

7.1.2 Abstract syntax

The supported abstract syntax corresponds to context-free grammars (no dependent types). The definition of an abstract syntax is limited to

- a list of *category names*, *Cat_1* ; ... ; *Cat_n*,
- a list of *functions* of the form *Fun* : *Cat_1* -> ... -> *Cat_n*,

- and a *start category*.

Available editing operations:

- Categories can be added, removed and renamed. When renaming a category, occurrences of it in function types will be updated accordingly.
- Functions can be added, removed and edited. Concrete syntaxes are updated to reflect changes.
- Functions can be reordered using drag-and-drop.

Error checks:

- Syntactically incorrect function definitions are refused.
- Semantic problem such as duplicated definitions or references to undefined categories, are highlighted.

7.1.3 Concrete syntax

At the moment, the concrete syntax for a language L is limited to

- opening the Resource Grammar Library modules `Syntax`L and `Paradigms`L,
- *linearization types* for the categories in the abstract syntax,
- *linearizations* for the functions in the abstract syntax,
- *parameter type definitions*, $P = C\_1 \mid ... \mid C\_n$,
- and *operation definitions*, $op = expr$.

Available editing operations:

- The LHSs of the linearization types and linearizations are determined by the abstract syntax and do not need to be entered manually. The RHSs can be edited.
- Parameter types can be added, removed and edited.
- Operation definitons can be added, removed and edited.
- Definitions can be reordered (using drag-and-drop)

Also,

- When a new concrete syntax is added to the grammar, a copy of the currently open concrete syntax is created, since copying and modifying is usually easier than creating something new from scratch. (If the abstract syntax is currently open, the new conrete syntax will start out empty.)

Error checks:

- The RHSs in the concrete syntax are not checked for errors. Arbitrary strings can be entered.

7.1.4 Testing grammars

By pressing the **Upload** button, a grammar can be uploaded to the GF server. It will then be compiled with GF, and any errors not detected by the editor will be reported. If the grammar is free from errors, the user can test the grammar by clicking on links to the online GF shell, the Minibar or the Translation Quiz.

7.1.5 Future work

This prototype gives an idea of how a web based GF grammar editor could work. While this editor is implemented in JavaScript and runs entirely in the web browser, we do not expect to create a full implementation of GF that runs in the web browser, but let the editor communicate with a server running GF.

By developing a GF server with an appropriate API, it should be possible to extend the editor to support a larger fragment of GF, to do proper error checking and make more of the existing GF shell functionality accessible directly from the editor.

Grammars are currently stored locally in the browser, but a future version could allow grammars to be stored "in the cloud", allowing the same grammars to be accessed from multiple devices.

7.2 Web services for a grammar IDE

To develop the above outlined web-based grammar IDE further, or implement other grammar IDEs, a web service interface to the GF compiler API will be useful. The basic services provided by the GF server is outlined in the form of a Haskell API below. The exact syntax for the HTTP server requests and responses remain to be defined. It should be straight forward to extended this to cover more of the rich GF functionality currently available only through the GF shell, as illustrated in previous sections.

```
module GF.Service.API where

-- A stateful GF compiler service API

type SessionId  = Int

type Error      = ...     -- structured representation
type Warning    = ...     --
type ModuleName = String  -- grammar module name
type Name       = String  -- identifiers defined in grammar modules

type SourceCode = String  -- GF source code, input from applicaiton
type HTML       = String  -- for highlighted source code, output to application

type GFS        = IO      -- ? GF Session monad

--- Starting/ending sessions ------------------------------------------------

newSession :: GFS SessionId
endSession :: SessionId -> GFS ()

-- A string-based interface to GF shell functionality ---------------------

shell_command :: SessionId -> String -> GFS String
```

```
-- A structured interface for grammar editors ----------------------------------

importModule :: SessionId -> SourceCode -> GFS ([Error],[Warning])

addJudgements:: SessionId -> ModuleName -> SourceCode -> GFS ([Error],[Warning])
removeJudgement :: SessionId -> ModuleName -> Name -> GFS ([Error],[Warning])

availableModules :: SessionId -> GFS [ModuleName]
defined, exports, inScope :: SessionId -> ModuleName -> GFS [Name]

syntax_highlight :: SourceCode -> HTML
```