The GF Resource Grammar Library

Author: Aarne Ranta, Ali El Dada, and Janna Khegai

Last update: Tue Jun 20 17:04:47 2006

Contents

This document is about the GF Resource Grammar Library. It presuppose knowledge of GF and its module system, knowledge that can be acquired e.g. from the GF tutorial. We start with an introduction to the library, and proceed to details with the aim of covering all that one needs to know in order to use the library. How to write one's own resource grammar (i.e. to implement the API for a new language), is covered by a separate Resource-HOWTO document (available in the www address below).

The main part of the document (the API documentation) is generated from the actual GF code by using the `gfdoc` tool. This documentation is also available on-line in HTML format in

http://www.cs.chalmers.se/~aarne/GF/lib/resource-1.0/doc/.

1 Motivation

The GF Resource Grammar Library contains grammar rules for 10 languages (some more are under construction). Its purpose is to make these rules available for application programmers, who can thereby concentrate on the semantic and stylistic aspects of their grammars, without having to think about grammaticality. The targeted level of application grammarians is that of a skilled programmer with a practical knowledge of the target languages, but without theoretical knowledge about their grammars. Such a combination of skills is typical of programmers who want to localize software to new languages.

The current resource languages are

- Arabic
- Danish
- English
- Finnish
- French
- German
- Italian
- Norwegian
- Russian

- Spanish
- Swedish

The first three letters (`Ara` etc) are used in grammar module names. The Arabic implementation is still incomplete.

To give an example application, consider music playing devices. In the application, we may have a semantical category `Kind`, examples of `Kind`s being `Song` and `Artist`. In German, for instance, `Song` is linearized into the noun "Lied", but knowing this is not enough to make the application work, because the noun must be produced in both singular and plural, and in four different cases. By using the resource grammar library, it is enough to write

```
lin Song = reg2N "Lied" "Lieder" neuter
```

and the eight forms are correctly generated. The resource grammar library contains a complete set of inflectional paradigms (such as `reg2N` here), enabling the definition of any lexical items.

The resource grammar library is not only about inflectional paradigms - it also has syntax rules. The music player application might also want to modify songs with properties, such as "American", "old", "good". The German grammar for adjectival modifications is particularly complex, because adjectives have to agree in gender, number, and case, and also depend on what determiner is used ("ein Amerikanisches Lied" vs. "das Amerikanische Lied"). All this variation is taken care of by the resource grammar function

```
fun AdjCN : AP -> CN -> CN
```

(see the tables in the end of this document for the list of all resource grammar functions). The resource grammar implementation of the rule adding properties to kinds is

```
lin PropKind kind prop = AdjCN prop kind
```

given that

```
lincat Prop = AP
lincat Kind = CN
```

The resource library API is devided into language-specific and language-independet parts. To put it roughly,

- the lexicon API is language-specific
- the syntax API is language-independent

Thus, to render the above example in French instead of German, we need to pick a different linearization of `Song`,

```
lin Song = regGenN "chanson" feminine
```

But to linearize `PropKind`, we can use the very same rule as in German. The resource function `AdjCN` has different implementations in the two languages (e.g. a different word order in French), but the application programmer need not care about the difference.

1.1 A complete example

To summarize the example, and also give a template for a programmer to work on, here is the complete implementation of a small system with songs and properties. The abstract syntax defines a "domain ontology":

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

The concrete syntax is defined by a functor (parametrize module), independently of language, by opening two interfaces: the resource `Grammar` and an application lexicon.

```
incomplete concrete MusicI of Music = open Grammar, MusicLex in {
  lincat
    Kind = CN ;
    Property = AP ;
  lin
    PropKind k p = AdjCN p k ;
    Song = UseN song_N ;
```

```
    American = PositA american_A ;
  }
```

The application lexicon `MusicLex` has an abstract syntax that extends the resource category system `Cat`.

```
abstract MusicLex = Cat ** {
  fun
    song_N : N ;
    american_A : A ;
  }
```

Each language has its own concrete syntax, which opens the inflectional paradigms module for that language:

```
concrete MusicLexGer of MusicLex =
    CatGer ** open ParadigmsGer in {
  lin
    song_N = reg2N "Lied" "Lieder" neuter ;
    american_A = regA "Amerikanisch" ;
  }

concrete MusicLexFre of MusicLex =
    CatFre ** open ParadigmsFre in {
  lin
    song_N = regGenN "chanson" feminine ;
    american_A = regA "américain" ;
  }
```

The top-level `Music` grammars are obtained by instantiating the two interfaces of `MusicI`:

```
concrete MusicGer of Music = MusicI with
  (Grammar = GrammarGer),
  (MusicLex = MusicLexGer) ;

concrete MusicFre of Music = MusicI with
  (Grammar = GrammarFre),
  (MusicLex = MusicLexFre) ;
```

Both of these files can use the same `path`, defined as

```
--# -path=.:present:prelude
```

The `present` category contains the compiled resources, restricted to present tense; `alltenses` has the full resources.

To localize the music player system to a new language, all that is needed is two modules, one implementing `MusicLex` and the other instantiating `Music`. The latter is completely trivial, whereas the former one involves the choice of correct vocabulary and inflectional paradigms. For instance, Finnish is added as follows:

```
concrete MusicLexFin of MusicLex =
    CatFin ** open ParadigmsFin in {
  lin
    song_N = regN "kappale" ;
    american_A = regA "amerikkalainen" ;
  }

concrete MusicFin of Music = MusicI with
  (Grammar = GrammarFin),
  (MusicLex = MusicLexFin) ;
```

More work is of course needed if the language-independent linearizations in MusicI are not satisfactory for some language. The resource grammar guarantees that the linearizations are possible in all languages, in the sense of grammatical, but they might of course be inadequate for stylistic reasons. Assume, for the sake of argument, that adjectival modification does not sound good in English, but that a relative clause would be preferrable. One can then start as before,

```
concrete MusicLexEng of MusicLex =
    CatEng ** open ParadigmsEng in {
  lin
    song_N = regN "song" ;
    american_A = regA "American" ;
  }

concrete MusicEng0 of Music = MusicI with
  (Grammar = GrammarEng),
  (MusicLex = MusicLexEng) ;
```

The module `MusicEng0` would not be used on the top level, however, but another module would be built on top of it, with a restricted import from

`MusicEng0`. `MusicEng` inherits everything from `MusicEng0` except `PropKind`, and gives its own definition of this function:

```
concrete MusicEng of Music =
    MusicEng0 - [PropKind] ** open GrammarEng in {
  lin
    PropKind k p =
      RelCN k (UseRCl TPres ASimul PPos
        (RelVP IdRP (UseComp (CompAP p)))) ;
  }
```

1.2 Lock fields

When the categories of the resource grammar are used in applications, a **lock field** is added to their linearization types. The lock field makes the linearization type of each category unique, so that categories with the same implementation are not confused with each other. (This is inspired by the `newtype` discipline in Haskell.) For instance, the lincats of adverbs and conjunctions are the same in `CatEng` (and therefore in `GrammarEng`, which inherits it):

```
lincat Adv  = {s : Str} ;
lincat Conj = {s : Str} ;
```

But when these category symbols are used to denote their linearization types in an application, these definitions are translated to

```
oper Adv  : Type = {s : Str  ; lock_Adv  : {}} ;
oper Conj : Type = {s : Str} ; lock_Conj : {}} ;
```

In this way, the user of a resource grammar cannot confuse adverbs with conjunctions. In other words, the lock fields force the type checker to function as grammaticality checker.

When the resource grammar is `open`ed in an application grammar, and only functions from the resource are used in type-correct way, the lock fields are never seen (except possibly in type error messages). If an application grammarian has to write lock fields herself, it is a sign that the guarantees given by the resource grammar no longer hold. But since the resource may be incomplete, the application grammarian may occasionally have to provide the dummy values of lock fields (always <>, the empty record). Here is an example:

```
mkUtt : Str -> Utt ;
mkUtt s = {s = s ; lock_Utt = <>} ;
```

Currently, missing lock field produce warnings rather than errors, but this behaviour of GF may change in future.

1.3 Parsing with resource grammars?

The intended use of the resource grammar is as a library for writing application grammars. It is not designed for parsing e.g. newspaper text. There are several reasons why this is not practical:

- Efficiency: the resource grammar uses complex data structures, in particular, discontinuous constituents, which make parsing slow and the parser size huge.
- Completeness: the resource grammar does not necessarily cover all rules of the language - only enough many to be able to express everything in one way or another.
- Lexicon: the resource grammar has a very small lexicon, only meant for test purposes.
- Semantics: the resource grammar has very little semantic control, and may accept strange input or deliver strange interpretations.
- Ambiguity: parsing in the resource grammar may return lots of results many of which are implausible.

All of these problems should be solved in application grammars. The task of resource grammars is just to take care of low-level linguistic details such as inflection, agreement, and word order.

It is for the same reasons that resource grammars are not adequate for translation. That the syntax API is implemented for different languages of course makes it possible to translate via it - but there is no guarantee of translation equivalence. Of course, the use of functor implementations such as `MusicI` above only extends to those cases where the syntax API does give translation equivalence - but this must be seen as a limiting case, and bigger applications will often use only restricted inheritance of `MusicI`.

2 To find rules in the resource grammar library

2.1 Inflection paradigms

Inflection paradigms are defined separately for each language L in the module `Paradigms`L. To test them, the command `cc` (= `compute_concrete`) can be used:

```
> i -retain german/ParadigmsGer.gf

> cc regN "Schlange"
{
  s : Number => Case => Str = table Number {
    Sg => table Case {
      Nom => "Schlange" ;
      Acc => "Schlange" ;
      Dat => "Schlange" ;
      Gen => "Schlange"
      } ;
    Pl => table Case {
      Nom => "Schlangen" ;
      Acc => "Schlangen" ;
      Dat => "Schlangen" ;
      Gen => "Schlangen"
      }
    } ;
  g : Gender = Fem
}
```

For the sake of convenience, every language implements these five paradigms:

```
oper
  regN  : Str -> N ;   -- regular nouns
  regA  : Str -> A :   -- regular adjectives
  regV  : Str -> V ;   -- regular verbs
  regPN : Str -> PN ;  -- regular proper names
  dirV  : V   -> V2 ;  -- direct transitive verbs
```

It is often possible to initialize a lexicon by just using these functions, and later revise it by using the more involved paradigms. For instance, in German we cannot use `regN "Lied"` for `Song`, because the result would be a

Masculine noun with the plural form `"Liede"`. The individual `Paradigms` modules tell what cases are covered by the regular heuristics.

As a limiting case, one could even initialize the lexicon for a new language by copying the English (or some other already existing) lexicon. This would produce language with correct grammar but with content words directly borrowed from English - maybe not so strange in certain technical domains.

2.2 Syntax rules

Syntax rules should be looked for in the abstract modules defining the API. There are around 10 such modules, each defining constructors for a group of one or more related categories. For instance, the module `Noun` defines how to construct common nouns, noun phrases, and determiners. Thus the proper place to find out how nouns are modified with adjectives is `Noun`, because the result of the construction is again a common noun.

Browsing the libraries is helped by the gfdoc-generated HTML pages, whose LaTeX versions are included in the present document. However, this is still not easy, and the most efficient way is probably to use the parser. Even though parsing is not an intended end-user application of resource grammars, it is a useful technique for application grammarians to browse the library. To find out which resource function implements a particular structure, one can just parse a string that exemplifies this structure. For instance, to find out how sentences are built using transitive verbs, write

```
> i english/LangEng.gf

> p -cat=Cl -fcfg "she loves him"

PredVP (UsePron she_Pron) (ComplV2 love_V2 (UsePron he_Pron))
```

Parsing with the English resource grammar has an acceptable speed, but with most languages it takes just too much resources even to build the parser. However, examples parsed in one language can always be linearized into other languages:

```
> i italian/LangIta.gf

> l PredVP (UsePron she_Pron) (ComplV2 love_V2 (UsePron he_Pron))

lo ama
```

Therefore, one can use the English parser to write an Italian grammar, and also to write a language-independent (incomplete) grammar. One can also parse strings that are bizarre in English but the intended way of expression in another language. For instance, the phrase for "I am hungry" in Italian is literally "I have hunger". This can be built by parsing "I have beer" in LanEng and then writing

```
lin IamHungry =
  let beer_N = regGenN "fame" feminine
  in
  PredVP (UsePron i_Pron) (ComplV2 have_V2
    (DetCN (DetSg MassDet NoOrd) (UseN beer_N))) ;
```

which uses ParadigmsIta.regGenN.

2.3 Example-based grammar writing

The technique of parsing with the resource grammar can be used in GF source files, endowed with the suffix `.gfe` ("GF examples"). The suffix tells GF to preprocess the file by replacing all expressions of the form

```
in Module.Cat "example string"
```

by the syntax trees obtained by parsing "example string" in `Cat` in `Module`. For instance,

```
lin IamHungry =
  let beer_N = regGenN "fame" feminine
  in
  (in LangEng.Cl "I have beer") ;
```

will result in the rule displayed in the previous section. The normal binding rules of functional programming (and GF) guarantee that local bindings of identifiers take precedence over constants of the same forms. Thus it is also possible to linearize functions taking arguments in this way:

```
lin
  PropKind car_N old_A = in LangEng.CN "old car" ;
```

However, the technique of example-based grammar writing has some limitations:

- Ambiguity. If a string has several parses, the first one is returned, and it may not be the intended one. The other parses are shown in a comment, from where they must/can be picked manually.
- Lexicality. The arguments of a function must be atomic identifiers, and are thus not available for categories that have no lexical items. For instance, the `PropKind` rule above gives the result

  ```
  lin
    PropKind car_N old_A = AdjCN (UseN car_N) (PositA old_A) ;
  ```

  However, it is possible to write a special lexicon that gives atomic rules for all those categories that can be used as arguments, for instance,

  ```
  fun
    cat_CN : CN ;
    old_AP : AP ;
  ```

  and then use this lexicon instead of the standard one included in `Lang`.

2.4 Special-purpose APIs

To give an analogy with the well-known type setting software, GF can be compared with TeX and the resource grammar library with LaTeX. Just like TeX frees the author from thinking about low-level problems of page layout, so GF frees the grammarian from writing parsing and generation algorithms. But quite a lot of knowledge of *how* to write grammars is still needed, and the resource grammar library helps GF grammarians in a way similar to how the LaTeX macro package helps TeX authors.

But even LaTeX is often too detailed and low-level, and users are encouraged to develop their own macro packages. The same applies to GF resource grammars: the application grammarian might not need all the choises that the resource provides, but would prefer less writing and higher-level programming. To this end, application grammarians may want to write their own views on the resource grammar. An example of this is already provided, in `mathematical/Predication`. Instead of the `NP-VP` structure, it permits clause construction directly from verbs and adjectives and their arguments:

```
predV     : V  -> NP -> Cl ;             -- "x converges"
predV2    : V2 -> NP -> NP -> Cl ;       -- "x intersects y"
predV3    : V3 -> NP -> NP -> NP -> Cl ; -- "x intersects y at z"
predVColl : V  -> NP -> NP -> Cl ;       -- "x and y intersect"
predA     : A  -> NP -> Cl ;             -- "x is even"
predA2    : A2 -> NP -> NP -> Cl ;       -- "x is divisible by y"
```

The implementation of this module is the functor `PredicationI`:

```
predV v x = PredVP x (UseV v) ;
predV2 v x y = PredVP x (ComplV2 v y) ;
predV3 v x y z = PredVP x (ComplV3 v y z) ;
predVColl v x y = PredVP (ConjNP and_Conj (BaseNP x y)) (UseV v) ;
predA a x = PredVP x (UseComp (CompAP (PositA a))) ;
predA2 a x y = PredVP x (UseComp (CompAP (ComplA2 a y))) ;
```

Of course, `Predication` can be opened together with `Grammar`, but using the resulting grammar for parsing can be frustrating, since having both ways of building clauses simultaneously available will produce spurious ambiguities. But using just `Predication` without `Verb` for parsing is a good idea, since parsing is more efficient without rules producing verb phrases.

The use of special-purpose APIs is to some extent just an alternative to grammar writing by parsing, and its importance may decrease as parsing with resource grammars becomes more practical.

3 Overview of syntactic structures

3.1 Texts. phrases, and utterances

The outermost linguistic structure is `Text`. `Text`s are composed from Phrases (`Phr`) followed by punctuation marks - either of ".", "?" or "!" (with their proper variants in Spanish and Arabic). Here is an example of a `Text` string.

```
John walks. Why? He doesn't want to sleep!
```

Phrases are mostly built from Utterances (`Utt`), which in turn are declarative sentences, questions, or imperatives - but there are also "one-word utterances" consisting of noun phrases or other subsentential phrases. Some Phrases are atomic, for instance "yes" and "no". Here are some examples of Phrases.

```
yes
come on, John
but John walks
give me the stick please
don't you know that he is sleeping
a glass of wine
a glass of wine please
```

There is no connection between the punctuation marks and the types of utterances. This reflects the fact that the punctuation mark in a real text is selected as a function of the speech act rather than the grammatical form of an utterance. The following text is thus well-formed.

```
John walks. John walks? John walks!
```

What is the difference between Phrase and Utterance? Just technical: a Phrase is an Utterance with an optional leading conjunction ("but") and an optional tailing vocative ("John", "please").

3.2 Sentences and clauses

The richest of the categories below Utterance is `S`, Sentence. A Sentence is formed from a Clause (`Cl`), by fixing its Tense, Anteriority, and Polarity. The difference between Sentence and Clause is thus also rather technical. For example, each of the following strings has a distinct syntax tree in the category Sentence:

```
John walks
John doesn't walk
John walked
John didn't walk
John has walked
John hasn't walked
John will walk
John won't walk
...
```

whereas in the category Clause all of them are just different forms of the same tree.

Figure 1 shows a type-annotated syntax tree of the Text "John walks." and gives an overview of the structural levels.

Here are some examples of the results of changing constructors.

```
1. TFullStop -> TQuestMark   John walks?
3. NoPConj   -> but_PConj    But John walks.
6. TPres     -> TPast        John walked.
7. ASimul    -> AAnter       John has walked.
8. PPos      -> PNeg         John doesn't walk.
```

```
Node Constructor             Value type  Other constructors
-------------------------------------------------------------
 1.  TFullStop               Text        TQuestMark
 2.    (PhrUtt               Phr
 3.      NoPConj             PConj       but_PConj
 4.      (UttS               Utt         UttQS
 5.        (UseCl            S           UseQCl
 6.           TPres          Tense       TPast
 7.           ASimul         Anter       AAnter
 8.           PPos           Pol         PNeg
 9.           (PredVP        Cl
10.             (UsePN       NP          UsePron, DetCN
11.               john_PN)   PN          mary_PN
12.             (UseV        VP          ComplV2, ComplV3
13.               walk_V)))) V           sleep_V
14.      NoVoc)              Voc         please_Voc
15.    TEmpty                Text
```

Figure 1. Type-annotated syntax tree of the Text "John walks."

```
11. john_PN   -> mary_PN      Mary walks.
13. walk_V    -> sleep_V      John sleeps.
14. NoVoc     -> please_Voc   John sleeps please.
```

All constructors cannot of course be changed so freely, because the resulting tree would not remain well-typed. Here are some changes involving many constructors:

```
 4- 5. UttS (UseCl ...) ->
         UttQS (UseQCl (... QuestCl ...)) Does John walk?
10-11. UsePN john_PN    ->
         UsePron we_Pron                  We walk.
12-13. UseV walk_V      ->
         ComplV2 love_V2 this_NP          John loves this.
```

3.3 Parts of sentences

The linguistic phenomena mostly discussed in both traditional grammars and modern syntax belong to the level of Clauses, that is, lines 9-13, and occasionally to Sentences, lines 5-13. At this level, the major categories are `NP` (Noun Phrase) and `VP` (Verb Phrase). A Clause typically consists of just

an `NP` and a `VP`. The internal structure of both `NP` and `VP` can be very complex, and these categories are mutually recursive: not only can a `VP` contain an `NP`,

```
[VP loves [NP Mary]]
```

but also an `NP` can contain a `VP`

```
[NP every man [RS who [VP walks]]]
```

(a labelled bracketing like this is of course just a rough approximation of a GF syntax tree, but still a useful device of exposition).

Most of the resource modules thus define functions that are used inside NPs and VPs. Here is a brief overview:

Noun. How to construct NPs. The main three mechanisms for constructing NPs are

- from proper names: "John"
- from pronouns: "we"
- from common nouns by determiners: "this man"

The `Noun` module also defines the construction of common nouns. The most frequent ways are

- lexical noun items: "man"
- adjectival modification: "old man"
- relative clause modification: "man who sleeps"
- application of relational nouns: "successor of the number"

Verb. How to construct VPs. The main mechanism is verbs with their arguments, for instance,

- one-place verbs: "walks"
- two-place verbs: "loves Mary"

- three-place verbs: "gives her a kiss"
- sentence-complement verbs: "says that it is cold"
- VP-complement verbs: "wants to give her a kiss"

A special verb is the copula, "be" in English but not even realized by a verb in all languages. A copula can take different kinds of complement:

- an adjectival phrase: "(John is) old"
- an adverb: "(John is) here"
- a noun phrase: "(John is) a man"

Adjective. How to constuct `AP`s. The main ways are

- positive forms of adjectives: "old"
- comparative forms with object of comparison: "older than John"

Adverb. How to construct `Adv`s. The main ways are

- from adjectives: "slowly"
- as prepositional phrases: "in the car"

3.4 Modules and their names

The resource modules are named after the kind of phrases that are constructed in them, and they can be roughly classified by the "level" or "size" of expressions that are formed in them:

- Larger than sentence: `Text`, `Phrase`
- Same level as sentence: `Sentence`, `Question`, `Relative`
- Parts of sentence: `Adjective`, `Adverb`, `Noun`, `Verb`
- Cross-cut (coordination): `Conjunction`

Because of mutual recursion such as in embedded sentences, this classification is not a complete order. However, no mutual dependence is needed between the modules in a formal sense - they can all be compiled separately. This is due to the module `Cat`, which defines the type system common to the other modules. For instance, the types `NP` and `VP` are defined in `Cat`, and the module `Verb` only needs to know what is given in `Cat`, not what is given in `Noun`. To implement a rule such as

```
Verb.ComplV2 : V2 -> NP -> VP
```

it is enough to know the linearization type of `NP` (as well as those of `V2` and `VP`, all given in `Cat`). It is not necessary to know what ways there are to build `NP`s (given in `Noun`), since all these ways must conform to the linearization type defined in `Cat`. Thus the format of category-specific modules is as follows:

```
abstract Adjective = Cat ** {...}
abstract Noun      = Cat ** {...}
abstract Verb      = Cat ** {...}
```

3.5 Top-level grammar and lexicon

The module `Grammar` collects all the category-specific modules into a complete grammar:

```
abstract Grammar =
  Adjective, Noun, Verb, ..., Structural, Idiom
```

The module `Structural` is a lexicon of structural words (function words), such as determiners.

The module `Idiom` is a collection of idiomatic structures whose implementation is very language-dependent. An example is existential structures ("there is", "es gibt", "il y a", etc).

The module `Lang` combines `Grammar` with a `Lexicon` of ca. 350 content words:

```
abstract Lang = Grammar, Lexicon
```

Using `Lang` instead of `Grammar` as a library may give for free some words needed in an application. But its main purpose is to help testing the resource

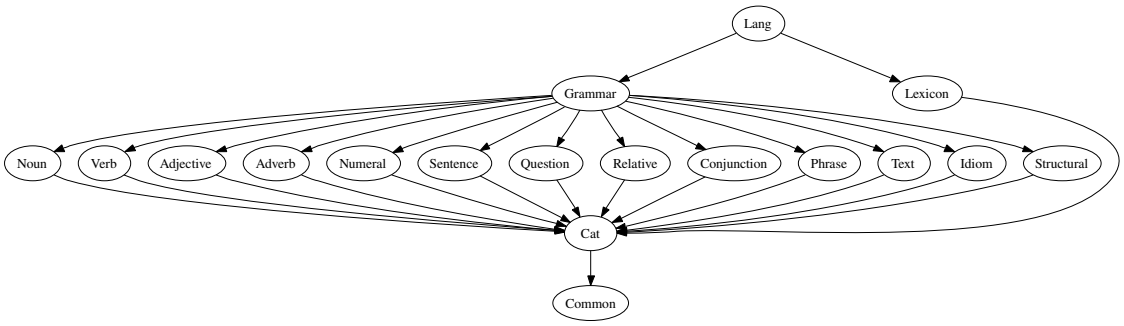

Figure 2. The resource syntax API.

library. It does not seem possible to maintain a general-purpose multilingual lexicon, and this is the form that the module `Lexicon` has.

The diagram in Figure 2 shows the structure of the API.

3.6 Language-specific syntactic structures

The API collected in `Grammar` has been designed to be implementable for all languages in the resource package. It does contain some rules that are strange or superfluous in some languages; for instance, the distinction between definite and indefinite articles does not apply to Finnish and Russian. But such rules are still easy to implement: they only create some superfluous ambiguity in the languages in question.

But the library makes no claim that all languages should have exactly the same abstract syntax. The common API is therefore extended by language-dependent rules. The top level of each languages looks as follows (with English as example):

```
abstract English = Grammar, ExtraEngAbs, DictEngAbs
```

where `ExtraEngAbs` is a collection of syntactic structures specific to English, and `DictEngAbs` is an English dictionary (at the moment, it consists of `IrregEngAbs`, the irregular verbs of English). Each of these language-specific grammars has the potential to grow into a full-scale grammar of the language. These grammar can also be used as libraries, but the possibility of using functors is lost.

To give a better overview of language-specific structures, modules like `ExtraEngAbs` are built from a language-independent module `ExtraAbs` by restricted inheritance:

```
abstract ExtraEngAbs = Extra [f,g,...]
```

Thus any category and function in `Extra` may be shared by a subset of all languages. One can see this set-up as a matrix, which tells what `Extra` structures are implemented in what languages. For the common API in `Grammar`, the matrix is filled with 1's (everything is implemented in every language).

Language-specific extensions and the use of restricted inheritance is a recent addition to the resource grammar library, and has only been exploited in a very small scale so far.

4 API Documentation

4.1 Top-level modules

4.1.1 Grammar: the Main Module of the Resource Grammar

This grammar a collection of the different grammar modules, To test the resource, import Lang, which also contains a lexicon.

```
abstract Grammar =
  Noun,
  Verb,
  Adjective,
  Adverb,
  Numeral,
  Sentence,
  Question,
  Relative,
  Conjunction,
  Phrase,
  Text,
  Structural,
  Idiom
  ** {} ;
```

4.1.2 Lang: a Test Module for the Resource Grammar

This grammar is for testing the resource as included in the language-independent API, consisting of a grammar and a lexicon. The grammar without a lexicon

is Grammar, which may be more suitable to open in applications.

```
abstract Lang =
  Grammar,
  Lexicon
  ** {} ;
```

4.2 Type system

4.2.1 Cat: the Category System

The category system is central to the library in the sense that the other modules (`Adjective`, `Adverb`, `Noun`, `Verb` etc) communicate through it. This means that a e.g. a function using `NP`s in `Verb` need not know how `NP`s are constructed in `Noun`: it is enough that both `Verb` and `Noun` use the same type `NP`, which is given here in `Cat`.

Some categories are inherited from Common. The reason they are defined there is that they have the same implementation in all languages in the resource (typically, just a string). These categories are `AdA, AdN, AdV, Adv, Ant, CAdv, IAdv, PConj, Phr, Pol, SC, Tense, Text, Utt, Voc`.

Moreover, the list categories `ListAdv, ListAP, ListNP, ListS` are defined on `Conjunction` and only used locally there.

```
abstract Cat = Common ** {

  cat
```

Sentences and clauses

Constructed in Sentence, and also in Idiom.

```
    S ;     -- declarative sentence                e.g. "she lived here"
    QS ;    -- question                            e.g. "where did she live"
    RS ;    -- relative                            e.g. "in which she lived"
    Cl ;    -- declarative clause, with all tenses e.g. "she looks at this"
    Slash ; -- clause missing NP (S/NP in GPSG)    e.g. "she looks at"
    Imp ;   -- imperative                          e.g. "look at this"
```

Questions and interrogatives

Constructed in Question.

```
QCl ;   -- question clause, with all tenses    e.g. "why does she walk"
IP ;    -- interrogative pronoun               e.g. "who"
IComp ; -- interrogative complement of copula  e.g. "where"
IDet ;  -- interrogative determiner            e.g. "which"
```

Relative clauses and pronouns

Constructed in Relative.

```
RCl ;   -- relative clause, with all tenses    e.g. "in which she lives"
RP ;    -- relative pronoun                    e.g. "in which"
```

Verb phrases

Constructed in Verb.

```
VP ;    -- verb phrase                         e.g. "is very warm"
Comp ;  -- complement of copula, such as AP    e.g. "very warm"
```

Adjectival phrases

Constructed in Adjective.

```
AP ;    -- adjectival phrase                   e.g. "very warm"
```

Nouns and noun phrases

Constructed in Noun. Many atomic noun phrases e.g. *everybody* are constructed in Structural. The determiner structure is

```
Predet (QuantSg | QuantPl Num) Ord
```

as defined in Noun.

```
CN ;    -- common noun (without determiner)    e.g. "red house"
NP ;    -- noun phrase (subject or object)     e.g. "the red house"
Pron ;  -- personal pronoun                    e.g. "she"
Det ;   -- determiner phrase                   e.g. "all the seven"
Predet; -- predeterminer (prefixed Quant)      e.g. "all"
QuantSg;-- quantifier ('nucleus' of sing. Det) e.g. "every"
QuantPl;-- quantifier ('nucleus' of plur. Det) e.g. "many"
Quant ; -- quantifier with both sg and pl      e.g. "this/these"
Num ;   -- cardinal number (used with QuantPl) e.g. "seven"
Ord ;   -- ordinal number (used in Det)        e.g. "seventh"
```

Numerals

Constructed in Numeral.

```
    Numeral;-- cardinal or ordinal,              e.g. "five/fifth"
```

Structural words

Constructed in Structural.

```
    Conj ;  -- conjunction,                      e.g. "and"
    DConj ; -- distributed conj.                 e.g. "both - and"
    Subj ;  -- subjunction,                      e.g. "if"
    Prep ;  -- preposition, or just case         e.g. "in"
```

Words of open classes

These are constructed in Lexicon and in additional lexicon modules.

```
    V ;     -- one-place verb                    e.g. "sleep"
    V2 ;    -- two-place verb                    e.g. "love"
    V3 ;    -- three-place verb                  e.g. "show"
    VV ;    -- verb-phrase-complement verb       e.g. "want"
    VS ;    -- sentence-complement verb          e.g. "claim"
    VQ ;    -- question-complement verb          e.g. "ask"
    VA ;    -- adjective-complement verb         e.g. "look"
    V2A ;   -- verb with NP and AP complement    e.g. "paint"

    A ;     -- one-place adjective               e.g. "warm"
    A2 ;    -- two-place adjective               e.g. "divisible"

    N ;     -- common noun                       e.g. "house"
    N2 ;    -- relational noun                   e.g. "son"
    N3 ;    -- three-place relational noun       e.g. "connection"
    PN ;    -- proper name                       e.g. "Paris"

}
```

4.2.2 Common: Structures with Common Implementations

This module defines the categories that uniformly have the linearization {`s : Str`} in all languages. Moreover, this module defines the abstract parameters of tense, polarity, and anteriority, which are used in Phrase to generate

different forms of sentences. Together they give 2 x 4 x 4 = 16 sentence forms. These tenses are defined for all languages in the library. More tenses can be defined in the language extensions, e.g. the *passé simple* of Romance languages.

```
abstract Common = {

  cat
```

Top-level units

Constructed in Text: `Text`.

```
    Text ;  -- text consisting of several phrases  e.g. "He is here. Why?"
```

Constructed in Phrase:

```
    Phr ;   -- phrase in a text                    e.g. "but be quiet please"
    Utt ;   -- sentence, question, word...         e.g. "be quiet"
    Voc ;   -- vocative or "please"                e.g. "my darling"
    PConj ; -- phrase-beginning conj.              e.g. "therefore"
```

Constructed in Sentence:

```
    SC ;    -- embedded sentence or question       e.g. "that it rains"
```

Adverbs

Constructed in Adverb. Many adverbs are constructed in Structural.

```
    Adv ;   -- verb-phrase-modifying adverb,       e.g. "in the house"
    AdV ;   -- adverb directly attached to verb    e.g. "always"
    AdA ;   -- adjective-modifying adverb,         e.g. "very"
    AdN ;   -- numeral-modifying adverb,           e.g. "more than"
    IAdv ;  -- interrogative adverb                e.g. "why"
    CAdv ;  -- comparative adverb                  e.g. "more"
```

Tense, polarity, and anteriority

```
    Tense ; -- tense: present, past, future, conditional
    Pol ;   -- polarity: positive, negative
    Ant ;   -- anteriority: simultaneous, anterior

  fun
    PPos, PNeg : Pol ;           -- I sleep/don't sleep

    TPres  : Tense ;
    ASimul : Ant ;
    TPast, TFut, TCond : Tense ; -- I slept/will sleep/would sleep --# notpresent
    AAnter : Ant ;               -- I have slept                   --# notpresent

}
```

4.3 Syntax rule modules

4.3.1 Adjective: Adjectives and Adjectival Phrases

```
abstract Adjective = Cat ** {

  fun
```

The principal ways of forming an adjectival phrase are positive, comparative, relational, reflexive-relational, and elliptic-relational. (The superlative use is covered in Noun.`SuperlA`.)

```
    PositA  : A -> AP ;          -- warm
    ComparA : A -> NP -> AP ;    -- warmer than Spain
    ComplA2 : A2 -> NP -> AP ;   -- divisible by 2
    ReflA2  : A2 -> AP ;         -- divisible by itself
    UseA2   : A2 -> A ;          -- divisible
```

Sentence and question complements defined for all adjectival phrases, although the semantics is only clear for some adjective.

```
    SentAP  : AP -> SC -> AP ;  -- great that she won, uncertain if she did
```

An adjectival phrase can be modified by an **adadjective**, such as *very*.

```
    AdAP    : AdA -> AP -> AP ; -- very uncertain
```

The formation of adverbs from adjective (e.g. *quickly*) is covered by Adverb.

```
}
```

4.3.2 Adverb: Adverbs and Adverbial Phrases

```
abstract Adverb = Cat ** {

  fun
```

The two main ways of forming adverbs are from adjectives and by prepositions from noun phrases.

```
    PositAdvAdj : A -> Adv ;                -- quickly
    PrepNP      : Prep -> NP -> Adv ;       -- in the house
```

Comparative adverbs have a noun phrase or a sentence as object of comparison.

```
    ComparAdvAdj  : CAdv -> A -> NP -> Adv ; -- more quickly than John
    ComparAdvAdjS : CAdv -> A -> S -> Adv ;  -- more quickly than he runs
```

Adverbs can be modified by 'adadjectives', just like adjectives.

```
    AdAdv  : AdA -> Adv -> Adv ;            -- very quickly
```

Subordinate clauses can function as adverbs.

```
    SubjS : Subj -> S -> Adv ;              -- when he arrives
    AdvSC : SC -> Adv ;                     -- that he arrives ---- REMOVE?
```

Comparison adverbs also work as numeral adverbs.

```
    AdnCAdv : CAdv -> AdN ;                 -- more (than five)
```

```
}
```

4.3.3 Conjunction: Coordination

Coordination is defined for many different categories; here is a sample. The rules apply to **lists** of two or more elements, and define two general patterns:

- ordinary conjunction: X,...X and X
- distributed conjunction: both X,...,X and X

Note. This module uses right-recursive lists. If backward compatibility with API 0.9 is needed, use SeqConjunction.

```
abstract Conjunction = Cat ** {
```

Rules

```
fun
  ConjS    : Conj -> [S] -> S ;     -- "John walks and Mary runs"
  ConjAP   : Conj -> [AP] -> AP ;   -- "even and prime"
  ConjNP   : Conj -> [NP] -> NP ;   -- "John or Mary"
  ConjAdv  : Conj -> [Adv] -> Adv ; -- "quickly or slowly"

  DConjS   : DConj -> [S] -> S ;    -- "either John walks or Mary runs"
  DConjAP  : DConj -> [AP] -> AP ;  -- "both even and prime"
  DConjNP  : DConj -> [NP] -> NP ;  -- "either John or Mary"
  DConjAdv : DConj -> [Adv] -> Adv; -- "both badly and slowly"
```

Categories

These categories are only used in this module.

```
cat
  [S]{2} ;
  [Adv]{2} ;
  [NP]{2} ;
  [AP]{2} ;
```

List constructors

The list constructors are derived from the list notation and therefore not given explicitly. But here are their type signatures:

```
  --  BaseC : C -> C   -> [C] ;  -- for C = S, AP, NP, Adv
  --  ConsC : C -> [C] -> [C] ;
}
```

4.3.4 Idiom: Idiomatic Expressions

```
abstract Idiom = Cat ** {
```

This module defines constructions that are formed in fixed ways, often different even in closely related languages.

```
  fun
    ImpersCl  : VP -> Cl ;         -- it rains
    GenericCl : VP -> Cl ;         -- one sleeps

    CleftNP   : NP  -> RS -> Cl ; -- it is you who did it
    CleftAdv  : Adv -> S  -> Cl ; -- it is yesterday she arrived

    ExistNP   : NP -> Cl ;         -- there is a house
    ExistIP   : IP -> QCl ;        -- which houses are there

    ProgrVP   : VP -> VP ;         -- be sleeping

    ImpPl1    : VP -> Utt ;        -- let's go

}
```

4.3.5 Noun: Nouns, Noun Phrases, and Determiners

```
abstract Noun = Cat ** {
```

Noun phrases

The three main types of noun phrases are

- common nouns with determiners
- proper names
- pronouns

```
fun
  DetCN   : Det -> CN -> NP ;   -- the man
  UsePN   : PN -> NP ;          -- John
  UsePron : Pron -> NP ;        -- he
```

Pronouns are defined in the module Structural. A noun phrase already formed can be modified by a `Predet`erminer.

```
PredetNP : Predet -> NP -> NP; -- only the man
```

A noun phrase can also be postmodified by the past participle of a verb or by an adverb.

```
PPartNP : NP -> V2  -> NP ;    -- the number squared
AdvNP   : NP -> Adv -> NP ;    -- Paris at midnight
```

Determiners

The determiner has a fine-grained structure, in which a 'nucleus' quantifier and two optional parts can be discerned. The cardinal numeral is only available for plural determiners. (This is modified from CLE by further dividing their `Num` into cardinal and ordinal.)

```
DetSg : QuantSg ->        Ord -> Det ;  -- this best man
DetPl : QuantPl -> Num -> Ord -> Det ;  -- these five best men
```

Quantifiers that have both forms can be used in both ways.

```
SgQuant : Quant -> QuantSg ;            -- this
PlQuant : Quant -> QuantPl ;            -- these
```

Pronouns have possessive forms. Genitives of other kinds of noun phrases are not given here, since they are not possible in e.g. Romance languages.

```
PossPron : Pron -> Quant ;    -- my (house)
```

All parts of the determiner can be empty, except `Quant`, which is the *kernel* of a determiner.

```
NoNum  : Num ;
NoOrd  : Ord ;
```

`Num` consists of either digits or numeral words.

```
NumInt     : Int -> Num ;     -- 51
NumNumeral : Numeral -> Num ; -- fifty-one
```

The construction of numerals is defined in Numeral. `Num` can be modified by certain adverbs.

```
AdNum : AdN -> Num -> Num ;   -- almost 51
```

`Ord` consists of either digits or numeral words.

```
OrdInt     : Int -> Ord ;     -- 51st
OrdNumeral : Numeral -> Ord ; -- fifty-first
```

Superlative forms of adjectives behave syntactically in the same way as ordinals.

```
OrdSuperl : A -> Ord ;        -- largest
```

Definite and indefinite constructions are sometimes realized as neatly distinct words (Spanish *un, unos ; el, los*) but also without any particular word (Finnish; Swedish definites).

```
DefArt   : Quant ;            -- the (house), the (houses)
IndefArt : Quant ;            -- a (house), (houses)
```

Nouns can be used without an article as mass nouns. The resource does not distinguish mass nouns from other common nouns, which can result in semantically odd expressions.

```
MassDet  : QuantSg ;          -- (beer)
```

Other determiners are defined in Structural.

Common nouns

Simple nouns can be used as nouns outright.

```
    UseN : N -> CN ;              -- house
```

Relational nouns take one or two arguments.

```
    ComplN2 : N2 -> NP -> CN ;    -- son of the king
    ComplN3 : N3 -> NP -> N2 ;    -- flight from Moscow (to Paris)
```

Relational nouns can also be used without their arguments. The semantics is typically derivative of the relational meaning.

```
    UseN2   : N2 -> CN ;          -- son
    UseN3   : N3 -> CN ;          -- flight
```

Nouns can be modified by adjectives, relative clauses, and adverbs (the last rule will give rise to many 'PP attachement' ambiguities when used in connection with verb phrases).

```
    AdjCN   : AP -> CN  -> CN ;   -- big house
    RelCN   : CN -> RS  -> CN ;   -- house that John owns
    AdvCN   : CN -> Adv -> CN ;   -- house on the hill
```

Nouns can also be modified by embedded sentences and questions. For some nouns this makes little sense, but we leave this for applications to decide. Sentential complements are defined in Verb.

```
    SentCN  : CN -> SC  -> CN ;   -- fact that John smokes, question if he does
```

Apposition

This is certainly overgenerating.

```
    ApposCN : CN -> NP -> CN ;    -- number x, numbers x and y

} ;
```

4.3.6 Numeral: Cardinal and Ordinal Numerals

This grammar defines numerals from 1 to 999999. The implementations are adapted from the numerals library which defines numerals for 88 languages.

The resource grammar implementations add to this inflection (if needed) and ordinal numbers. **Note**. Number 1 as defined in the category `Numeral` here should not be used in the formation of noun phrases, and should therefore be removed. Instead, one should use Structural`.one_Quant`. This makes the grammar simpler because we can assume that numbers form plural noun phrases.

```
abstract Numeral = Cat ** {

cat
  Digit ;       -- 2..9
  Sub10 ;       -- 1..9
  Sub100 ;      -- 1..99
  Sub1000 ;     -- 1..999
  Sub1000000 ;  -- 1..999999

fun
  num : Sub1000000 -> Numeral ;

  n2, n3, n4, n5, n6, n7, n8, n9 : Digit ;

  pot01 : Sub10 ;                                -- 1
  pot0 : Digit -> Sub10 ;                        -- d * 1
  pot110 : Sub100 ;                              -- 10
  pot111 : Sub100 ;                              -- 11
  pot1to19 : Digit -> Sub100 ;                   -- 10 + d
  pot0as1 : Sub10 -> Sub100 ;                    -- coercion of 1..9
  pot1 : Digit -> Sub100 ;                       -- d * 10
  pot1plus : Digit -> Sub10 -> Sub100 ;          -- d * 10 + n
  pot1as2 : Sub100 -> Sub1000 ;                  -- coercion of 1..99
  pot2 : Sub10 -> Sub1000 ;                      -- m * 100
  pot2plus : Sub10 -> Sub100 -> Sub1000 ;        -- m * 100 + n
  pot2as3 : Sub1000 -> Sub1000000 ;              -- coercion of 1..999
  pot3 : Sub1000 -> Sub1000000 ;                 -- m * 1000
  pot3plus : Sub1000 -> Sub1000 -> Sub1000000 ; -- m * 1000 + n

}
```

4.3.7 Phrase: Phrases and Utterances

```
abstract Phrase = Cat ** {
```

When a phrase is built from an utterance it can be prefixed with a phrasal conjunction (such as *but*, *therefore*) and suffixing with a vocative (typically a noun phrase).

```
  fun
    PhrUtt   : PConj -> Utt -> Voc -> Phr ; -- But go home my friend.
```

Utterances are formed from sentences, questions, and imperatives.

```
    UttS     : S -> Utt ;                  -- John walks
    UttQS    : QS -> Utt ;                 -- is it good
    UttImpSg : Pol -> Imp -> Utt;          -- (don't) help yourself
    UttImpPl : Pol -> Imp -> Utt;          -- (don't) help yourselves
```

There are also 'one-word utterances'. A typical use of them is as answers to questions. **Note**. This list is incomplete. More categories could be covered. Moreover, in many languages e.g. noun phrases in different cases can be used.

```
    UttIP   : IP   -> Utt ;                -- who
    UttIAdv : IAdv -> Utt ;                -- why
    UttNP   : NP   -> Utt ;                -- this man
    UttAdv  : Adv  -> Utt ;                -- here
    UttVP   : VP   -> Utt ;                -- to sleep
```

The phrasal conjunction is optional. A sentence conjunction can also used to prefix an utterance.

```
    NoPConj   : PConj ;
    PConjConj : Conj -> PConj ;            -- and
```

The vocative is optional. Any noun phrase can be made into vocative, which may be overgenerating (e.g. *I*).

```
    NoVoc   : Voc ;
    VocNP   : NP -> Voc ;                  -- my friend

}
```

4.3.8 Question: Questions and Interrogative Pronouns

```
abstract Question = Cat ** {
```

A question can be formed from a clause ('yes-no question') or with an interrogative.

```
  fun
    QuestCl     : Cl -> QCl ;                 -- does John walk
    QuestVP     : IP -> VP -> QCl ;           -- who walks
    QuestSlash  : IP -> Slash -> QCl ;        -- who does John love
    QuestIAdv   : IAdv -> Cl -> QCl ;         -- why does John walk
    QuestIComp  : IComp -> NP -> QCl ;        -- where is John
```

Interrogative pronouns can be formed with interrogative determiners.

```
    IDetCN  : IDet -> Num -> Ord -> CN -> IP;  -- which five best songs
    AdvIP   : IP -> Adv -> IP ;                -- who in Europe

    PrepIP  : Prep -> IP -> IAdv ;             -- with whom

    CompIAdv : IAdv -> IComp ;                 -- where
```

More `IP`, `IDet`, and `IAdv` are defined in Structural.

```
}
```

4.3.9 Relative: Relative Clauses and Relative Pronouns

```
abstract Relative = Cat ** {

  fun
```

The simplest way to form a relative clause is from a clause by a pronoun similar to *such that*.

```
    RelCl    : Cl -> RCl ;            -- such that John loves her
```

The more proper ways are from a verb phrase (formed in Verb) or a sentence with a missing noun phrase (formed in Sentence).

```
    RelVP    : RP -> VP -> RCl ;      -- who loves John
    RelSlash : RP -> Slash -> RCl ;   -- whom John loves
```

Relative pronouns are formed from an 'identity element' by prefixing or suffixing (depending on language) prepositional phrases.

```
    IdRP  : RP ;                      -- which
    FunRP : Prep -> NP -> RP -> RP ;  -- all the roots of which

}
```

4.3.10 Sentence: Sentences, Clauses, and Imperatives

```
abstract Sentence = Cat ** {
```

Clauses

The `NP VP` predication rule form a clause whose linearization gives a table of all tense variants, positive and negative. Clauses are converted to `S` (with fixed tense) in Tensed.

```
  fun
    PredVP    : NP -> VP -> Cl ;          -- John walks
```

Using an embedded sentence as a subject is treated separately. This can be overgenerating. E.g. *whether you go* as subject is only meaningful for some verb phrases.

```
    PredSCVP  : SC -> VP -> Cl ;          -- that you go makes me happy
```

Clauses missing object noun phrases

This category is a variant of the 'slash category' `S/NP` of GPSG and categorial grammars, which in turn replaces movement transformations in the formation of questions and relative clauses. Except `SlashV2`, the construction rules can be seen as special cases of function composition, in the style of CCG. **Note** the set is not complete and lacks e.g. verbs with more than 2 places.

```
    SlashV2   : NP -> V2 -> Slash ;       -- (whom) he sees
    SlashVVV2 : NP -> VV -> V2 -> Slash;  -- (whom) he wants to see
    AdvSlash  : Slash -> Adv -> Slash ;   -- (whom) he sees tomorrow
    SlashPrep : Cl -> Prep -> Slash ;     -- (with whom) he walks
```

Imperatives

An imperative is straightforwardly formed from a verb phrase. It has variation over positive and negative, singular and plural. To fix these parameters, see Phrase.

```
    ImpVP     : VP -> Imp ;              -- go
```

Embedded sentences

Sentences, questions, and infinitival phrases can be used as subjects and (adverbial) complements.

```
    EmbedS    : S  -> SC ;               -- that you go
    EmbedQS   : QS -> SC ;               -- whether you go
    EmbedVP   : VP -> SC ;               -- to go
```

Sentences

These are the 2 x 4 x 4 = 16 forms generated by different combinations of tense, polarity, and anteriority, which are defined in Tense.

```
  fun
    UseCl  : Tense -> Ant -> Pol -> Cl  -> S ;
    UseQCl : Tense -> Ant -> Pol -> QCl -> QS ;
    UseRCl : Tense -> Ant -> Pol -> RCl -> RS ;

}
```

4.3.11 Structural: Structural Words

Here we have some words belonging to closed classes and appearing in all languages we have considered. Sometimes they are not really meaningful, e.g. `we_Pron` in Spanish should be replaced by masculine and feminine variants.

```
abstract Structural = Cat ** {

  fun
```

This is an alphabetical list of structural words

```
above_Prep : Prep ;
after_Prep : Prep ;
all_Predet : Predet ;
almost_AdA : AdA ;
almost_AdN : AdN ;
although_Subj : Subj ;
always_AdV : AdV ;
and_Conj : Conj ;
because_Subj : Subj ;
before_Prep : Prep ;
behind_Prep : Prep ;
between_Prep : Prep ;
both7and_DConj : DConj ;
but_PConj : PConj ;
by8agent_Prep : Prep ;
by8means_Prep : Prep ;
can8know_VV : VV ;
can_VV : VV ;
during_Prep : Prep ;
either7or_DConj : DConj ;
every_Det : Det ;
everybody_NP : NP ;
everything_NP : NP ;
everywhere_Adv : Adv ;
first_Ord : Ord ;
few_Det : Det ;
from_Prep : Prep ;
he_Pron : Pron ;
here_Adv : Adv ;
here7to_Adv : Adv ;
here7from_Adv : Adv ;
how_IAdv : IAdv ;
how8many_IDet : IDet ;
i_Pron : Pron ;
if_Subj : Subj ;
in8front_Prep : Prep ;
in_Prep : Prep ;
it_Pron : Pron ;
less_CAdv : CAdv ;
many_Det : Det ;
more_CAdv : CAdv ;
most_Predet : Predet ;
much_Det : Det ;
must_VV : VV ;
```

```
no_Phr : Phr ;
on_Prep : Prep ;
one_Quant : QuantSg ;
only_Predet : Predet ;
or_Conj : Conj ;
otherwise_PConj : PConj ;
part_Prep : Prep ;
please_Voc : Voc ;
possess_Prep : Prep ;
quite_Adv : AdA ;
she_Pron : Pron ;
so_AdA : AdA ;
someSg_Det : Det ;
somePl_Det : Det ;
somebody_NP : NP ;
something_NP : NP ;
somewhere_Adv : Adv ;
that_Quant : Quant ;
that_NP : NP ;
there_Adv : Adv ;
there7to_Adv : Adv ;
there7from_Adv : Adv ;
therefore_PConj : PConj ;
these_NP : NP ;
they_Pron : Pron ;
this_Quant : Quant ;
this_NP : NP ;
those_NP : NP ;
through_Prep : Prep ;
to_Prep : Prep ;
too_AdA : AdA ;
under_Prep : Prep ;
very_AdA : AdA ;
want_VV : VV ;
we_Pron : Pron ;
whatPl_IP : IP ;
whatSg_IP : IP ;
when_IAdv : IAdv ;
when_Subj : Subj ;
where_IAdv : IAdv ;
whichPl_IDet : IDet ;
whichSg_IDet : IDet ;
whoPl_IP : IP ;
whoSg_IP : IP ;
```

```
    why_IAdv : IAdv ;
    with_Prep : Prep ;
    without_Prep : Prep ;
    yes_Phr : Phr ;
    youSg_Pron : Pron ;
    youPl_Pron : Pron ;
    youPol_Pron : Pron ;

  }
```

4.3.12 Text: Texts

Texts are built from an empty text by adding `Phr`ases, using as constructors the punctuation marks ., *?*, and *!*. Any punctuation mark can be attached to any kind of phrase.

```
  abstract Text = Common ** {

    fun
      TEmpty     : Text ;                  --
      TFullStop  : Phr -> Text -> Text ;   -- John walks. ...
      TQuestMark : Phr -> Text -> Text ;   -- Are you OK? ...
      TExclMark  : Phr -> Text -> Text ;   -- John walks! ...

  }
```

4.3.13 Verb: Verb Phrases

```
  abstract Verb = Cat ** {
```

Complementization rules

Verb phrases are constructed from verbs by providing their complements. There is one rule for each verb category.

```
    fun
      UseV     : V   -> VP ;              -- sleep
      ComplV2  : V2  -> NP -> VP ;        -- use it
      ComplV3  : V3  -> NP -> NP -> VP ;  -- send a message to her

      ComplVV  : VV  -> VP -> VP ;        -- want to run
```

```
ComplVS  : VS  -> S  -> VP ;        -- know that she runs
ComplVQ  : VQ  -> QS -> VP ;        -- ask if she runs

ComplVA  : VA  -> AP -> VP ;        -- look red
ComplV2A : V2A -> NP -> AP -> VP ;  -- paint the house red
```

Other ways of forming verb phrases

Verb phrases can also be constructed reflexively and from copula-preceded complements.

```
ReflV2   : V2 -> VP ;               -- use itself
UseComp  : Comp -> VP ;             -- be warm
```

Passivization of two-place verbs is another way to use them. In many languages, the result is a participle that is used as complement to a copula (*is used*), but other auxiliary verbs are possible (Ger. *wird angewendet*, It. *viene usato*), as well as special verb forms (Fin. *käytetään*, Swe. *används*).

Note. the rule can be overgenerating, since the `V2` need not take a direct object.

```
PassV2   : V2 -> VP ;               -- be used
```

Adverbs can be added to verb phrases. Many languages make a distinction between adverbs that are attached in the end vs. next to (or before) the verb.

```
AdvVP    : VP -> Adv -> VP ;        -- sleep here
AdVVP    : AdV -> VP -> VP ;        -- always sleep
```

Agents of passives are constructed as adverbs with the preposition Structural.`8agent_Prep`.

Complements to copula

Adjectival phrases, noun phrases, and adverbs can be used.

```
CompAP   : AP  -> Comp ;            -- (be) small
CompNP   : NP  -> Comp ;            -- (be) a soldier
CompAdv  : Adv -> Comp ;            -- (be) here
```

Coercions

Verbs can change subcategorization patterns in systematic ways, but this is very much language-dependent. The following two work in all the languages we cover.

```
    UseVQ   : VQ -> V2 ;                -- ask (a question)
    UseVS   : VS -> V2 ;                -- know (a secret)

}
```

4.4 Inflectional paradigms

4.4.1 Arabic

Ali El Dada 2005–2006

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoAra.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularAra`, which covers all irregularly inflected words.

The following modules are presupposed:

```
resource ParadigmsAra = open
  Predef,
  Prelude,
  MorphoAra,
  OrthoAra,
  CatAra
  in {
```

```
flags optimize = noexpand;

oper

  --lexical paradigms for nouns

  mkN : NTable -> Gender -> Species -> N =
    \nsc,gen,spec ->
    { s = nsc;
      g = gen;
      h = spec;
      lock_N = <>
    };

  --takes a root string, a singular pattern string, a broken plural
  --pattern string, a gender, and species. Gives a noun
  regN : Str -> Str -> Str -> Gender -> Species -> N =
    \root,sg,pl,gen,spec ->
    let { raw = regN' root sg pl gen spec} in
    { s = \\n,d,c =>
        case root of {
          _ + "Ø" + _ => rectifyHmz(raw.s ! n ! d ! c);
          _ => raw.s ! n ! d ! c
        };
      g = gen;
      h = spec
    };

  regN' : Str -> Str -> Str -> Gender -> Species -> N =
    \root,sg,pl,gen,spec ->
    let { kitAb = mkWord sg root;
          kutub = mkWord pl root
    } in mkN (reg kitAb kutub) gen spec;

  --takes a root string, a singular pattern string, a gender,
  --and species. Gives a noun whose plural is sound feminine
  sdfN : Str -> Str -> Gender -> Species -> N =
    \root,sg,gen,spec ->
    let { kalima = mkWord sg root;
    } in mkN (sndf kalima) gen spec;

  --takes a root string, a singular pattern string, a gender,
  --and species. Gives a noun whose plural is sound masculine
  sdmN : Str -> Str -> Gender -> Species -> N =
```

```
    \root,sg,gen,spec ->
    let { mucallim = mkWord sg root;
    } in mkN (sndm mucallim) gen spec;

  --    mkN3 : N -> Str -> Str -> N3 =
  --      \n,p,q -> n ** {c2 = p ; c3 = q; lock_N3 = <>} ;

  --lexical paradigms for adjectives

  --takes a root string and a pattern string
  regA : Str -> Str -> A =
    \root,pat ->
    let { raw = regA' root pat } in
    { s = \\g,n,d,c =>
        case root of {
          _ + "Ø" + _ => rectifyHmz(raw.s ! g ! n ! d ! c);
          _ => raw.s ! g ! n ! d ! c
        };
      lock_A = <>
    };

  regA' : Str -> Str -> A =
    \root,pat ->
    let { kabIr = mkWord pat root
    } in {
      s = adj kabIr;
    };

--takes a root string only
clrA : Str -> A =
  \root ->
  let { eaHmar = mkWord "Ø£ÙÙØ¹ÙÙ" root;
        HamrA' = mkWord "ÙÙØ¹ÙØ§Ø¡" root;
        Humr   = mkWord "ÙÙØ¹Ù" root
  } in {
    s = clr eaHmar HamrA' Humr;
    lock_A = <>
  };

--lexical paradigms for verbs

v1 : Str -> Vowel -> Vowel -> V =
  \rootStr,vPerf,vImpf ->
  let { raw = v1' rootStr vPerf vImpf } in
```

```
{ s = \\vf =>
    case rootStr of {
      _ + "Ø" + _ => rectifyHmz(raw.s ! vf);
      _ => raw.s ! vf
    };
  lock_V = <>
} ;

v1' : Str ->  Vowel -> Vowel -> Verb =
  \rootStr,vPerf,vImpf ->
  let { root = mkRoot3 rootStr ;
        l = dp 2 rootStr } in --last rootStr
  case <l, root.c> of {
    <"Ù",_>     => v1geminate rootStr vPerf vImpf ;
    <"Ù"|"Ù",_> => v1defective root vImpf ;
    <_,"Ù"|"Ù"> => v1hollow root vImpf ;
    _     => v1sound root vPerf vImpf
  };

--Verb Form II : faccala

v2 : Str -> V =
  \rootStr ->
  let {
    root = mkRoot3 rootStr
  } in {
    s =
      case root.l of {
        "Ù"|"Ù" => (v2defective root).s;
        _       => (v2sound root).s
      };
    lock_V = <>
  };

--Verb Form III : fAcala

v3 : Str -> V =
  \rootStr ->
  let {
    tbc = mkRoot3 rootStr ;
  } in {
    s = (v3sound tbc).s  ;
    lock_V = <>
  };
```

```
--Verb Form IV : >afcala

v4 : Str -> V =
  \rootStr ->
  let {
    root = mkRoot3 rootStr
  } in {
    s =
      case root.l of {
        "Ù"|"Ù" => (v4defective root).s;
        _       => (v4sound root).s
      };
    lock_V = <>
  };

--Verb Form V : tafaccala

v5 : Str -> V =
  \rootStr ->
  let { raw = v5' rootStr } in
  { s = \\vf =>
      case rootStr of {
        _ + "Ø" + _ => rectifyHmz(raw.s ! vf);
        _ => raw.s ! vf
      };
    lock_V = <>
  };

v5' : Str -> V =
  \rootStr ->
  let {
    nfs = mkRoot3 rootStr ;
  } in {
    s = (v5sound nfs).s  ;
  };

--Verb Form VI : tafaacala

v6 : Str -> V =
  \rootStr ->
  let {
    fqm = mkRoot3 rootStr ;
  } in {
```

```
    s = (v6sound fqm).s  ;
    lock_V = <>
  };

--Verb Form VIII <iftacala

v8 : Str -> V =
  \rootStr ->
  let {
    rbT = mkRoot3 rootStr ;
  } in {
    s = (v8sound rbT).s ;
    lock_V = <>
  };
```

– Prepositions are used in many-argument functions for rection.

```
Preposition : Type ;
```

Nouns

Use the function `mkPreposition` or see the section on prepositions below to form other prepositions.

Relational nouns

Relational nouns (*Ø?ÙÙØ?ÙØ?Ø± ÙÙ Ø?*) need a preposition.

```
mkN2 : N -> Preposition -> N2 ;
```

Three-place relational nouns (*Ø?ÙÙ Ø¹ÙÙÙØ¹Ø?ÙÙÙ ÙØ±ÙÙ Ø? Ø?Ù Ù*) need two prepositions.

```
mkN3 : N -> Preposition -> Preposition -> N3 ;
```

–3 Relational common noun phrases – – In some cases, you may want to make a complex `CN` into a – relational noun (e.g. *Ø?ÙÙ ÙÙØ? Ø?ÙÙÙ ÙÙÙÙÙ ÙÙ*).

cnN2 : CN -> Preposition -> N2 ; cnN3 : CN -> Preposition -> Preposition -> N3 ;

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows

```
mkPN : Str -> Gender -> PN ;
```

– Sometimes you can reuse a common noun as a proper name, e.g. *ÙÙÙ*.

nounPN : N -> PN ;

– To form a noun phrase that can also be plural and have an irregular – genitive, you can use the worst-case function.

mkNP : Str -> Str -> Number -> Gender -> NP ;

–2 Adjectives

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Preposition -> A2 ;
```

– Comparison adjectives may two more forms.

ADeg : Type ;

mkADeg : (good,better,best,well : Str) -> ADeg ;

– The regular pattern recognizes two common variations: – *Ù* (*Ø±ÙØ?Ù* - *Ø±ÙØ?Ø±* - *Ø±ÙØ?Ø³Ø?*) and – *Ù* (*ÙÙØ©ÙÙ ÙÙØ©ÙÙØ± ÙÙØ©ÙÙØ³Ø? ÙÙØ©ÙÙÙÙ*)

regADeg : Str -> ADeg ; – long, longer, longest

– However, the duplication of the final consonant is nor predicted, – but a separate pattern is used:

duplADeg : Str -> ADeg ; – fat, fatter, fattest

– If comparison is formed by *ÙÙØ±ÙØ //most*, as in general for// – long adjective, the following pattern is used:

compoundADeg : A -> ADeg ; – -/more/most ridiculous

– From a given `ADeg`, it is possible to get back to `A`.

adegA : ADeg -> A ;

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *ÙÙÙÙÙØ³*).

```
mkAdv : Str -> Adv ;
```

```
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Prepositions

A preposition as used for rection in the lexicon, as well as to build `PP`s in the resource API, just requires a string.

```
mkPreposition : Str -> Preposition ;
mkPrep        : Str -> Prep ;
```

(These two functions are synonyms.)

Verbs

–3 Verbs with a particle. – – The particle, such as in $\emptyset^3\,\grave{U}\grave{U}\emptyset\mathbf{?}\,\emptyset^1\,\grave{U}\ \grave{U}\grave{U}$, is given as a string.

partV : V -> Str -> V ;

–3 Reflexive verbs – – By default, verbs are not reflexive; this function makes them that.

reflV : V -> V ;

–3 Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Preposition -> V2 ;

dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Preposition -> Preposition -> V3 ; -- speak, with, about
dirV3    : V -> Preposition -> V3 ;                -- give,_,to
dirdirV3 : V -> V3 ;                               -- give,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Str -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Str -> Str -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Str -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Str -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Str -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Str -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

–2 Definitions of paradigms – – The definitions should not bother the user of the API. So they are – hidden from the document. –.

Gender = MorphoAra.Gender ; Number = MorphoAra.Number ; Case = MorphoAra.Case ; human = Masc ; nonhuman = Neutr ; masculine = Masc ; feminine = Fem ; singular = Sg ; plural = Pl ; nominative = Nom ; genitive = Gen ;

```
Preposition = Str ;

mkN2 = \n,p -> n ** {lock_N2 = <> ; c2 = p} ;
```

regN2 n = mkN2 (regN n) (mkPreposition $\grave{U}\grave{U}$) ;

```
mkN3 = \n,p,q -> n ** {lock_N3 = <> ; c2 = p ; c3 = q} ;
```

cnN2 = \n,p -> n **** {lock_N2 = <> ; c2 = p} ; cnN3 = \n,p,q -> n **** {lock_N3 = <> ; c2 = p ; c3 = q} ;

mkPN n g = nameReg n g **** {lock_PN = <>} ; nounPN n = {s = n.s ! singular ; g = n.g ; lock_PN = <>} ;

```
mkNP : (_,_,_ : Str) -> PerGenNum -> NP = \ana,nI,I,pgn ->
  { s =
      table {
        Nom => ana;
        Acc => nI;
        Gen => I
      };
    a = {pgn = pgn; isPron = True };
    lock_NP = <>
  };
```

mkNP x y n g = {s = table {Gen => x ; _ => y} ; a = agrP3 n ; lock_NP = <>} ;

```
mkQuant7 : (_,_,_,_,_,_,_ : Str) -> State -> Quant =
  \hava,havihi,havAn,havayn,hAtAn,hAtayn,hA'ulA,det ->
  { s = \\n,s,g,c =>
      case <s,g,c,n> of {
        <_,Masc,_,Sg>  => hava;
        <_,Fem,_,Sg>   => havihi;
        <_,Masc,Nom,Dl>=> havAn;
        <_,Masc,_,Dl>  => havayn;
        <_,Fem,Nom,Dl> => hAtAn;
        <_,Fem,_,Dl>   => hAtayn;
        <Hum,_,_,Pl>   => hA'ulA;
        _              => havihi
      };
    d = Def;
    lock_Quant = <>
  };

mkQuant3 : (_,_,_ : Str) -> State -> Quant =
  \dalika,tilka,ula'ika,det ->
  { s = \\n,s,g,c =>
      case <s,g,c,n> of {
        <_,Masc,_,Sg>  => dalika;
        <_,Fem,_,Sg>   => tilka;
```

```
        <Hum,_,_,_>   => ula'ika;
        _              => tilka
      };
    d = Def;
    lock_Quant = <>
  };
```

mkA a b = mkAdjective a a a b **** {lock_A = <>} ; regA a = regAdjective a **** {lock_A = <>} ;

```
mkA2 a p = a ** {c2 = p ; lock_A2 = <>} ;
```

ADeg = A ; —-

mkADeg a b c d = mkAdjective a b c d **** {lock_A = <>} ;

duplADeg fat = mkADeg fat (fat + last fat + Ø±) (fat + last fat + Ø³Ø?) (fat + ÙÙ) ;

compoundADeg a = let ad = (a.s ! AAdj Posit) in mkADeg ad (ÙÙØ±Ù ++ ad) (ÙÙØ³Ø? ++ ad) (a.s ! AAdv) ;

adegA a = a ;

```
mkAdv x = ss x ** {lock_Adv = <>} ;
mkAdV x = ss x ** {lock_AdV = <>} ;
mkAdA x = ss x ** {lock_AdA = <>} ;

mkPreposition p = p ;
```

mkPrep p = ss p **** {lock_Prep = <>} ;

mkV a b c d e = mkVerb a b c d e **** {s1 = [] ; lock_V = <>} ;

partV v p = verbPart v p **** {lock_V = <>} ; reflV v = {s = v.s ; part = v.part ; lock_V = v.lock_V ; isRefl = True} ;

```
mkV2 v p = v ** {s = v.s ; c2 = p ; lock_V2 = <>} ;
dirV2 v = mkV2 v [] ;

mkV3 v p q = v ** {s = v.s ; c2 = p ; c3 = q ; lock_V3 = <>} ;
dirV3 v p = mkV3 v [] p ;
dirdirV3 v = dirV3 v [] ;

mkVS  v = v ** {lock_VS = <>} ;
```

mkVV v = { s = table {VVF vf => v.s ! vf ; _ => variants {}} ; isAux = False ; lock_VV = <> } ;

```
    mkVQ  v = v ** {lock_VQ = <>} ;

    V0 : Type = V ;
    V2S, V2V, V2Q, V2A : Type = V2 ;
    AS, A2S, AV : Type = A ;
    A2V : Type = A2 ;

    mkV0  v = v ** {lock_V = <>} ;
    mkV2S v p = mkV2 v p ** {lock_V2 = <>} ;
    mkV2V v p t = mkV2 v p ** {s4 = t ; lock_V2 = <>} ;
    mkVA  v = v ** {lock_VA = <>} ;
    mkV2A v p = mkV2 v p ** {lock_V2A = <>} ;
    mkV2Q v p = mkV2 v p ** {lock_V2 = <>} ;

    mkAS  v = v ** {lock_A = <>} ;
    mkA2S v p = mkA2 v p ** {lock_A = <>} ;
    mkAV  v = v ** {lock_A = <>} ;
    mkA2V v p = mkA2 v p ** {lock_A2 = <>} ;

  } ;
```

4.4.2 Danish

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoDan.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be

needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

```
resource ParadigmsDan =
  open
    (Predef=Predef),
    Prelude,
    CommonScand,
    ResDan,
    MorphoDan,
    CatDan in {
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  utrum   : Gender ;
  neutrum : Gender ;
```

To abstract over number names, we define the following.

```
  Number : Type ;

  singular : Number ;
  plural   : Number ;
```

To abstract over case names, we define the following.

```
  Case : Type ;

  nominative : Case ;
  genitive   : Case ;
```

Prepositions used in many-argument functions are just strings.

```
  mkPrep : Str -> Prep ;
  noPrep : Prep ;         -- empty string
```

Nouns

Worst case: give all four forms. The gender is computed from the last letter of the second form (if *n*, then `utrum`, otherwise `neutrum`).

```
mkN  : (dreng,drengen,drenger,drengene : Str) -> N ;
```

The regular function takes the singular indefinite form and computes the other forms and the gender by a heuristic. The heuristic is that all nouns are `utrum` with the plural ending *er///*r//.

```
regN : Str -> N ;
```

Giving gender manually makes the heuristic more reliable.

```
regGenN : Str -> Gender -> N ;
```

This function takes the singular indefinite and definite forms; the gender is computed from the definite form.

```
mk2N : (bil,bilen : Str) -> N ;
```

This function takes the singular indefinite and definite and the plural indefinite

```
mk3N : (bil,bilen,biler : Str) -> N ;
```

Compound nouns

All the functions above work quite as well to form compound nouns, such as *fotboll*.

Relational nouns

Relational nouns (*daughter of x*) need a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common preposition is *av*, and the following is a shortcut for regular, `nonhuman` relational nouns with *av*.

```
regN2 : Str -> Gender -> N2 ;
```

Use the function `mkPrep` or see the section on prepositions below to form other prepositions.

Three-place relational nouns (*the connection from x to y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows

```
mkPN  : Str -> Gender -> PN ;          -- Paris neutrum
regPN : Str -> PN ;                    -- utrum gender
```

Sometimes you can reuse a common noun as a proper name, e.g. *Bank*.

```
nounPN : N -> PN ;
```

To form a noun phrase that can also be plural and have an irregular genitive, you can use the worst-case function.

```
mkNP : Str -> Str -> Number -> Gender -> NP ;
```

Adjectives

Non-comparison one-place adjectives need three forms:

```
mkA : (galen,galet,galne : Str) -> A ;
```

For regular adjectives, the other forms are derived.

```
regA : Str -> A ;
```

In most cases, two forms are enough.

```
mk2A : (stor,stort : Str) -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives may need as many as five forms.

```
mkADeg : (stor,stort,store,storre,storst : Str) -> A ;
```

The regular pattern works for many adjectives, e.g. those ending with *ig*.

```
regADeg : Str -> A ;
```

Just the comparison forms can be irregular.

```
irregADeg : (tung,tyngre,tyngst : Str) -> A ;
```

Sometimes just the positive forms are irregular.

```
mk3ADeg : (galen,galet,galna : Str) -> A ;
mk2ADeg : (bred,bredt        : Str) -> A ;
```

If comparison is formed by *mer, //mest*, as in general for// long adjective, the following pattern is used:

```
compoundA : A -> A ; -- -/mer/mest norsk
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *always*).

```
mkAdv : Str -> Adv ;
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

The worst case needs six forms.

```
mkV : (spise,spiser,spises,spiste,spist,spis : Str) -> V ;
```

The 'regular verb' function is the first conjugation.

```
regV : (snakke : Str) -> V ;
```

The almost regular verb function needs the infinitive and the preteritum.

```
mk2V : (leve,levde : Str) -> V ;
```

There is an extensive list of irregular verbs in the module `IrregDan`. In practice, it is enough to give three forms, as in school books.

```
irregV : (drikke, drakk, drukket  : Str) -> V ;
```

Verbs with //være// as auxiliary

By default, the auxiliary is *have*. This function changes it to *være*.

```
vaereV : V -> V ;
```

Verbs with a particle

The particle, such as in *switch on*, is given as a string.

```
partV  : V -> Str -> V ;
```

Deponent verbs

Some words are used in passive forms only, e.g. *hoppas*, some as reflexive e.g. *ångra sig*.

```
depV  : V -> V ;
reflV : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;    -- speak, with, about
dirV3    : V -> Prep -> V3 ;           -- give,_,to
dirdirV3 : V -> V3 ;                  -- give,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
```

```
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.3 English

Aarne Ranta 2003–2005

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoEng.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

The following modules are presupposed:

```
resource ParadigmsEng = open
  (Predef=Predef),
  Prelude,
  MorphoEng,
  CatEng
  in {
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  human     : Gender ;
  nonhuman  : Gender ;
  masculine : Gender ;
```

To abstract over number names, we define the following.

```
  Number : Type ;

  singular : Number ;
  plural   : Number ;
```

To abstract over case names, we define the following.

```
  Case : Type ;

  nominative : Case ;
  genitive   : Case ;
```

Prepositions are used in many-argument functions for rection. The resource category `Prep` is used.

Nouns

Worst case: give all four forms and the semantic gender.

```
  mkN  : (man,men,man's,men's : Str) -> N ;
```

The regular function captures the variants for nouns ending with *s*,*sh*,*x*,*z* or *y*: *kiss* - *kisses*, *flash* - *flashes*; *fly* - *flies* (but *toy* - *toys*),

```
  regN : Str -> N ;
```

In practice the worst case is just: give singular and plural nominative.

```
  mk2N : (man,men : Str) -> N ;
```

All nouns created by the previous functions are marked as `nonhuman`. If you want a `human` noun, wrap it with the following function:

```
genderN : Gender -> N -> N ;
```

Compound nouns

A compound noun ia an uninflected string attached to an inflected noun, such as *baby boom*, *chief executive officer*.

```
compoundN : Str -> N -> N ;
```

Relational nouns

Relational nouns (*daughter of x*) need a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common preposition is *of*, and the following is a shortcut for regular relational nouns with *of*.

```
regN2 : Str -> N2 ;
```

Use the function `mkPrep` or see the section on prepositions below to form other prepositions.

Three-place relational nouns (*the connection from x to y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*).

```
cnN2 : CN -> Prep -> N2 ;
cnN3 : CN -> Prep -> Prep -> N3 ;
```

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows

```
regPN    : Str -> PN ;
regGenPN : Str -> Gender -> PN ;     -- John, John's
```

Sometimes you can reuse a common noun as a proper name, e.g. *Bank*.

```
nounPN : N -> PN ;
```

To form a noun phrase that can also be plural and have an irregular genitive, you can use the worst-case function.

```
mkNP : Str -> Str -> Number -> Gender -> NP ;
```

Adjectives

Non-comparison one-place adjectives need two forms: one for the adjectival and one for the adverbial form (*free* - *freely*)

```
mkA : (free,freely : Str) -> A ;
```

For regular adjectives, the adverbial form is derived. This holds even for cases with the variation *happy* - *happily*.

```
regA : Str -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives may two more forms.

```
ADeg : Type ;

mkADeg : (good,better,best,well : Str) -> ADeg ;
```

The regular pattern recognizes two common variations: *-e* (*rude* - *ruder* - *rudest*) and *-y* (*happy* - *happier* - *happiest* - *happily*)

```
regADeg : Str -> ADeg ;      -- long, longer, longest
```

However, the duplication of the final consonant is nor predicted, but a separate pattern is used:

```
duplADeg : Str -> ADeg ;      -- fat, fatter, fattest
```

If comparison is formed by *more, //most*, as in general for// long adjective, the following pattern is used:

```
compoundADeg : A -> ADeg ; -- -/more/most ridiculous
```

From a given `ADeg`, it is possible to get back to `A`.

```
adegA : ADeg -> A ;
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *always*).

```
mkAdv : Str -> Adv ;
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Prepositions

A preposition as used for rection in the lexicon, as well as to build `PP`s in the resource API, just requires a string.

```
mkPrep : Str -> Prep ;
noPrep : Prep ;
```

(These two functions are synonyms.)

Verbs

Except for *be*, the worst case needs five forms: the infinitive and the third person singular present, the past indicative, and the past and present participles.

```
mkV : (go, goes, went, gone, going : Str) -> V ;
```

The regular verb function recognizes the special cases where the last character is *y* (*cry* - *cries* but *buy* - *buys*) or *s*, *sh*, *x*, *z* (*fix* - *fixes*, etc).

```
regV : Str -> V ;
```

The following variant duplicates the last letter in the forms like *rip* - *ripped* - *ripping*.

```
regDuplV : Str -> V ;
```

There is an extensive list of irregular verbs in the module `IrregularEng`. In practice, it is enough to give three forms, e.g. *drink* - *drank* - *drunk*, with a variant indicating consonant duplication in the present participle.

```
irregV     : (drink, drank, drunk  : Str) -> V ;
irregDuplV : (get,   got,   gotten : Str) -> V ;
```

Verbs with a particle.

The particle, such as in *switch on*, is given as a string.

```
partV  : V -> Str -> V ;
```

Reflexive verbs

By default, verbs are not reflexive; this function makes them that.

```
reflV  : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;   -- speak, with, about
dirV3    : V -> Prep -> V3 ;           -- give,_,to
dirdirV3 : V -> V3 ;                   -- give,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.4 Finnish

Aarne Ranta 2003–2005

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoFin.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularFin`, which covers all irregularly inflected words.

The following modules are presupposed:

```
resource ParadigmsFin = open
  (Predef=Predef),
  Prelude,
  MorphoFin,
  CatFin
  in {
```

flags optimize=all ;

```
  flags optimize=noexpand ;
```

Parameters

To abstract over gender, number, and (some) case names, we define the following identifiers. The application programmer should always use these constants instead of their definitions in `TypesInf`.

```
oper
  Number   : Type ;

  singular : Number ;
  plural   : Number ;

  Case        : Type ;
  nominative  : Case ;
  genitive    : Case ;
```

```
partitive   : Case ;
translative : Case ;
inessive    : Case ;
elative     : Case ;
illative    : Case ;
adessive    : Case ;
ablative    : Case ;
allative    : Case ;
```

The following type is used for defining **rection**, i.e. complements of many-place verbs and adjective. A complement can be defined by just a case, or a pre/postposition and a case.

```
prePrep     : Case -> Str -> Prep ;  -- ilman, partitive
postPrep    : Case -> Str -> Prep ;  -- takana, genitive
postGenPrep :         Str -> Prep ;  -- takana
casePrep    : Case ->        Prep ;  -- adessive
```

Nouns

The worst case gives ten forms and the semantic gender. In practice just a couple of forms are needed, to define the different stems, vowel alternation, and vowel harmony.

```
oper
  mkN : (talo,   talon,   talona, taloa, taloon,
         taloina,taloissa,talojen,taloja,taloihin : Str) -> N ;
```

The regular noun heuristic takes just one form (singular nominative) and analyses it to pick the correct paradigm. It does automatic grade alternation, and is hence not usable for words like *auto* (whose genitive would become *audon*).

```
regN : (talo : Str) -> N ;
```

If `regN` does not give the correct result, one can try and give two or three forms as follows. Examples of the use of these functions are given in `BasicFin`. Most notably, `reg2N` is used for nouns like *kivi - kiviä*, which would otherwise become like *rivi - rivejä*. `regN3` is used e.g. for *sydän - sydämen - sydämiä*, which would otherwise become *sydän - sytämen*.

```
reg2N : (savi,savia : Str) -> N ;
reg3N : (vesi,veden,vesiä : Str) -> N ;
```

Some nouns have an unexpected singular partitive, e.g. *meri*, *lumi*.

```
sgpartN : (meri : N) -> (merta : Str) -> N ;
nMeri   : (meri : Str) -> N ;
```

The rest of the noun paradigms are mostly covered by the three heuristics.

Nouns with partitive *a///ä//* are a large group. To determine for grade and vowel alternation, three forms are usually needed: singular nominative and genitive, and plural partitive. Examples: *talo*, *kukko*, *huippu*, *koira*, *kukka*, *syylä*, *särki*...

```
nKukko : (kukko,kukon,kukkoja : Str) -> N ;
```

A special case are nouns with no alternations: the vowel harmony is inferred from the last letter, which must be one of *o*, *u*, *ö*, *y*.

```
nTalo : (talo : Str) -> N ;
```

Another special case are nouns where the last two consonants undergo regular weak-grade alternation: *kukko - kukon*, *rutto - ruton*, *hyppy - hypyn*, *sampo - sammon*, *kunto - kunnon*, *sisältö - sisällön*, .

```
nLukko : (lukko : Str) -> N ;
```

arpi - arven, *sappi - sapen*, *kampi - kammen*;*sylki - syljen*

```
nArpi  : (arpi : Str) -> N ;
nSylki : (sylki : Str) -> N ;
```

Foreign words ending in consonants are actually similar to words like *malli///*mallin/malleja, *with the exception that the //i* is not attached to the singular nominative. Examples: *linux*, *savett*, *screen*. The singular partitive form is used to get the vowel harmony. (N.B. more than 1-syllabic words ending in *n* would have variant plural genitive and partitive forms, like *sultanien///*sultaneiden//, which are not covered.)

```
nLinux : (linuxia : Str) -> N ;
```

Nouns of at least 3 syllables ending with *a* or *ä*, like *peruna*, *tavara*, *rytinä*.

```
nPeruna : (peruna : Str) -> N ;
```

The following paradigm covers both nouns ending in an aspirated *e*, such as *rae*, *perhe*, *savuke*, and also many ones ending in a consonant (*rengas*, *kätkyt*). The singular nominative and essive are given.

```
nRae : (rae, rakeena : Str) -> N ;
```

The following covers nouns with partitive *ta*///tä//, such as *susi*, *vesi*, *pieni*. To get all stems and the vowel harmony, it takes the singular nominative, genitive, and essive.

```
nSusi : (susi,suden,sutta : Str) -> N ;
```

Nouns ending with a long vowel, such as *puu*, *pää*, *pii*, *leikkuu*, are inflected according to the following.

```
nPuu : (puu : Str) -> N ;
```

One-syllable diphthong nouns, such as *suo*, *tie*, *työ*, are inflected by the following.

```
nSuo : (suo : Str) -> N ;
```

Many adjectives but also nouns have the nominative ending *nen* which in other cases becomes *s*: *nainen*, *ihminen*, *keltainen*. To capture the vowel harmony, we use the partitive form as the argument.

```
nNainen : (naista : Str) -> N ;
```

The following covers some nouns ending with a consonant, e.g. *tilaus*, *kaulin*, *paimen*, *laidun*.

```
nTilaus : (tilaus,tilauksena : Str) -> N ;
```

Special case:

```
nKulaus : (kulaus : Str) -> N ;
```

The following covers nouns like *nauris* and adjectives like *kallis*, *tyyris*. The partitive form is taken to get the vowel harmony.

```
nNauris : (naurista : Str) -> N ;
```

Separately-written compound nouns, like *sambal oelek*, *Urho Kekkonen*, have only their last part inflected.

```
compN : Str -> N -> N ;
```

Nouns used as functions need a case, of which by far the commonest is the genitive.

```
mkN2  : N -> Prep -> N2 ;
genN2 : N -> N2 ;

mkN3  : N -> Prep -> Prep -> N3 ;
```

Proper names can be formed by using declensions for nouns. The plural forms are filtered away by the compiler.

```
regPN : Str -> PN ;
mkPN  : N -> PN ;
mkNP  : N -> Number -> NP ;
```

Adjectives

Non-comparison one-place adjectives are just like nouns.

```
mkA : N -> A ;
```

Two-place adjectives need a case for the second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives have three forms. The comparative and the superlative are always inflected in the same way, so the nominative of them is actually enough (except for the superlative *paras* of *hyvä*).

```
mkADeg : (kiva : N) -> (kivempaa,kivinta : Str) -> A ;
```

The regular adjectives are based on `regN` in the positive.

```
regA : (punainen : Str) -> A ;
```

Verbs

The grammar does not cover the potential mood and some nominal forms. One way to see the coverage is to linearize a verb to a table. The worst case needs twelve forms, as shown in the following.

```
mkV   : (tulla,tulee,tulen,tulevat,tulkaa,tullaan,
         tuli,tulin,tulisi,tullut,tultu,tullun : Str) -> V ;
```

The following heuristics cover more and more verbs.

```
regV  : (soutaa : Str) -> V ;
reg2V : (soutaa,souti : Str) -> V ;
reg3V : (soutaa,soudan,souti : Str) -> V ;
```

The subject case of verbs is by default nominative. This dunction can change it.

```
subjcaseV : V -> Case -> V ;
```

The rest of the paradigms are special cases mostly covered by the heuristics. A simple special case is the one with just one stem and without grade alternation.

```
vValua : (valua : Str) -> V ;
```

With two forms, the following function covers a variety of verbs, such as *ottaa*, *käyttää*, *löytää*, *huoltaa*, *hiihtää*, *siirtää*.

```
vKattaa : (kattaa, katan : Str) -> V ;
```

When grade alternation is not present, just a one-form special case is needed (*poistaa*, *ryystää*).

```
vOstaa : (ostaa : Str) -> V ;
```

The following covers *juosta*, *piestä*, *nousta*, *rangaista*, *kävellä*, *surra*, *panna*.

```
vNousta : (nousta, nousen : Str) -> V ;
```

This is for one-syllable diphthong verbs like *juoda*, *syödä*.

```
vTuoda : (tuoda : Str) -> V ;
```

All the patterns above have `nominative` as subject case. If another case is wanted, use the following.

```
caseV : Case -> V -> V ;
```

The verbs *be* is special.

```
vOlla : V ;
```

Two-place verbs need a case, and can have a pre- or postposition.

```
mkV2 : V -> Prep -> V2 ;
```

If the complement needs just a case, the following special function can be used.

```
caseV2 : V -> Case -> V2 ;
```

Verbs with a direct (accusative) object are special, since their complement case is finally decided in syntax. But this is taken care of by `ClauseFin`.

```
dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;    -- speak, with, about
dirV3    : V -> Case -> V3 ;            -- give,_,to
dirdirV3 : V         -> V3 ;            -- acc, allat
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Prep -> V2V ;
mkVA  : V -> Prep -> VA ;
mkV2A : V -> Prep -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.5 French

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoFre.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

```
resource ParadigmsFre =
  open
    (Predef=Predef),
    Prelude,
    CommonRomance,
    ResFre,
    MorphoFre,
    CatFre in {

  flags optimize=all ;
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  masculine : Gender ;
  feminine  : Gender ;
```

To abstract over number names, we define the following.

```
  Number : Type ;

  singular : Number ;
  plural   : Number ;
```

Prepositions used in many-argument functions are either strings (including the 'accusative' empty string) or strings that amalgamate with the following word (the 'genitive' *de* and the 'dative' *à*).

```
accusative : Prep ;
genitive   : Prep ;
dative     : Prep ;

mkPrep : Str -> Prep ;
```

Nouns

Worst case: give both two forms and the gender.

```
mkN  : (oeil,yeux : Str) -> Gender -> N ;
```

The regular function takes the singular form, and computes the plural and the gender by a heuristic. The plural heuristic currently covers the cases *pas-pas*, *prix-prix*, *nez-nez*, *bijou-bijoux*, *cheveu-cheveux*, *plateau-plateaux*, *cheval-chevaux*. The gender heuristic is less reliable: it treats as feminine all nouns ending with *e* and *ion*, all others as masculine. If in doubt, use the `cc` command to test!

```
regN : Str -> N ;
```

Adding gender information widens the scope of the foregoing function.

```
regGenN : Str -> Gender -> N ;
```

Compound nouns

Some nouns are ones where the first part is inflected as a noun but the second part is not inflected. e.g. *numéro de téléphone*. They could be formed in syntax, but we give a shortcut here since they are frequent in lexica.

```
compN : N -> Str -> N ;
```

Relational nouns

Relational nouns (*fille de x*) need a case and a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common cases are the genitive *de* and the dative *à*, with the empty preposition.

```
deN2 : N -> N2 ;
aN2  : N -> N2 ;
```

Three-place relational nouns (*la connection de x à y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names need a string and a gender.

```
mkPN  : Str -> Gender -> PN ;    -- Jean

regPN : Str -> PN ;              -- masculine
```

To form a noun phrase that can also be plural, you can use the worst-case function.

```
mkNP : Str -> Gender -> Number -> NP ;
```

Adjectives

Non-comparison one-place adjectives need four forms in the worst case (masc and fem singular, masc plural, adverbial).

```
mkA : (banal,banale,banaux,banalement : Str) -> A ;
```

For regular adjectives, all other forms are derived from the masculine singular. The heuristic takes into account certain deviant endings: *banal- -banaux*, *chinois- -chinois*, *heureux-heureuse-heureux*, *italien-italienne*, *jeune-jeune*, *amer-amère*, *carré- - -carrément*, *joli- - -joliment*.

```
regA : Str -> A ;
```

These functions create postfix adjectives. To switch them to prefix ones (i.e. ones placed before the noun in modification, as in *petite maison*), the following function is provided.

```
prefA : A -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives

Comparison adjectives are in the worst case put up from two adjectives: the positive (*bon*), and the comparative (*meilleure*).

```
mkADeg : A -> A -> A ;
```

If comparison is formed by *plus*, as usual in French, the following pattern is used:

```
compADeg : A -> A ;
```

For prefixed adjectives, the following function is provided.

```
prefA : A -> A ;
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb.

```
mkAdv : Str -> Adv ;
```

Some appear next to the verb (e.g. *toujours*).

```
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

Irregular verbs are given in the module `VerbsFre`. If a verb should be missing in that list, the module `BeschFre` gives all the patterns of the *Bescherelle* book.

Regular verbs are ones with the infinitive *er* or *ir*, the latter with plural present indicative forms as *finissons*. The regular verb function is the first conjugation recognizes these endings, as well as the variations among *aimer, céder, placer, peser, jeter, placer, manger, assiéger, payer*.

```
regV : Str -> V ;
```

Sometimes, however, it is not predictable which variant of the *er* conjugation is to be selected. Then it is better to use the function that gives the third person singular present indicative and future ((*il*) *jette*, *jettera*) as second argument.

```
reg3V : (jeter,jette,jettera : Str) -> V ;
```

The function `regV` gives all verbs the compound auxiliary *avoir*. To change it to *être*, use the following function. Reflexive implies *être*.

```
etreV : V -> V ;
reflV : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

You can reuse a `V2` verb in `V`.

```
v2V : V2 -> V ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;  -- parler, à, de
dirV3    : V -> Prep -> V3 ;          -- donner,_,à
dirdirV3 : V -> V3 ;                  -- donner,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;  -- plain infinitive: "je veux parler"
deVV  : V -> VV ;  -- "j'essaie de parler"
aVV   : V -> VV ;  -- "j'arrive à parler"
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> Prep -> AV ;
mkA2V : A -> Prep -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.6 German

Aarne Ranta & Harald Hammarström 2003–2006

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoGer.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularGer`, which covers all irregularly inflected words.

```
resource ParadigmsGer = open
  (Predef=Predef),
  Prelude,
  MorphoGer,
  CatGer
  in {
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender    : Type ;

  masculine : Gender ;
  feminine  : Gender ;
  neuter    : Gender ;
```

To abstract over case names, we define the following.

```
  Case       : Type ;

  nominative : Case ;
  accusative : Case ;
  dative     : Case ;
  genitive   : Case ;
```

To abstract over number names, we define the following.

```
Number    : Type ;

singular  : Number ;
plural    : Number ;
```

Nouns

Worst case: give all four singular forms, two plural forms (others + dative), and the gender.

```
mkN : (x1,_,_,_,_,x6 : Str) -> Gender -> N ;
                                  -- mann, mann, manne, mannes, männer, männern
```

The regular heuristics recognizes some suffixes, from which it guesses the gender and the declension: *e, ung, ion* give the feminine with plural ending *-n, -en*, and the rest are masculines with the plural *-e* (without Umlaut).

```
regN : Str -> N ;
```

The 'almost regular' case is much like the information given in an ordinary dictionary. It takes the singular and plural nominative and the gender, and infers the other forms from these.

```
reg2N : (x1,x2 : Str) -> Gender -> N ;
```

Relational nouns need a preposition. The most common is *von* with the dative. Some prepositions are constructed in StructuralGer.

```
mkN2  : N -> Prep -> N2 ;
vonN2 : N -> N2 ;
```

Use the function `mkPrep` or see the section on prepositions below to form other prepositions.

Three-place relational nouns (*die Verbindung von x nach y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows The regular genitive is *s*, omitted after *s*.

```
mkPN  : (karolus, karoli : Str) -> PN ; -- karolus, karoli
regPN : (Johann : Str) -> PN ;          -- Johann, Johanns ; Johannes, Johannes
```

Adjectives

Adjectives need three forms, one for each degree.

```
mkA : (x1,_,x3 : Str) -> A ; -- gut,besser,beste
```

The regular adjective formation works for most cases, and includes variations such as *teuer - teurer*, *böse - böser*.

```
regA : Str -> A ;
```

Invariable adjective are a special case.

```
invarA : Str -> A ;             -- prima
```

Two-place adjectives are formed by adding a preposition to an adjective.

```
mkA2 : A -> Prep -> A2 ;
```

Adverbs

Adverbs are just strings.

```
mkAdv : Str -> Adv ;
```

Prepositions

A preposition is formed from a string and a case.

```
mkPrep : Str -> Case -> Prep ;
```

Often just a case with the empty string is enough.

```
accPrep : Prep ;
datPrep : Prep ;
genPrep : Prep ;
```

A couple of common prepositions (always with the dative).

```
von_Prep : Prep ;
zu_Prep  : Prep ;
```

Verbs

The worst-case constructor needs six forms:

- Infinitive,
- 3p sg pres. indicative,
- 2p sg imperative,
- 1/3p sg imperfect indicative,
- 1/3p sg imperfect subjunctive (because this uncommon form can have umlaut)
- the perfect participle

```
mkV : (x1,_,_,_,_,x6 : Str) -> V ;   -- geben, gibt, gib, gab, gäbe, gegeben
```

Weak verbs are sometimes called regular verbs.

```
regV : Str -> V ;                    -- führen
```

Irregular verbs use Ablaut and, in the worst cases, also Umlaut.

```
irregV : (x1,_,_,_,x5 : Str) -> V ; -- sehen, sieht, sah, sähe, gesehen
```

To remove the past participle prefix *ge*, e.g. for the verbs prefixed by *be-*, *ver-*.

```
no_geV : V -> V ;
```

To add a movable suffix e.g. *auf(fassen)*.

```
prefixV : Str -> V -> V ;
```

To change the auxiliary from *haben* (default) to *sein* and vice-versa.

```
seinV  : V -> V ;
habenV : V -> V ;
```

Reflexive verbs can take reflexive pronouns of different cases.

```
reflV  : V -> Case -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object (accusative, transitive verbs). There is also a case for dative objects.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
datV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;  -- speak, with, about
dirV3    : V -> Prep -> V3 ;                -- give,_,to
accdatV3 : V -> V3 ;                              -- give,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
```

```
mkVV  : V -> VV ;
mkV2V : V -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.7 Italian

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoIta.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

```
resource ParadigmsIta =
```

```
  open
    (Predef=Predef),
    Prelude,
    CommonRomance,
    ResIta,
    MorphoIta,
    BeschIta,
    CatIta in {

  flags optimize=all ;
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  masculine : Gender ;
  feminine  : Gender ;
```

To abstract over number names, we define the following.

```
  Number : Type ;

  singular : Number ;
  plural   : Number ;
```

Prepositions used in many-argument functions are either strings (including the 'accusative' empty string) or strings that amalgamate with the following word (the 'genitive' *de* and the 'dative' *à*).

```
  Prep : Type ;

  accusative : Prep ;
  genitive   : Prep ;
  dative     : Prep ;

  mkPrep : Str -> Prep ;
```

Nouns

Worst case: give both two forms and the gender.

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

The regular function takes the singular form and the gender, and computes the plural and the gender by a heuristic. The heuristic says that the gender is feminine for nouns ending with *a*, and masculine for all other words.

```
regN : Str -> N ;
```

To force a different gender, use one of the following functions.

```
mascN : N -> N ;
femN  : N -> N ;
```

Compound nouns

Some nouns are ones where the first part is inflected as a noun but the second part is not inflected. e.g. *numéro de téléphone*. They could be formed in syntax, but we give a shortcut here since they are frequent in lexica.

```
compN : N -> Str -> N ;
```

Relational nouns

Relational nouns (*figlio di x*) need a case and a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common cases are the genitive *di* and the dative *a*, with the empty preposition.

```
diN2 : N -> N2 ;
aN2  : N -> N2 ;
```

Three-place relational nouns (*la connessione di x a y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names need a string and a gender.

```
mkPN  : Str -> Gender -> PN ;
regPN : Str -> PN ;              -- masculine
```

To form a noun phrase that can also be plural, you can use the worst-case function.

```
mkNP : Str -> Gender -> Number -> NP ;
```

Adjectives

Non-comparison one-place adjectives need five forms in the worst case (masc and fem singular, masc plural, adverbial).

```
mkA : (solo,sola,soli,sole, solamente : Str) -> A ;
```

For regular adjectives, all other forms are derived from the masculine singular.

```
regA : Str -> A ;
```

These functions create postfix adjectives. To switch them to prefix ones (i.e. ones placed before the noun in modification, as in *petite maison*), the following function is provided.

```
prefA : A -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives

Comparison adjectives are in the worst case put up from two adjectives: the positive (*buono*), and the comparative (*migliore*).

```
mkADeg : A -> A -> A ;
```

If comparison is formed by *più*, as usual in Italian, the following pattern is used:

```
compADeg : A -> A ;
```

The regular pattern is the same as `regA` for plain adjectives, with comparison by *plus*.

```
regADeg : Str -> A ;
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb.

```
mkAdv : Str -> Adv ;
```

Some appear next to the verb (e.g. *sempre*).

```
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

Regular verbs are ones with the infinitive *er* or *ir*, the latter with plural present indicative forms as *finissons*. The regular verb function is the first conjugation recognizes these endings, as well as the variations among *aimer, céder, placer, peser, jeter, placer, manger, assiéger, payer*.

```
regV : Str -> V ;
```

The module `BeschIta` gives all the patterns of the *Bescherelle* book. To use them in the category `V`, wrap them with the function

```
verboV : Verbo -> V ;
```

The function `regV` gives all verbs the compound auxiliary *avere*. To change it to *essere*, use the following function. Reflexive implies *essere*.

```
essereV : V -> V ;
reflV : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

You can reuse a `V2` verb in `V`.

```
v2V : V2 -> V ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ; -- parlare, a, di
dirV3    : V -> Prep -> V3 ;         -- dare,_,a
dirdirV3 : V -> V3 ;                 -- dare,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
```

```
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;  -- plain infinitive: "voglio parlare"
deVV  : V -> VV ;  -- "cerco di parlare"
aVV   : V -> VV ;  -- "arrivo a parlare"
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> Prep -> AV ;
mkA2V : A -> Prep -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.8 Norwegian

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoNor.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

```
resource ParadigmsNor =
  open
    (Predef=Predef),
    Prelude,
    CommonScand,
    ResNor,
    MorphoNor,
    CatNor in {
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  masculine : Gender ;
  feminine  : Gender ;
  neutrum   : Gender ;
```

To abstract over number names, we define the following.

```
  Number : Type ;

  singular : Number ;
  plural   : Number ;
```

To abstract over case names, we define the following.

```
  Case : Type ;

  nominative : Case ;
  genitive   : Case ;
```

Prepositions used in many-argument functions are just strings.

```
  mkPrep : Str -> Prep ;
  noPrep : Prep ;        -- empty string
```

Nouns

Worst case: give all four forms. The gender is computed from the last letter of the second form (if n, then `utrum`, otherwise `neutrum`).

```
mkN  : (dreng,drengen,drenger,drengene : Str) -> N ;
```

The regular function takes the singular indefinite form and computes the other forms and the gender by a heuristic. The heuristic is that nouns ending *e* are feminine like *kvinne*, all others are masculine like *bil*. If in doubt, use the `cc` command to test!

```
regN : Str -> N ;
```

Giving gender manually makes the heuristic more reliable.

```
regGenN : Str -> Gender -> N ;
```

This function takes the singular indefinite and definite forms; the gender is computed from the definite form.

```
mk2N : (bil,bilen : Str) -> N ;
```

Compound nouns

All the functions above work quite as well to form compound nouns, such as *fotboll*.

Relational nouns

Relational nouns (*daughter of x*) need a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common preposition is *av*, and the following is a shortcut for regular, `nonhuman` relational nouns with *av*.

```
regN2 : Str -> Gender -> N2 ;
```

Use the function `mkPrep` or see the section on prepositions below to form other prepositions.

Three-place relational nouns (*the connection from x to y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows

```
regPN    : Str -> PN ;             -- utrum
regGenPN : Str -> Gender -> PN ;
```

Sometimes you can reuse a common noun as a proper name, e.g. *Bank*.

```
nounPN : N -> PN ;
```

To form a noun phrase that can also be plural and have an irregular genitive, you can use the worst-case function.

```
mkNP : Str -> Str -> Number -> Gender -> NP ;
```

Adjectives

Non-comparison one-place adjectives need three forms:

```
mkA : (galen,galet,galne : Str) -> A ;
```

For regular adjectives, the other forms are derived.

```
regA : Str -> A ;
```

In most cases, two forms are enough.

```
mk2A : (stor,stort : Str) -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives may need as many as five forms.

```
mkADeg : (stor,stort,store,storre,storst : Str) -> A ;
```

The regular pattern works for many adjectives, e.g. those ending with *ig*.

```
regADeg : Str -> A ;
```

Just the comparison forms can be irregular.

```
irregADeg : (tung,tyngre,tyngst : Str) -> A ;
```

Sometimes just the positive forms are irregular.

```
mk3ADeg : (galen,galet,galna : Str) -> A ;
mk2ADeg : (bred,bredt        : Str) -> A ;
```

If comparison is formed by *mer, //mest*, as in general for// long adjective, the following pattern is used:

```
compoundA : A -> A ; -- -/mer/mest norsk
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *always*).

```
mkAdv : Str -> Adv ;
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

The worst case needs six forms.

```
mkV : (spise,spiser,spises,spiste,spist,spis : Str) -> V ;
```

The 'regular verb' function is the first conjugation.

```
regV : (snakke : Str) -> V ;
```

The almost regular verb function needs the infinitive and the preteritum.

```
mk2V : (leve,levde : Str) -> V ;
```

There is an extensive list of irregular verbs in the module `IrregNor`. In practice, it is enough to give three forms, as in school books.

```
irregV : (drikke, drakk, drukket  : Str) -> V ;
```

Verbs with //være// as auxiliary

By default, the auxiliary is *have*. This function changes it to *være*.

```
vaereV : V -> V ;
```

Verbs with a particle.

The particle, such as in *switch on*, is given as a string.

```
partV  : V -> Str -> V ;
```

Deponent verbs.

Some words are used in passive forms only, e.g. *hoppas*, some as reflexive e.g. *ångra sig*.

```
depV  : V -> V ;
reflV : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;    -- speak, with, about
dirV3    : V -> Prep -> V3 ;            -- give,_,to
dirdirV3 : V -> V3 ;                    -- give,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

4.4.9 Russian

Janna Khegai 2003–2005

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoEng.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

The following modules are presupposed:

```
resource ParadigmsRus = open
  (Predef=Predef),
  Prelude,
  MorphoRus,
  CatRus,
  NounRus
  in {

flags  coding=utf8 ;
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;
  masculine : Gender ;
  feminine  : Gender ;
  neuter    : Gender ;
```

To abstract over case names, we define the following.

```
  Case : Type ;
```

```
nominative    : Case ;
genitive      : Case ;
dative        : Case ;
accusative    : Case ;
instructive   : Case ;
prepositional : Case ;
```

In some (written in English) textbooks accusative case is put on the second place. However, we follow the case order standard for Russian textbooks. To abstract over number names, we define the following.

```
Number : Type ;

singular : Number ;
plural   : Number ;
```

Nouns

Best case: indeclinabe nouns: Đ?Đ$\frac{3}{4}$ÑĐ?, Đ¿Đ?Đ>>ÑÑĐ$\frac{3}{4}$, ĐĐ£Đ.

```
Animacy: Type ;

animate: Animacy;
inanimate: Animacy;

 mkIndeclinableNoun: Str -> Gender -> Animacy -> N ;
```

Worst case - give six singular forms: Nominative, Genetive, Dative, Accusative, Instructive and Prepositional; corresponding six plural forms and the gender. May be the number of forms needed can be reduced, but this requires a separate investigation. Animacy parameter (determining whether the Accusative form is equal to the Nominative or the Genetive one) is actually of no help, since there are a lot of exceptions and the gain is just one form less.

```
mkN  : (_,_,_,_,_,_,_,_,_,_,_,_ : Str) -> Gender -> Animacy -> N ;

   -- Đ¼ÑĐ¶ÑĐ¸Đ½Đ?, Đ¼ÑĐ¶ÑĐ¸Đ½Ñ, Đ¼ÑĐ¶ÑĐ¸Đ½Đ?, Đ¼ÑĐ¶ÑĐ¸Đ½Ñ, Đ¼ÑĐ¶ÑĐ¸Đ½Đ¾Đ¹, Đ¼ÑĐ
   -- Đ¼ÑĐ¶ÑĐ¸Đ½Ñ, Đ¼ÑĐ¶ÑĐ¸Đ½Đ¾, Đ¼ÑĐ¶ÑĐ¸Đ½Đ?Đ¼, Đ¼ÑĐ¶ÑĐ¸Đ½Đ¾, Đ¼ÑĐ¶ÑĐ¸Đ½Đ?Đ¼Đ¸, Đ¼Ñ
```

The regular function captures the variants for some popular nouns endings below:

```
regN : Str -> N ;
```

Here are some common patterns. The list is far from complete. Feminine patterns.

```
nMashina   : Str -> N ;    -- feminine, inanimate, ending with "-Đ?", Inst -"Đ¼Đ?Ñ
nEdinica   : Str -> N ;    -- feminine, inanimate, ending with "-Đ?", Inst -"Đ?Đ´Đ
nZhenchina : Str -> N ;    -- feminine, animate, ending with "-a"
nNoga      : Str -> N ;    -- feminine, inanimate, ending with "Đ³_Đ?_Ñ-a"
nMalyariya  : Str -> N ;    -- feminine, inanimate, ending with "-Đ¬Ñ"
nTetya     : Str -> N ;    -- feminine, animate, ending with "-Ñ"
nBol       : Str -> N ;    -- feminine, inanimate, ending with "-Ñ"(soft sign)
```

Neuter patterns.

```
nObezbolivauchee : Str -> N ;   -- neutral, inanimate, ending with "-ee"
nProizvedenie : Str -> N ;   -- neutral, inanimate, ending with "-e"
nChislo : Str -> N ;   -- neutral, inanimate, ending with "-o"
nZhivotnoe : Str -> N ;    -- masculine, inanimate, ending with "-Đ?Đ½Ñ"
```

Masculine patterns. Ending with consonant:

```
nPepel : Str -> N ;    -- masculine, inanimate, ending with "-Đ?Đ>>"- "Đ¿Đ?Đ¿-Đ>>Đ

nBrat: Str -> N ;   -- animate, Đ±ÑĐ?Ñ-ÑÑ
nStul: Str -> N ;    -- same as above, but inanimate
nMalush : Str -> N ; -- Đ¼Đ?Đ>>ÑÑĐ?Đ¹
nPotolok : Str -> N ; -- Đ¿Đ¾ÑĐ¾Đ>>-Đ¾Đ? - Đ¿Đ¾ÑĐ¾Đ>>-Đ?Đ?

-- the next four differ in plural nominative and/or accusative form(s) :
nBank: Str -> N ;    -- Đ±Đ?Đ½Đ?-Đ¬ (Nom=Acc)
nStomatolog : Str -> N ;  -- same as above, but animate
nAdres     : Str -> N ;     -- Đ?Đ´ÑĐ?Ñ-Đ? (Nom=Acc)
nTelefon   : Str -> N ;     -- ÑĐ?Đ>>Đ?ÑĐ¾Đ½-Ñ (Nom=Acc)

nNol       : Str -> N ;    -- masculine, inanimate, ending with "-Ñ" (soft sign)
nUroven    : Str -> N ;    -- masculine, inanimate, ending with "-Đ?Đ½Ñ"
```

Nouns used as functions need a preposition. The most common is with Genitive.

```
mkFun  : N -> Prep -> N2 ;
mkN2 : N -> N2 ;
mkN3 : N -> Prep -> Prep -> N3 ;
```

Proper names.

```
mkPN  : Str -> Gender -> Animacy -> PN ;          -- "ÐÐ²Ð?Ð½Ð¸Ñ", "ÐÐ?ÑÐ?"
regPN : Str -> PN ;
nounPN : N -> PN ;
```

On the top level, it is maybe `CN` that is used rather than `N`, and `NP` rather than `PN`.

```
mkCN  : N -> CN ;
mkNP  : Str -> Gender -> Animacy -> NP ;
```

Adjectives

Non-comparison (only positive degree) one-place adjectives need 28 (4 by 7) forms in the worst case: Masculine | Feminine | Neutral | Plural Nominative Genitive Dative Accusative Inanimate Accusative Animate Instructive Prepositional Notice that 4 short forms, which exist for some adjectives are not included in the current description, otherwise there would be 32 forms for positive degree. mkA : (: Str) -> A ; The regular function captures the variants for some popular adjective endings below:

```
regA : Str -> Str -> A ;
```

Invariable adjective is a special case.

```
adjInvar : Str -> A ;          -- khaki, mini, hindi, netto
```

Some regular patterns depending on the ending.

```
AStaruyj : Str -> Str -> A ;              -- ending with "-ÑÐ¹"
AMalenkij : Str -> Str -> A ;             -- ending with "-Ð¸Ð¹", Gen - "Ð¾Ð?Ð>>Ð
AKhoroshij : Str -> Str -> A ;          --  ending with "-Ð¾Ð¹", Gen - "ÑÐ¾Ð¾Ñ-Ð?
  AMolodoj : Str -> Str -> A ;             -- ending with "-Ð¾Ð¹",
                                          -- plural - Ð¾Ð¾Ð>>Ð¾Ð¾-ÑÐ?"
AKakoj_Nibud : Str -> Str -> Str -> A ;   -- ending with "-Ð¾Ð¹",
                                          -- plural - "Ð?Ð?Ð?-Ð¸Ð?"
```

Two-place adjectives need a preposition and a case as extra arguments.

```
mkA2 : A -> Str -> Case -> A2 ;  -- "Đ՞Đ؟Đ>>Đ˗Đ¼ Đ½Đ؟"
```

Comparison adjectives need a positive adjective (28 forms without short forms). Taking only one comparative form (non-syntaxic) and only one superlative form (syntaxic) we can produce the comparison adjective with only one extra argument - non-syntaxic comparative form. Syntaxic forms are based on the positive forms. mkADeg : A -> Str -> ADeg ; On top level, there are adjectival phrases. The most common case is just to use a one-place adjective. ap : A -> IsPostfixAdj -> AP ;

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *always*).

```
mkAdv : Str -> Adv ;
```

Verbs

In our lexicon description (*Verbum*) there are 62 forms: 2 (Voice) by { 1 (infinitive) + [2(number) by 3 (person)](imperative) + [[2(Number) by 3(Person)](present) + [2(Number) by 3(Person)](future) + 4(GenNum)(past)](indicative)+ 4 (GenNum) (subjunctive) } Participles (Present and Past) and Gerund forms are not included, since they fuction more like Adjectives and Adverbs correspondingly rather than verbs. Aspect regarded as an inherent parameter of a verb. Notice, that some forms are never used for some verbs. Actually, the majority of verbs do not have many of the forms.

```
Voice: Type;
Aspect: Type;
```

Tense : Type;

```
Bool: Type;
Conjugation: Type ;

first: Conjugation; -- "Đ³ÑĐ>>Ñ-ĐÑÑ, Đ³ÑĐ>>Ñ-ĐĐ¼"
firstE: Conjugation; -- Verbs with vowel "Ñ": "Đ՞Đ؟ÑÑÑ" (give), "Đ¿ÑÑÑÑ" (drink)
second: Conjugation; -- "Đ²Đ˗Đ՞-ĐÑÑ, Đ²Đ˗Đ՞-ĐĐ¼"
mixed: Conjugation; -- "ÑĐ¾Ñ-ĐÑÑ - ÑĐ¾Ñ-ĐĐ¼"
```

```
dolzhen: Conjugation; -- irregular

true: Bool;
false: Bool;

active: Voice ;
passive: Voice ;
imperfective: Aspect;
perfective: Aspect ;
```

present : Tense ; past : Tense ; The worst case need 6 forms of the present tense in indicative mood (Ñ Ð±Ð?Ð³Ñ, ÑÑ Ð±Ð?Ð¶Ð¸Ñ, Ð¾Ð½ Ð±Ð?Ð¶Ð¸Ð¼, Ð¼Ñ Ð±Ð?Ð¶Ð¸Ð¼, Ð²Ñ Ð±Ð?Ð¶Ð¸Ñ, Ð²Ð¾Ð½Ð¸ Ð±Ð?Ð³Ñ), a past form (singular, masculine: Ñ Ð±Ð?Ð¶Ð?Ð>>), an imperative form (singular, second person: Ð±Ð?Ð³Ð¸), an infinitive (Ð±Ð?Ð¶Ð?ÑÑ). Inherent aspect should also be specified.

```
mkVerbum : Aspect -> (_,_,_,_,_,_,_,_,_ : Str) -> V ;
```

Common conjugation patterns are two conjugations: first - verbs ending with -Ð?ÑÑ/-ÑÑÑ and second - -Ð¸ÑÑ/-Ð?ÑÑ. Instead of 6 present forms of the worst case, we only need a present stem and one ending (singular, first person): Ñ Ð>>ÑÐ±-Ð>>Ñ, Ñ Ð¶Ð´-Ñ, etc. To determine where the border between stem and ending lies it is sufficient to compare first person from with second person form: Ñ Ð>>ÑÐ±-Ð>>Ñ, ÑÑ Ð>>ÑÐ±-Ð¸ÑÑ. Stems shoud be the same. So the definition for verb Ð>>ÑÐ±Ð¸ÑÑ looks like: regV Imperfective Second Ð>>ÑÐ± Ð>>Ñ Ð>>ÑÐ±Ð¸Ð>> Ð>>ÑÐ±Ð¸ Ð>>ÑÐ±Ð¸ÑÑ;

```
regV :Aspect -> Conjugation -> (_,_,_,_,_ : Str) -> V ;
```

For writing an application grammar one usualy doesn't need the whole inflection table, since each verb is used in a particular context that determines some of the parameters (Tense and Voice while Aspect is fixed from the beginning) for certain usage. The *V* type, that have these parameters fixed. We can extract the *V* from the lexicon. mkV: Verbum -> Voice -> V ; mkPresentV: Verbum -> Voice -> V ; Two-place verbs, and the special case with direct object. Notice that a particle can be included in a `V`.

```
mkV2     : V   -> Str -> Case -> V2 ;   -- "Ð²Ð¾Ð¹Ð½Ð¸ Ð² Ð¼Ð¾Ð¹Ð½Ñ"; "Ð²", accusative
mkV3  : V -> Str -> Str -> Case -> Case -> V3 ; -- "ÑÐ>>Ð¾Ð¶Ð¸Ñ Ð¿Ð¸ÑÑÐ¼Ð¾ Ð¼Ð°Ð¼Ðµ Ð
dirV2    : V -> V2 ;                    -- "Ð²Ð¸Ð´Ð?ÑÑ", "Ð>>ÑÐ±Ð¸ÑÑ"
tvDirDir : V -> V3 ;
```

4.4.10 Spanish

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoSpa.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`.

```
resource ParadigmsSpa =
  open
    (Predef=Predef),
    Prelude,
    CommonRomance,
    ResSpa,
    MorphoSpa,
    BeschSpa,
    CatSpa in {

  flags optimize=all ;
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  masculine : Gender ;
  feminine  : Gender ;
```

To abstract over number names, we define the following.

```
Number : Type ;

singular : Number ;
plural   : Number ;
```

Prepositions used in many-argument functions are either strings (including the 'accusative' empty string) or strings that amalgamate with the following word (the 'genitive' *de* and the 'dative' *à*).

```
Prep : Type ;

accusative : Prep ;
genitive   : Prep ;
dative     : Prep ;

mkPrep : Str -> Prep ;
```

Nouns

Worst case: two forms (singular + plural), and the gender.

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

The regular function takes the singular form and the gender, and computes the plural and the gender by a heuristic. The heuristic says that the gender is feminine for nouns ending with *a* or *z*, and masculine for all other words. Nouns ending with *a*, *o*, *e* have the plural with *s*, those ending with *z* have *ces* in plural; all other nouns have *es* as plural ending. The accent is not dealt with.

```
regN : Str -> N ;
```

To force a different gender, use one of the following functions.

```
mascN : N -> N ;
femN  : N -> N ;
```

Compound nouns

Some nouns are ones where the first part is inflected as a noun but the second part is not inflected. e.g. *numéro de téléphone*. They could be formed in syntax, but we give a shortcut here since they are frequent in lexica.

```
compN : N -> Str -> N ;
```

Relational nouns

Relational nouns (*fille de x*) need a case and a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common cases are the genitive *de* and the dative *a*, with the empty preposition.

```
deN2 : N -> N2 ;
aN2  : N -> N2 ;
```

Three-place relational nouns (*la connessione di x a y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names need a string and a gender.

```
mkPN  : Str -> Gender -> PN ;          -- Jean
regPN : Str -> PN ;                    -- masculine
```

To form a noun phrase that can also be plural, you can use the worst-case function.

```
mkNP : Str -> Gender -> Number -> NP ;
```

Adjectives

Non-comparison one-place adjectives need five forms in the worst case (masc and fem singular, masc plural, adverbial).

```
mkA : (solo,sola,soli,sole, solamente : Str) -> A ;
```

For regular adjectives, all other forms are derived from the masculine singular. The types of adjectives that are recognized are *alto*, *fuerte*, *util*.

```
regA : Str -> A ;
```

These functions create postfix adjectives. To switch them to prefix ones (i.e. ones placed before the noun in modification, as in *petite maison*), the following function is provided.

```
prefA : A -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Comparison adjectives

Comparison adjectives are in the worst case put up from two adjectives: the positive (*bueno*), and the comparative (*mejor*).

```
mkADeg : A -> A -> A ;
```

If comparison is formed by *mas*, as usual in Spanish, the following pattern is used:

```
compADeg : A -> A ;
```

The regular pattern is the same as `regA` for plain adjectives, with comparison by *mas*.

```
regADeg : Str -> A ;
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb.

```
mkAdv : Str -> Adv ;
```

Some appear next to the verb (e.g. *siempre*).

```
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

Regular verbs are ones inflected like *cortar*, *deber*, or *vivir*. The regular verb function is the first conjugation (*ar*) recognizes the variations corresponding to the patterns *actuar, cazar, guiar, pagar, sacar*. The module `BeschSpa` gives the complete set of *Bescherelle* conjugations.

```
regV : Str -> V ;
```

The module `BeschSpa` gives all the patterns of the *Bescherelle* book. To use them in the category `V`, wrap them with the function

```
verboV : Verbum -> V ;
```

To form reflexive verbs:

```
reflV : V -> V ;
```

Verbs with a deviant passive participle: just give the participle in masculine singular form as second argument.

```
special_ppV : V -> Str -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

You can reuse a `V2` verb in `V`.

```
v2V : V2 -> V ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ;   -- parler, à, de
dirV3    : V -> Prep -> V3 ;          -- donner,_,à
dirdirV3 : V -> V3 ;                  -- donner,_,_
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;  -- plain infinitive: "je veux parler"
deVV  : V -> VV ;  -- "j'essaie de parler"
aVV   : V -> VV ;  -- "j'arrive à parler"
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;

mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> Prep -> AV ;
mkA2V : A -> Prep -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2Q : Type ;
AS, A2S, AV, A2V  : Type ;
```

4.4.11 Swedish

Aarne Ranta 2003

This is an API to the user of the resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural.gf`.

The main difference with `MorphoSwe.gf` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`. However, this function should only seldom be needed: we have a separate module `IrregularEng`, which covers all irregularly inflected words.

```
resource ParadigmsSwe =
  open
    (Predef=Predef),
    Prelude,
    CommonScand,
    ResSwe,
    MorphoSwe,
    CatSwe in {
```

Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;

  utrum     : Gender ;
  neutrum   : Gender ;
```

To abstract over number names, we define the following.

```
Number : Type ;

singular : Number ;
plural   : Number ;
```

To abstract over case names, we define the following.

```
Case : Type ;

nominative : Case ;
genitive   : Case ;
```

Prepositions used in many-argument functions are just strings.

```
mkPrep : Str -> Prep ;
noPrep : Prep ;         -- empty string
```

Nouns

Worst case: give all four forms. The gender is computed from the last letter of the second form (if *n*, then `utrum`, otherwise `neutrum`).

```
mkN  : (apa,apan,apor,aporna : Str) -> N ;
```

The regular function takes the singular indefinite form and computes the other forms and the gender by a heuristic. The heuristic is currently to treat all words ending with *a* like *flicka*, with *e* like *rike*, and otherwise like *bil*. If in doubt, use the `cc` command to test!

```
regN : Str -> N ;
```

Adding the gender manually greatly improves the correction of `regN`.

```
regGenN : Str -> Gender -> N ;
```

In practice the worst case is often just: give singular and plural indefinite.

```
mk2N : (nyckel,nycklar : Str) -> N ;
```

This heuristic takes just the plural definite form and infers the others. It does not work if there are changes in the stem.

```
mk1N : (bilarna : Str) -> N ;
```

Compound nouns

All the functions above work quite as well to form compound nouns, such as *fotboll*.

Relational nouns

Relational nouns (*daughter of x*) need a preposition.

```
mkN2 : N -> Prep -> N2 ;
```

The most common preposition is *av*, and the following is a shortcut for regular, `nonhuman` relational nouns with *av*.

```
regN2 : Str -> Gender -> N2 ;
```

Use the function `mkPreposition` or see the section on prepositions below to form other prepositions.

Three-place relational nouns (*the connection from x to y*) need two prepositions.

```
mkN3 : N -> Prep -> Prep -> N3 ;
```

Relational common noun phrases

In some cases, you may want to make a complex `CN` into a relational noun (e.g. *the old town hall of*). However, `N2` and `N3` are purely lexical categories. But you can use the `AdvCN` and `PrepNP` constructions to build phrases like this.

Proper names and noun phrases

Proper names, with a regular genitive, are formed as follows

```
regGenPN : Str -> Gender -> PN ;
regPN    : Str -> PN ;            -- utrum
```

Sometimes you can reuse a common noun as a proper name, e.g. *Bank*.

```
nounPN : N -> PN ;
```

To form a noun phrase that can also be plural and have an irregular genitive, you can use the worst-case function.

```
mkNP : Str -> Str -> Number -> Gender -> NP ;
```

Adjectives

Adjectives may need as many as seven forms.

```
mkA : (liten, litet, lilla, sma, mindre, minst, minsta : Str) -> A ;
```

The regular pattern works for many adjectives, e.g. those ending with *ig*.

```
regA : Str -> A ;
```

Just the comparison forms can be irregular.

```
irregA : (tung,tyngre,tyngst : Str) -> A ;
```

Sometimes just the positive forms are irregular.

```
mk3A : (galen,galet,galna : Str) -> A ;
mk2A : (bred,brett        : Str) -> A ;
```

Comparison forms may be compound (*mera svensk* - *mest svensk*).

```
compoundA : A -> A ;
```

Two-place adjectives

Two-place adjectives need a preposition for their second argument.

```
mkA2 : A -> Prep -> A2 ;
```

Adverbs

Adverbs are not inflected. Most lexical ones have position after the verb. Some can be preverbal (e.g. *always*).

```
mkAdv : Str -> Adv ;
mkAdV : Str -> AdV ;
```

Adverbs modifying adjectives and sentences can also be formed.

```
mkAdA : Str -> AdA ;
```

Verbs

The worst case needs five forms.

```
mkV : (supa,super,sup,söp,supit,supen : Str) -> V ;
```

The 'regular verb' function is inspired by Lexin. It uses the present tense indicative form. The value is the first conjugation if the argument ends with *ar* (*tala - talar - talade - talat*), the second with *er* (*leka - leker - lekte - lekt*, with the variations like *gräva*, *vända*, *tyda*, *hyra*), and the third in other cases (*bo - bor - bodde - bott*).

```
regV : (talar : Str) -> V ;
```

The almost regular verb function needs the infinitive and the preteritum. It is not really more powerful than the new implementation of `regV` based on the indicative form.

```
mk2V : (leka,lekte : Str) -> V ;
```

There is an extensive list of irregular verbs in the module `IrregularSwe`. In practice, it is enough to give three forms, as in school books.

```
irregV : (dricka, drack, druckit : Str) -> V ;
```

Verbs with a particle.

The particle, such as in *passa på*, is given as a string.

```
partV  : V -> Str -> V ;
```

Deponent verbs.

Some words are used in passive forms only, e.g. *hoppas*, some as reflexive e.g. *ångra sig*.

```
depV  : V -> V ;
reflV : V -> V ;
```

Two-place verbs

Two-place verbs need a preposition, except the special case with direct object. (transitive verbs). Notice that a particle comes from the `V`.

```
mkV2  : V -> Prep -> V2 ;

dirV2 : V -> V2 ;
```

Three-place verbs

Three-place (ditransitive) verbs need two prepositions, of which the first one or both can be absent.

```
mkV3     : V -> Prep -> Prep -> V3 ; -- tala med om
dirV3    : V -> Prep -> V3 ;                -- ge _ till
dirdirV3 : V -> V3 ;                            -- ge _ _
```

Other complement patterns

Verbs and adjectives can take complements such as sentences, questions, verb phrases, and adjectives.

```
mkV0  : V -> V0 ;
mkVS  : V -> VS ;
mkV2S : V -> Prep -> V2S ;
mkVV  : V -> VV ;
mkV2V : V -> Prep -> Prep -> V2V ;
mkVA  : V -> VA ;
mkV2A : V -> Prep -> V2A ;
mkVQ  : V -> VQ ;
mkV2Q : V -> Prep -> V2Q ;
```

```
mkAS  : A -> AS ;
mkA2S : A -> Prep -> A2S ;
mkAV  : A -> AV ;
mkA2V : A -> Prep -> A2V ;
```

Notice: categories `V2S, V2V, V2A, V2Q` are in v 1.0 treated just as synonyms of `V2`, and the second argument is given as an adverb. Likewise `AS, A2S, AV, A2V` are just `A`. `V0` is just `V`.

```
V0, V2S, V2V, V2A, V2Q : Type ;
AS, A2S, AV, A2V : Type ;
```

5 Summary of Categories and Functions

These tables show all categories and functions in `Grammar`, except the functions in `Structural`. All example strings can be parsed in `LangEng`.

5.1 Categories

Category	Module	Explanation	Example
A2	Cat	two place adjective	"married"
A	Cat	one place adjective	"old"
AdA	Common	adjective modifying adverb,	"very"
AdN	Common	numeral modifying adverb,	"more than"
AdV	Common	adverb directly attached to verb	"always"
Adv	Common	verb phrase modifying adverb,	"in the house"
Ant	Common	anteriority	simultaneous
AP	Cat	adjectival phrase	"very old"
CAdv	Common	comparative adverb	"more"
Cl	Cat	declarative clause, with all tenses	"she walks"
CN	Cat	common noun (without determiner)	"red house"
Comp	Cat	complement of copula, such as AP	"very warm"
Conj	Cat	conjunction,	"and"
DConj	Cat	distributed conj.	"both" - "and"
Det	Cat	determiner phrase	"these seven"
Digit	Numeral	digit from 2 to 9	"4"
IAdv	Common	interrogative adverb	"why"
IComp	Cat	interrogative complement of copula	"where"
IDet	Cat	interrogative determiner	"which"
Imp	Cat	imperative	"look at this"
IP	Cat	interrogative pronoun	"who"
N2	Cat	relational noun	"brother"
N3	Cat	three place relational noun	"connection"
N	Cat	common noun	"house"
NP	Cat	noun phrase (subject or object)	"the red house"
Num	Cat	cardinal number (used with QuantPl)	"seven"
Numeral	Cat	cardinal or ordinal,	"five" / "fifth"
Ord	Cat	ordinal number (used in Det)	"seventh"
PConj	Common	phrase beginning conj.	"therefore"
Phr	Common	phrase in a text	"but look at this please"
PN	Cat	proper name	"Paris"
Pol	Common	polarity	positive
Predet	Cat	predeterminer (prefixed Quant)	"all"
Prep	Cat	preposition, or just case	"in"
Pron	Cat	personal pronoun	"she"
QCl	Cat	question clause, with all tenses	"why does she walk"
QS	Cat	question	"where did she walk"
Quant	Cat	quantifier with both sg and pl	"this"/"these"
QuantPl	Cat	quantifier ('nucleus' of plur. Det)	"many"
QuantSg	Cat	quantifier ('nucleus' of sing. Det)	"every"
RCl	Cat	relative clause, with all tenses	"in which she walks"
RP	Cat	relative pronoun	"in which"
RS	Cat	relative	"that she loves"
S	Cat	declarative sentence	"she was here"
SC	Common	embedded sentence or question	"that it rains"
Slash	Cat	clause missing NP (S/NP in GPSG)	"she loves"

Category	Module	Explanation	Example
Sub10	Numeral	numeral under 10	"9"
Sub100	Numeral	numeral under 100	"99"
Sub1000	Numeral	numeral under 1000	"999"
Sub1000000	Numeral	numeral under million	123456
Subj	Cat	subjunction,	"if"
Tense	Common	tense	present
Text	Common	text consisting of several phrases	"He is here. Why?"
Utt	Common	sentence, question, word...	"be quiet"
V2A	Cat	verb with NP and AP complement	"paint"
V2	Cat	two place verb	"love"
V3	Cat	three place verb	"show"
VA	Cat	adjective complement verb	"look"
V	Cat	one place verb	"sleep"
Voc	Common	vocative or	"please" "my darling"
VP	Cat	verb phrase	"is very warm"
VQ	Cat	question complement verb	"ask"
VS	Cat	sentence complement verb	"claim"
VV	Cat	verb phrase complement verb	"want"
[Adv]	Conjunction	adverb list	"here, oddly"
[AP]	Conjunction	adjectival phrase list	"even, very odd"
[NP]	Conjunction	noun phrase list	"John, all women"
[S]	Conjunction	sentence list	"I walk, you run"

5.2 Functions

Function	Module	Type	Example
AAnter	Common	Ant	""
ASimul	Common	Ant	""
AdAdv	Adverb	AdA -> Adv -> Adv	"very"
AdAP	Adjective	AdA -> AP -> AP	"very old"
AdjCN	Noun	AP -> CN -> CN	"big house"
AdnCAdv	Adverb	CAdv -> AdN	"more than"
AdNum	Noun	AdN -> Num -> Num	"almost ten"
AdvCN	Noun	CN -> Adv -> CN	"house on the mountain"
AdvIP	Question	IP -> Adv -> IP	"who in Paris"
AdvNP	Noun	NP -> Adv -> NP	"Paris without wine"
AdvSC	Adverb	SC -> Adv	"that he sleeps"
AdvSlash	Sentence	Slash -> Adv -> Slash	"she sees here"
AdVVP	Verb	AdV -> VP -> VP	"always sleep"
AdvVP	Verb	VP -> Adv -> VP	"sleep here"
ApposCN	Noun	CN -> NP -> CN	"number x"
BaseAdv	Conjunction	Adv -> Adv -> [Adv]	"here" - "today"
BaseAP	Conjunction	AP -> AP -> [AP]	"even" - "odd"
BaseNP	Conjunction	NP -> NP -> [NP]	"the car" - "the house"
BaseS	Conjunction	S -> S -> [S]	"I walk" - "you run"
CleftAdv	Idiom	Adv -> S -> Cl	"it is here that she sleeps"
CleftNP	Idiom	NP -> RS -> Cl	"it is she who sleeps"
CompAdv	Verb	Adv -> Comp	"here"
CompAP	Verb	AP -> Comp	"old"
ComparA	Adjective	A -> NP -> AP	"warmer than the house"
ComparAdvAdj	Adverb	CAdv -> A -> NP -> Adv	"more heavily than Paris"
ComparAdvAdjS	Adverb	CAdv -> A -> S -> Adv	"more heavily than she sleeps"

Function	Module	Type	Example
CompIAdv	Question	IAdv -> IComp	"where"
ComplA2	Adjective	A2 -> NP -> AP	"married to her"
ComplN2	Noun	N2 -> NP -> CN	"brother of the woman"
ComplN3	Noun	N3 -> NP -> N2	"connection from that city to Paris"
ComplV2A	Verb	V2A -> NP -> AP -> VP	"paint the house red"
ComplV2	Verb	V2 -> NP -> VP	"love it"
ComplV3	Verb	V3 -> NP -> NP -> VP	"send flowers to us"
ComplVA	Verb	VA -> AP -> VP	"become red"
ComplVQ	Verb	VQ -> QS -> VP	"ask if she runs"
ComplVS	Verb	VS -> S -> VP	"say that she runs"
ComplVV	Verb	VV -> VP -> VP	"want to run"
CompNP	Verb	NP -> Comp	"a man"
ConjAdv	Conjunction	Conj -> [Adv] -> Adv	"here or in the car"
ConjAP	Conjunction	Conj -> [AP] -> AP	"warm or cold"
ConjNP	Conjunction	Conj -> [NP] -> NP	"the man or the woman"
ConjS	Conjunction	Conj -> [S] -> S	"he walks or she runs"
ConsAdv	Conjunction	Adv -> [Adv] -> [Adv]	"here" - "without them, with us"
ConsAP	Conjunction	AP -> [AP] -> [AP]	"warm" - "red, old"
ConsNP	Conjunction	NP -> [NP] -> [NP]	"she" - "you, I"
ConsS	Conjunction	S -> [S] -> [S]	"I walk" - "she runs, he sleeps"
DConjAdv	Conjunction	DConj -> [Adv] -> Adv	"either here or there"
DConjAP	Conjunction	DConj -> [AP] -> AP	"either warm or cold"
DConjNP	Conjunction	DConj -> [NP] -> NP	"either the man or the woman"
DConjS	Conjunction	DConj -> [S] -> S	"either he walks or she runs"
DefArt	Noun	Quant	"the"
DetCN	Noun	Det -> CN -> NP	"the man"
DetPl	Noun	QuantPl -> Num -> Ord -> Det	"the five best"
DetSg	Noun	QuantSg -> Ord -> Det	"this"
EmbedQS	Sentence	QS -> SC	"whom she loves"
EmbedS	Sentence	S -> SC	"that you go"
EmbedVP	Sentence	VP -> SC	"to love it"
ExistIP	Idiom	IP -> QCl	"which cars are there"
ExistNP	Idiom	NP -> Cl	"there is a car"
FunRP	Relative	Prep -> NP -> RP -> RP	"all houses in which"
GenericCl	Idiom	VP -> Cl	"one sleeps"
IDetCN	Question	IDet -> Num -> Ord -> CN -> IP	"which five hottest songs"
IdRP	Relative	RP	"which"
ImpersCl	Idiom	VP -> Cl	"it rains"
ImpPl1	Idiom	VP -> Utt	"let's go"
ImpVP	Sentence	VP -> Imp	"go to the house"
IndefArt	Noun	Quant	"a"
MassDet	Noun	QuantSg	("beer")
NoNum	Noun	Num	""
NoOrd	Noun	Ord	""
NoPConj	Phrase	PConj	""
NoVoc	Phrase	Voc	""
NumInt	Noun	Int -> Num	"51"
NumNumeral	Noun	Numeral -> Num	"five hundred"
OrdInt	Noun	Int -> Ord	"13 th"
OrdNumeral	Noun	Numeral -> Ord	"thirteenth"
OrdSuperl	Noun	A -> Ord	"hottest"
PassV2	Verb	V2 -> VP	"be seen"
PConjConj	Phrase	Conj -> PConj	"and"
PhrUtt	Phrase	PConj -> Utt -> Voc -> Phr	"but come here please"
PlQuant	Noun	Quant -> QuantPl	"these"
PositA	Adjective	A -> AP	"warm"
PositAdvAdj	Adverb	A -> Adv	"warmly"

Function	Module	Type	Example
PossPron	Noun	Pron -> Quant	"my"
PPartNP	Noun	NP -> V2 -> NP	"the city seen"
PNeg	Common	Pol	""
PPos	Common	Pol	""
PredetNP	Noun	Predet -> NP -> NP	"only the man"
PredSCVP	Sentence	SC -> VP -> Cl	"that she sleeps is good"
PredVP	Sentence	NP -> VP -> Cl	"she walks"
PrepIP	Question	Prep -> IP -> IAdv	"with whom"
PrepNP	Adverb	Prep -> NP -> Adv	"in the house"
ProgrVP	Idiom	VP -> VP	"be sleeping"
QuestCl	Question	Cl -> QCl	"does she walk"
QuestIAdv	Question	IAdv -> Cl -> QCl	"why does she walk"
QuestIComp	Question	IComp -> NP -> QCl	"where is she"
QuestSlash	Question	IP -> Slash -> QCl	"whom does she love"
QuestVP	Question	IP -> VP -> QCl	"who walks"
ReflA2	Adjective	A2 -> AP	"married to itself"
ReflV2	Verb	V2 -> VP	"see himself"
RelCl	Relative	Cl -> RCl	"such that she loves him"
RelCN	Noun	CN -> RS -> CN	"house that she buys"
RelSlash	Relative	RP -> Slash -> RCl	"that she loves"
RelVP	Relative	RP -> VP -> RCl	"that loves her"
SentAP	Adjective	AP -> SC -> AP	"good that she came"
SentCN	Noun	CN -> SC -> CN	"fact that she smokes"
SgQuant	Noun	Quant -> QuantSg	"this"
SlashPrep	Sentence	Cl -> Prep -> Slash	(with whom) "he walks"
SlashV2	Sentence	NP -> V2 -> Slash	(whom) "he sees"
SlashVVV2	Sentence	NP -> VV -> V2 -> Slash	(whom) "he wants to see"
SubjS	Adverb	Subj -> S -> Adv	"when he came"
TCond	Common	Tense	""
TEmpty	Text	Text	""
TFut	Common	Tense	""
TExclMark	Text	Phr -> Text -> Text	"She walks!"
TFullStop	Text	Phr -> Text -> Text	"She walks."
TPast	Common	Tense	""
TPres	Common	Tense	""
TQuestMark	Text	Phr -> Text -> Text	"Does she walk?"
UseA2	Adjective	A2 -> A	"married"
UseCl	Sentence	Tense -> Ant -> Pol -> Cl -> S	"she wouldn't have walked"
UseComp	Verb	Comp -> VP	"be warm"
UseN2	Noun	N2 -> CN	"brother"
UseN3	Noun	N3 -> CN	"connection"
UseN	Noun	N -> CN	"house"
UsePN	Noun	PN -> NP	"Paris"
UsePron	Noun	Pron -> NP	"she"
UseQCl	Sentence	Tense -> Ant -> Pol -> QCl -> QS	"where hadn't she walked"
UseRCl	Sentence	Tense -> Ant -> Pol -> RCl -> RS	"that she hadn't seen"
UseVQ	Verb	VQ -> V2	"ask" (a question)
UseVS	Verb	VS -> V2	"know" (a secret)
UseV	Verb	V -> VP	"sleep"
UttAdv	Phrase	Adv -> Utt	"here"
UttIAdv	Phrase	IAdv -> Utt	"why"
UttImpPl	Phrase	Pol -> Imp -> Utt	"love yourselves"
UttImpSg	Phrase	Pol -> Imp -> Utt	"love yourself"
UttIP	Phrase	IP -> Utt	"who"
UttNP	Phrase	NP -> Utt	"this man"
UttQS	Phrase	QS -> Utt	"is it good"
UttS	Phrase	S -> Utt	"she walks"
UttVP	Phrase	VP -> Utt	"to sleep"
VocNP	Phrase	NP -> Voc	"my brother"

Function	Module	Type	Example
num	Numeral	Sub1000000 -> Numeral	"2"
n2	Numeral	Digit	"2"
n3	Numeral	Digit	"3"
n4	Numeral	Digit	"4"
n5	Numeral	Digit	"5"
n6	Numeral	Digit	"6"
n7	Numeral	Digit	"7"
n8	Numeral	Digit	"8"
n9	Numeral	Digit	"9"
pot01	Numeral	Sub10	"1"
pot0	Numeral	Digit -> Sub10	"3"
pot110	Numeral	Sub100	"10"
pot111	Numeral	Sub100	"11"
pot1to19	Numeral	Digit -> Sub100	"18"
pot0as1	Numeral	Sub10 -> Sub100	"3"
pot1	Numeral	Digit -> Sub100	"50"
pot1plus	Numeral	Digit -> Sub10 -> Sub100	"54"
pot1as2	Numeral	Sub100 -> Sub1000	"99"
pot2	Numeral	Sub10 -> Sub1000	"600"
pot2plus	Numeral	Sub10 -> Sub100 -> Sub1000	"623"
pot2as3	Numeral	Sub1000 -> Sub1000000	"999"
pot3	Numeral	Sub1000 -> Sub1000000	"53000"
pot3plus	Numeral	Sub1000 -> Sub1000 -> Sub1000000	"53201"