Technical Report no. 2007-09

The Functional Morphology Library

Markus Forsberg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2007

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2007-09
ISSN 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Göteborg, Sweden, 2007

1 Introduction

This document contains the technical report of *Functional Morphology (FM) version 2.0*. The aim of the text is to supply a detailed description on how to use FM and to provide some insights into the implementation of FM.

FM is a library for **programming** lexical resources. It is **not** a new linguistic formalism. It helps creating a lexical resource in a structured and efficient way. It is also a compiler, able to translate a lexical resource, defined in FM, into many other lexical resource formats, such as SQL or XFST source code.

Note that to be able to fully benefit from this document, a basic knowledge of the functional programming language Haskell is required.

2 FM Tutorial

This section presents a detailed walk-through of a fragment of a Latin morphology implemented in FM. Even though the choice of Latin is arbitrary, it works as a nice example for FM since it is a highly inflected language, which fits perfectly with the word-and-paradigm model of FM.

2.1 Overview

An implementation of a lexical resource in FM consists of clearly distinct components, which is naturally put into different Haskell modules. The components, listed below with short explanations, will be presented one by one in the following sections.

Type system The type system consists of inflectional, inherent and dictionary types, i.e. the parameters of the lexical resource, defined with algebraic data types.

Paradigm functions The paradigms of the lexical resource expressed as finite functions over the algebraic data types.

Interface functions The paradigm functions are translated to interface functions. An interface function connects a paradigm function to the dictionary, which includes the production of the inflection table and the addition of the inherent parameters.

Lexicon There are two kinds of lexicon for a lexicon resource, an *internal* and an *external* one. We will see that even though it may be convenient with internal lexica, there are reasons to only use external lexica.

Paradigm names The words in the external lexicon are annotated with paradigm names. The paradigm name module connects these names with interface functions.

Compound analysis (optional) If compound analysis is used, all word forms are given an attribute that defines how they can be combined with other words forms. The compound analysis function defines which of the attribute sequences correspond to possible compounds.

Main module The main module puts everything together into a runtime system.

2.2 Type System

The type system defines all inflectional and inherent parameters of the morphology. The parameters are defined with algebraic data types. Inflectional parameters, for example number and case, are parameters dictating how a word is inflected. Inherent parameters are attributes associated to a word, such as gender or subcategorization frame. Inherent parameters differ from inflectional parameters in that inflectional parameters are associated to a word form, but an inherent parameter is associated to the word — e.g. a feminine noun is not inflected in feminine, it *is* feminine.

The use of algebraic data types instead of ordinary strings gives many advantages. It gives a guarantee that the correct parameters are used for a paradigm as long as the correct word class is chosen. Furthermore, it is possible to define the types in such a way that only valid parameter configurations are possible to construct. For example, the cross product of the inflectional parameters of Latin verbs generates 1260 forms, but only 147 forms are existing — with algebraic types we can define a type system which disallows the 1113 non-existing forms.

Furthermore, if *incomplete pattern detection* is activated in the Haskell compiler, we can get information about missing cases. That is, if we forget to define the word forms for some parameter configurations, then the compiler will complain.

As an example, consider the Latin noun *causa* (Eng. 'cause'). It is inflected in number and case, i.e. number and case are the inflectional parameters. It also has one inherent parameter: *gender*. The inflection of *causa* in plural nominative is *causae*, but it has a feminine gender.

The parameters of Latin nouns are described with the help of Haskell's data types. To describe them, we introduce the types: `Gender`, `Case` and `Number`, given in Fig. 1. The `deriving` part is needed to ensure that the

```
data Gender = Feminine  |
              Masculine |
              Neuter
 deriving (Show,Eq,Enum,Ord,Bounded)

data Case   = Nominative |
              Genitive   |
              Dative     |
              Accusative |
              Ablative   |
              Vocative
 deriving (Show,Eq,Enum,Ord,Bounded)

data Number = Singular |
              Plural
 deriving (Show,Eq,Enum,Ord,Bounded)
```

Figure 1: Type system

type is finite and enumerable — it gives us a way to enumerate all objects in a type.

The inflectional parameter types `Case` and `Number` are combined into one dictionary type `NounForm` describing all inflection forms of a Latin noun. `Gender` is not part of the dictionary type, since it is an inherent parameter.

```
data NounForm = NounForm Number Case
 deriving (Show,Eq,Ord,Bounded)
```

The `NounForm` type is missing a `deriving` for the `Enum` class. This is because the current main Haskell compiler is unable to derive the enumeration of a data type containing constructors with arguments. The reason for this is obvious for arguments that are not `Enum` and `Bounded`, since there is no obvious enumeration strategy in that case, but there is a natural strategy if they are: simply enumerating a constructor's arguments from left to right.

An important class in FM is `Param`, defined in `General.hs`. The most important method — the only one not defined by default — is `values`. It gives the complete list of the objects in a `Param` type. An instance of `Param` is easy to define for bounded enumerated types with the function `enum`, using the member functions of `Enum` (the list generator) and `Bounded` (`minBound` and `maxBound`).

```
enum :: (Enum a, Bounded a) => [a]
enum = [minBound .. maxBound]
```

We continue by instantiating the parameters of the Latin nouns in the class `Param`. The default definition for `prValue` has been redefined for `NounForm`, to remove the `NounForm` constructor. Usually, a more sophisticated printing scheme is preferred, using a particular *tag set*, i.e. adopting to a standard for describing the types of a language.

```
instance Param Gender  where values = enum
instance Param Case    where values = enum
instance Param Number  where values = enum
instance Param NounForm where
  values =
   [NounForm n c | n <- values ,
                   c <- values]
  prValue (NounForm n c) =
    unwords $ [prValue n, prValue c]
```

A paradigm of Latin nouns is defined as a finite function `Noun`, from a `NounForm` to a `Str`; from a parameter configuration to a word form. The type for word forms is actually a list of strings, instead of a single one. The reason for this is to be able to describe missing word forms and word form variants.

```
type Noun = NounForm -> Str
```

A `Noun` is translated to an inflection table by generating all `NounForm` objects and applying them to the `Noun`. This is done by the function `table`.

```
table :: Param a => (a -> Str) -> [(a,Str)]
table f = [(a,f a) | a <- values]
```

Note that this function is polymorphic — the only restriction of `a` is that it is an instance of the `Param` class.

2.3 String operations

FM provides a set of string operation functions capturing common phenomena in word inflection. For a complete reference, see `General.hs` in the FM API (see Sec. 15.1).

The set of string operations is by no means complete. An implementer of a lexical resource typically writes new functions reflecting some specifics of the target language. For example, if it is common in a language that the second last letter is dropped while inflecting a word, it is reasonable to write a function that does exactly that. These new functions can be delivered as an extended library, which will simplify the implementation of a similar language.

For example, among the string operations are the functions `tk` and `dp`, similar to Haskell's standard functions `take` and `drop`, but they focus on suffixes instead of prefixes. `tk` takes all but the `n` last characters and `dp` drops all but the last `n` characters.

```
tk :: Int -> String -> String
tk n s = take (max 0 (length s - n)) s

dp :: Int -> String -> String
dp n s = drop (max 0 (length s - n)) s
```

Yet another example is the operator `(+?)`, which implements the common phenomenon: if the last letter of a word and the first letter of an ending coincide, then one of them is dropped. An example of the usage is given in the function `mkCase`, where the genitive case of Swedish nouns is formed by adding an 's' to word forms, unless it does not already end in 's'. In that case, nothing is added.

```
(+?) :: String -> String -> String
s +? e = case (s,e) of
            (_:_,c:cs) | last s == c -> s ++ cs
            _ -> s ++ e

mkCase :: Case -> String -> String
mkCase c w = case c of
  Nom -> w
  Gen -> w +? "s"
```

The strings operations all share the property that they perform a small, specific task. And more, that their definitions are compact and easily understood.

3 Paradigms as functions

Let us start by considering the first declension noun paradigm, illustrated with the inflection table of the word `rosa` (Eng. 'rose'), in Fig. 2. The concept of inflection tables corresponds intuitively to a list of pairs in a programming language, but FM takes an indirect approach and uses finite

	Singular	**Plural**
Nominative	*rosa*	*rosae*
Vocative	*rosa*	*rosae*
Accusative	*rosam*	*rosas*
Genitive	*rosae*	*rosarum*
Dative	*rosae*	*rosis*
Ablative	*rosa*	*rosis*

Figure 2: The inflection table of *rosa*

functions, which is later translated to a list of pairs. The use of finite functions has many advantages: it allows the use of higher-order functions, e.g. see *exceptions* in Sec. 3.1; it allows us to divide the paradigm definitions into sets of finite functions, each solving a specific task, which in the end are combined into one function with function compositions; and it allows the use of pattern matching, which permits common cases to be defined simultaneously.

The paradigm function for the first declension paradigm, `decl1`, is directly defined based on the inflection table. It is defined as a single function.

```
decl1rosa :: String -> Noun
decl1rosa rosa (NounForm n c) =
 mkStr $
  case n of
   Singular ->
    case c of
     Accusative -> rosa ++ "m"
     Genitive   -> rosa ++ "e"
     Dative     -> rosa ++ "e"
     _          -> rosa
   Plural   ->
    case c of
     Nominative -> rosa ++ "e"
     Vocative   -> rosa ++ "e"
     Accusative -> rosa ++ "s"
     Genitive   -> rosa ++ "rum"
     _          -> rosa ++ "is"
```

Note that the paradigm function requires one argument, a citation word form. The functions `rosa` and `puella` are two nouns created by the application of the citation forms `"rosa"` and `"puella"` (Eng. 'girl').

```
rosa :: Noun
rosa   = decl1 "rosa"

puella :: Noun
puella = decl1 "puella"
```

3.1 Exceptions

Many paradigms of the same type are similar, just differing in one or two word forms. When defining a class of similar paradigms, its is convenient to use FM:s *exceptions*. Exceptions are used to describe inflection functions in terms of other inflection functions. Instead of defining a completely new paradigm, we use the old definition and only mark what is different. This is not only linguistically more satisfying, it saves a lot of work.

There are four different kinds of exception: `excepts`, `missing`, `only` and `variants`. All exceptions are higher-order functions that take a finite inflection function as an argument.

The exceptions `except` and `excepts`, take a finite inflection function and list of exceptions, and constructs a new finite function with the exceptions included. An example of its usage is given in the definition of `decl2gladius`.

```
except  :: Param a => (a -> Str) -> [(a,String)] -> (a -> Str)
excepts :: Param a => (a -> Str) -> [(a,Str)]    -> (a -> Str)

decl2gladius :: String -> Noun
decl2gladius gladius =
  except (decl2servus gladius)
     [(NounForm Singular Genitive, gladi),
      (NounForm Singular Vocative, gladi)]
 where gladi =  tk 2 gladius
```

The exception functions `missing` and `only` are used to express missing cases in a table. `missing` enumerates the cases with missing forms, and `only`, used for highly defective words, enumerates the cases that exists. An example is the paradigm of *vis* (Eng. 'force'), which inflects in the same manner as *hostis* (Eng. 'enemy'), with the exception that it is missing the singular vocative, genitive and dative case.

```
missing :: Param a => (a -> Str) -> [a] -> (a -> Str)
only    :: Param a => (a -> Str) -> [a] -> (a -> Str)

vis_paradigm :: String -> Noun
vis_paradigm s = (hostisParadigm s) 'missing'
   [NounForm Singular c | c <- [Vocative, Genitive, Dative]]
```

A very common exception is additional variants, i.e. that two paradigms differ only in the number of word forms for one or more parameter configuration. This type of exception is expressed with the functions `variant` and `variants`.

An example is given with the function `decl3parti`, a Swedish paradigm function. The function is defined in terms of a worst-case function `mkNoun`, which takes `String`s as arguments. This function is then augmented with two variant word forms through the use of `variant`.

```
variant  :: Param a => (a -> Str) -> [(a,String)] -> (a -> Str)
variants :: Param a => (a -> Str) -> [(a,Str)]    -> (a -> Str)

decl3parti :: String -> Substantive
decl3parti parti =
 mkNoun parti (parti ++ "et") (parti ++ "er") (parti ++ "erna")
  `variant`
  [(SF Sg Def c, mkCase c (parti++"t") | c <- values]
```

Note that we use `values` to generate the values of `c`. The type system is able to infer that `c` is of type `Case`, and since `Case` is an instance of the class `Param`, we can use the function `values` to generate the constructors `Nom` and `Gen`.

4 Interface Functions

A lexical resource has its own type system, so to be able to use generic translations, we need to translate it into an intermediary format, a `Dictionary`. A `Dictionary` is an untyped ADT consisting of a list of `Entry`:s. An `Entry` corresponds to dictionary entry, specifying information about a word, e.g. the inflection table and the inherent parameters.

The translation is done by first instantiating the dictionary types in the `Dict` class, defined in `Dictionary.hs` (see Sec. 15.2). Typically, the only information we need to supply is the name of the word class that the dictionary type represents. Note that since we have no access to the names of the types within Haskell, we must require that this information is supplied by the user.

Let us return to the Latin noun example with the dictionary type `NounForm`. When the `NounForm` type is made an instance of `Dict`, we also give the name of the word class that `NounForm` represents, i.e. `Noun`.

```
instance Dict NounForm where
 category _ = "Noun"
```

The next step is to define interface functions, i.e. functions that create `Entry`:s. We start by a general interface function, `noun`, which transforms a `Noun` together with its inherent parameter, `Gender`, and its paradigm identifier, to an `Entry`. The identifier is not the same as the paradigm names in a command map (see Sec. 6), it is used in the *word identifier*. A word identifier is built up from the citation form, the word class, the inherent parameters and the identifier. For example, the word *rosa* has the identifier `rosa_Noun_Feminine__n1`. The main difference between the command map identifiers and the ones in the word identifiers is that command map identifiers must be unique.

We also define a function for every gender: `masculine`, `feminine` and `neuter`.

```
noun :: Noun -> Gender -> Paradigm -> Entry
noun n g p = entryIP n [prValue g] p

feminine :: Noun -> Paradigm -> Entry
feminine n = noun n Feminine

masculine :: Noun -> Paradigm -> Entry
masculine n = noun n Masculine

neuter :: Noun -> Paradigm -> Entry
neuter n = noun n Neuter
```

We can now define interface functions for all our paradigm functions. Let us have a look at three interface functions, one for each gender: `d1rosa` (Eng. 'rose'), `d2servus` (Eng. 'servant'), and `d2bellum` (Eng. 'war').

```
d1rosa :: DictForm -> Entry
d1rosa w = feminine (decl1rosa w) "n1"

d2servus :: DictForm -> Entry
d2servus w = masculine (decl2servus w) "n2"

d2bellum :: DictForm -> Entry
d2bellum w = neuter (decl2bellum w) "n2"
```

We can now create a small lexicon with the interface functions we have defined. The function `dictionary` is an abstraction function that creates a `Dictionary` ADT from a list of `Entry`:s.

```
latinDict :: Dictionary
latinDict =
 dictionary $
  [
   d1rosa    "puella",
   d2servus  "somnus",
   d2servus  "amicus",
   d2bellum  "donum"
  ]
```

The lexicon we just defined is referred to as an *internal lexicon*, since it is defined within Haskell. If we add a new word, we need to recompile our FM implementation, but on the other hand, we have the full power of Haskell at our disposal. This is contrasted with an *external lexicon*, which is simply a text file (discussed in Sec. 6). If we add a new word to the *external lexicon* there is no need to recompile. We are, however, more restricted in what we can express, since we no longer have access to Haskell.

In previous documentations of FM, we recommended that the irregularly inflected words should be defined in the internal lexicon, and the regularly ones in the external lexicon. We have reconsidered this somewhat, since it is very convenient to have all words listed in the external lexicon — all in the same place, in the same format. A definite preference, however, depends on the intended usage of the lexical resource in question.

5 Compound Analysis

FM offers the possibility to perform compound analysis. By default, all words are assumed to appear outside compounds, so the compound analysis is invisible to someone who does not use it.

We will use the particle *ne* (Eng. approximately '?') as our example, a clitic element that can be placed on any word, and by that, it expresses that the word it attaches to is in some way questioned.

The first thing we need to do is to define *attributes*, which will be used to describe the compound behaviour of *ne*. Attributes are integers greater than one and we use the type `Attr` to refer to them.

For the purpose of our example, we will only define one attribute, `atS`, which will be used for words that may only appear as a suffix on another word form.

```
 atS :: Attr
 atS = 1
```

The next step is to associate the attribute to the dictionary type. This is done in the instantiation of the class `Dict`, or more precisely, in the class function `defaultAttr`.

```
instance Dict ParticleForm where
 category    _ = "Particle"
 defaultAttr _ = atS
```

All words with the dictionary type `ParticleForm` now have the default attribute `atS`. As the name `defaultAttr` implies, it is also possible to associate an attribute value to any parameter configuration.

All word forms have an attribute associated to them. If no attribute association has been defined for a word form, it receives the default attribute value. The default attribute value is `0`, which explains why a user-defined attribute value must be larger than `0`.

The compound analysis try to divide an input word form into all possible sequences of word forms with their associated attributes. Some of these sequences will, of course, not be valid. It is the *compound function*, defined by the implementer of the lexical resource, which decides what attributes are valid.

The compound function `latin_compound` defines the valid attribute sequences of our Latin lexical resource.

```
latin_compound :: [Attr] -> Bool
latin_compound [x,y] = (x /= y) && atS == y
latin_compound [x]   = x /= atS
latin_compound _     = False
```

Word forms with the attribute `atS` may only occur as suffixes. All other word forms may only occur as single words.

6 Paradigm Identifiers and External Lexicon

We do not want to recompile the whole system every time we add a new word, in fact, we may not want to recompile at all. This is where the external lexicon comes into picture. An external lexicon is a text file containing a list of citation forms marked with *paradigm names*. A paradigm name refers to an interface function.

The first thing we need to do to get things working is to define a *command map*. The command map defines the mapping between paradigm name and interface functions. It consists of a list of triplets, where the first element is the paradigm name, the second element is example citation forms for the paradigm, and the third element is the interface function.

The interface functions are applied to a function `app1` that requires a special explanation. First of all, we want to be able to have interface functions that have more than one argument. But then we have a problem, since the type system of Haskell does not allow functions of different types to appear at the same position in a list. The solution provided by FM is to use one of a set of wrapper functions, named `app1`, `app2`, `app3`, et cetera, where the number corresponds to the argument count. These wrapper functions encapsulate the interface functions, creating new functions of the type `[String] -> Entry`. Since all wrapper functions create a function of the same type, we can have interface functions of different argument count appearing in the command list.

```
commands =
 [
  ("d1puella",     ["rosa"],   app1 d1puella),
  ("d1puellaMasc", ["poeta"],  app1 d1puellaMasc),
  ("d2servus",     ["servus"], app1 d2servus),
  ("d2servusFem",  ["pinus"],  app1 d2servusFem),
  ("d2servusNeu",  ["virus"],  app1 d2servusNeu),
  ("d2bellum",     ["bellum"], app1 d2bellum),
  ("d2puer",       ["puer"],   app1 d2puer),
  ("d2liber",      ["liber"],  app1 d2liber),
  ("prep",         ["ad"],     app1 prep),
  ("v1amare",      ["amare"],  app1 v1amare),
  ("v2habere",     ["habere"], app1 v2habere)
  ]
```

Given that we have defined our command map (and our runtime system, see Sec. 7), then we can start developing our external lexicon.

```
latin.lexicon:
 v1amare  amare
 v1amare  portare
 v1amare  demonstrare
 v1amare  laborare
```

The external lexicon is in a file `latin.lexicon`, where we have defined four words in the first conjugation: *amare* (Eng. 'to love'), *portare* (Eng. 'to carry'), *demonstrare* (Eng. 'to point out'), and *laborare* (Eng. 'to work'). Note that the format of an external lexicon is simple — it consists of paradigm names and citation forms. Single line comments are allowed, triggered by `--`, but besides that, there is nothing more.

7 Runtime System

The last thing we need to do is to connect our lexical resource with the runtime system of FM. For this, FM uses a class `Language`, defined in `Frontend.hs` (see Sec. 15.4), which gives the language-specific parts of the runtime system. We start by defining a type consisting of a single constructor, which is the default name of our lexical resource.

```
data Latin = Latin
 deriving Show
```

Next, we make our data type an instance of of the `Language` class, where we define functions needed for the runtime system. All class functions have a default definition, e.g. the internal dictionary may be empty; the list of commands may be empty; or there may be no compound analysis. In this instance, we define our internal dictionary, our compound function, and our command map, which is folded into a more efficient lookup table.

```
instance Language Latin where
 internDict   _ = latinDict
 composition  _ = latin_compound
 paradigms    _ = foldr insertCommand emptyC commands
```

We can now define our `main` function with the help of the FM library `commonMain` applied to our constructor `Latin`.

```
 main :: IO ()
 main = commonMain Latin
```

The constructor `Latin` is used to retrieve the information provided in the instance of the `Language` class. It is a convenient way to avoid having many optional arguments in the `commonMain` function.

This concludes the FM tutorial — we have now defined a complete fragment of a lexical resource for Latin. For information on how to compile FM, see Sec. 10, and on how to run FM, see Sec. 11.

8 Extending the Translator

This section lists how to add a new output format called `FORMAT`. For additional help, have a look at how another format is defined.

1. Define a function in `Print.hs`:
   `prFORMAT :: Dictionary -> String`

2. Define two functions in `GeneralIO.hs`:
   `writeFORMAT :: FilePath -> Dictionary -> IO()`
   `outputFORMAT :: Dictionary -> IO()`
   These functions, responsible of writing the output of `prFORMAT` to the file `Filepath` and standard output respectively, typically add an header to the output.

3. In `CommonMain.hs`: add a new command-line flag (e.g. `-format`) and document it in the help message `help_text`.

9 Compound Analysis in FM

A compound in FM is a word $w = w_1 w_2 ... w_n$ where $dictionary(w_i)$ and $valid(attr(w_1)...attr(w_n))$. *dictionary* is a boolean function defining the word forms of a language. *attr* is function that for every word form in the dictionary assigns a set of parameter values. The parameter values defines how the word forms can be composed with other words. *valid* is a boolean function that accepts as input a list of sets of attribute values and gives as result a boolean value that states if the sequence of attribute values is valid or not.

The compound analysis of FM consists of two functions, `unglue` and `valid`. The `unglue` function is a rewritten version of Huet's unglueing function [3], which splits an input word, based on a dictionary, into all possible compounds. Note that it is essential to have the dictionary check in the generator, since the generator would otherwise be subject to a combinatorial explosion.

```
unglue [] dictionary = [[]]
unglue  w dictionary = [map (pre:) (unglue suf) |
                          (pre,suf) <- zip (prefixes w)
                                           (suffixes w)
                          dictionary pre]
```

The `valid` function uses the compound function to filter out only the valid compound.

```
valid c_fun cs = filter_valid attr_values
  where
   attr_values = flatten $ map lookup_attr cs
   filter_valid [] = []
   filter_valid (as:ass)
    | c_fun (extract_attr as) = as : filter_valid ass
    | otherwise               = filter_valid ass
```

These two functions could be combined to avoid duplicate work, but are held separate for the presentation. It may seem inefficient to separate the validity test with the actual unglueing, but since Haskell is a lazy programming language, the situation is better. The laziness ensures that the unglueing process only continues on the suffix if the prefix is in the lexicon.

Since a word form may be a homograph, it can be associated with a set of attributes. Because of this, we need to use the function `flatten` to flatten the sequences of sets of attributes into a set of attribute sequences.

The function `compound_analysis` puts everything together.

```
compound_analysis c_fun w =
  concat $ map (valid c_fun) (unglue w)
```

Huet [3] uses a different approach to compounds: he uses rewrite rules to describe internal and external sandhi of Sanskrit. The rules are compiled into a dictionary trie with the addition of choice points, which encodes the rules. The sandhi of Sanskrit is complicated since the spelling exactly reflects the pronunciation of the sentences.

It is not clear that it would be possible to handle Sanskrit's sandhi in FM. It may be the case that the number of word forms would be to great to be feasible to define in FM style.

The translation in FM of a lexical resource with compounding to other systems is not complete — even though the compound information is exported, the compound function is missing. The compound function is currently a function in Haskell that is not readily translatable. The situation could be improved by an algebraic representation of the compound function, which in turn would be translatable. How such an algebraic representation is best implemented requires some future work.

10 Compiling FM

The source code is downloadable at FM's homepage[1].

FM requires the GHC compiler[2] to be built. Since FM is a command-line program, it should work on all platforms supported by the GHC compiler.

1. Unpack the source code: `tar -xvfz FM_LAT_v2.0.tgz`
2. Change directory: `cd ./FM_LAT_v2.0/`

[1] `http://www.cs.chalmers.se/~markus/FM`

[2] `http://www.haskell.org/ghc`

3. Compile FM: `make`

4. This produces a binary `morpho_lat`

11 Running FM

We assume that the tutorial language is downloaded — the lexical resource of Latin, and compiled in the manner described in Sec. 10. Other FM implementations are handled in an analogous way.

Before FM implementation can be run, one needs to refer to the external lexicon, `latin.lexicon`. This is done by either running the program in the same directory as the external lexicon, or by pointing the environment variable `FM_LAT` to it. Environment variables are set differently depending on which shell are in use, but in a Bash shell, and given that the lexicon file is placed in the directory `/home/dictionary`, we would write the command below. Or better, put the command in one of the system files declaring the environment variables.

```
$ export FM_LAT="/home/dictionary/latin.lexicon"
```

The runtime system of an FM implementation consists of four parts: the analyzer, the synthesizer, the inflection engine, and the translator. We will describe each of these parts in the rest of this section. An overview of the command-line flag of FM is printed with the help command `morpho_lat -h`. The output is given in Fig. 3.

11.1 The Analyzer

The *analyzer*, also referred to as the *tagger*, annotates the word forms of an input text with information collected from the current lexical resource. The analyzer is divided into two phases: *word segmentation*, or *tokenization*, and *word analysis*. The word segmentation splits the string of the input text into tokens, and the word analysis, which may be compound analysis, does the actual annotation.

An example of the analyzer in action is given in Fig. 4, where two word forms is being tagged: *servi* and *servusne*. The first word, *servi*, is the inflection form of *servus* (Eng. 'servant'). It is ambiguous: it may be singular genitive, plural nominative or plural vocative. The second word, *servusne*, is a compound word consisting of *servus* and the question particle *ne*.

The part about *Morphology Statistics* contains information about the lexical resource — here we see that 11 paradigms have been implemented;

```
|---------------------------------------|
|        Program parameters             |
|---------------------------------------|
| -h             | Display this message |
|---------------------------------------|
| <None>         | Enter tagger mode    |
|---------------------------------------|
| -s             | Enter interactive    |
|                | synthesiser mode     |
|---------------------------------------|
| -i             | Enter inflection     |
|                | mode                 |
|---------------------------------------|
| -lex    [file] | Full form lexicon    |
| -tables [file] | Tables               |
| -gf     [file] | GF source code       |
| -latex  [file] | LaTeX source code    |
| -xml    [file] | XML source code      |
| -lexc   [file] | LexC source code     |
| -xfst   [file] | XFST source code     |
| -sql    [file] | SQL source code      |
|---------------------------------------|
```

Figure 3: FM help

the lexicon consists of 196 entries, 173 in the external lexicon and 23 in the internal; these entries are expanded into 8131 word forms (yes, Latin is a highly inflected language!) of which 5417 are unique. Finally, the compile time for the dictionary and the building time of the analysis data structure sums to 1.00 seconds.

FM is, of course, capable of analyzing a complete text. Given a Latin text `Latina_Vulgate.txt`, we analyze the whole text by simply piping the text to the FM program, as illustrated below.

```
$ cat Latina_Vulgate.txt | morpho_lat
```

The analysis is reasonably fast. The analysis of 'Latina Vulgate', consisting of approximately 1.4 million word forms, took around 25 seconds on a Macbook (including the compile time of the lexical resource), which gives us an analysis speed in the ballpark of 56k word forms per second.

11.2 The Synthesizer

The synthesizer is used to retrieve all dictionary entries that include the input word form in their inflection table. An example is provided in Fig. 5 with the word form *puellae*. In this example, we only got one entry, since

```
$ morpho_lat

*************************************
*   Functional Morphology v2.0      *
* (c) M. Forsberg & A. Ranta 2007   *
* under GNU General Public License. *
*************************************

Morphology Statistics:
 # language id: latin
 # 11 paradigms
 # 0k entries (e: 173, i: 23)
 # 8k word forms (c: 8131, u: 5417)
 # compile time: 1.00 seconds

servi
[ <servi>
1. servus (servus_Noun_Masculine__n2:1)
          Noun - Plural Vocative - Masculine [0]
2. servus (servus_Noun_Masculine__n2:1)
          Noun - Plural Nominative - Masculine [0]
3. servus (servus_Noun_Masculine__n2:1)
          Noun - Singular Genitive - Masculine [0]
]
servusne
[ <servusne>
1. Composite: servus (servus_Noun_Masculine__n2:1) Noun -
              Singular Nominative - Masculine [0]  |
              ne (ne_Particle__inv:1) Particle - Invariant [1]
]
```

Figure 4: FM analysis example

the word form *puellae* only appears in one entry. If it were an homograph, on the other hand, appearing in more than one entry, then those entries would be listed also.

The current version of FM does not include compound words in the synthesizer: it only retrieves word forms that exists in the lexicon. How to include compound words in the synthesis is by no means obvious — we need a method to decide which word form in the compound that is the main word form, to be able to select the correct word class and the correct inherent parameters. And more, this method must be valid for any language, and preferably invisible to an FM implementer that do not use compound analysis.

A possible solution would be to strengthen the compound function so that it would, as a result, not only give true or false, but also which attribute corresponds to the main word form. Such an addition, however, would break the backward compatibility of FM.

12 The Inflection Engine

The inflection engine of FM translates paradigm names applied to citation word forms to dictionary entries. The inflection engine is also runnable in batch mode, i.e. the input can be piped to the program. An example of the inflection engine in interactive mode is given in Fig. 6, where the word *porta* (Eng. 'door') is marked as being in the first declension. The same result is achieved in batch mode with the following command.

```
$ echo "d1rosa porta" | ./morpho_lat -ib
```

Typing `'c'` in interactive mode gives the list of all paradigm names together with their example word forms.

13 The Translator

An important aspect of FM is its use as a compiler. The idea is that the user of FM should never get "stuck" in FM, but instead have the ability to translate the lexical resource to many other lexicon formats, and by doing that, maximize the usefulness of the resource. In fact, FM has been designed so that adding a new format is a relatively small task (see Sec. 8 for details).

The formats currently supported by FM will now be exemplified one by one. We will use the same example word, and only the part of the

```
$ morpho_lat -s

************************************
*   Functional Morphology v2.0      *
* (c) M. Forsberg & A. Ranta 2007   *
* under GNU General Public License. *
************************************


Synthesiser mode

Enter a Latin word in any form

or a [paradigm name] with [word forms].

Type 'c' to list paradigms.

Type 'q' to quit.

Morphology Statistics:
 # language id: latin
 # 11 paradigms
 # 0k entries (e: 173, i: 23)
 # 8k word forms (c: 8131, u: 5417)
 # compile time: 0.00 seconds

> puellae
[ <puellae>
{
lemma: puella
pos: Noun
inherent(s):  Feminine
paradigm id: n1

Singular Nominative : puella
Singular Vocative : puella
Singular Accusative : puellam
Singular Genitive : puellae
Singular Dative : puellae
Singular Ablative : puella
Plural Nominative : puellae
Plural Vocative : puellae
Plural Accusative : puellas
Plural Genitive : puellarum
Plural Dative : puellis
Plural Ablative : puellis
}
]
```

Figure 5: FM synthesis example

```
$ morpho_lat -i

************************************
*   Functional Morphology v2.0     *
* (c) M. Forsberg & A. Ranta 2007  *
* under GNU General Public License. *
************************************


[Inflection mode]

Enter [paradigm name] with [word forms].

Type 'c' to list paradigms.

Type 'q' to quit.

> d1rosa porta
porta
Noun
Feminine
Singular Nominative: porta
Singular Vocative: porta
Singular Accusative: portam
Singular Genitive: portae
Singular Dative: portae
Singular Ablative: porta
Plural Nominative: portae
Plural Vocative: portae
Plural Accusative: portas
Plural Genitive: portarum
Plural Dative: portis
Plural Ablative: portis

>
```

Figure 6: FM inflection mode example

generation that refers to that word, to enable comparisons between the different formats. The word used is *filius* (Eng. 'son').

13.1 Full Form Lexicon

A fundamental format in FM is the *full form lexicon*, which is the format that the analyzer builds on. The format consists of all word forms annotated with their analyses (separated by `':'`).

```
filius:filius (filius_Noun_Masculine__n2:1) Noun -
              Singular Nominative - Masculine [0]
fili:filius (filius_Noun_Masculine__n2:1) Noun -
            Singular Vocative - Masculine [0]
filium:filius (filius_Noun_Masculine__n2:1) Noun -
              Singular Accusative - Masculine [0]
fili:filius (filius_Noun_Masculine__n2:1) Noun -
            Singular Genitive - Masculine [0]
filio:filius (filius_Noun_Masculine__n2:1) Noun -
             Singular Dative - Masculine [0]
filio:filius (filius_Noun_Masculine__n2:1) Noun -
             Singular Ablative - Masculine [0]
filii:filius (filius_Noun_Masculine__n2:1) Noun -
             Plural Nominative - Masculine [0]
filii:filius (filius_Noun_Masculine__n2:1) Noun -
             Plural Vocative - Masculine [0]
filios:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Accusative - Masculine [0]
filiorum:filius (filius_Noun_Masculine__n2:1) Noun -
                Plural Genitive - Masculine [0]
filiis:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Dative - Masculine [0]
filiis:filius (filius_Noun_Masculine__n2:1) Noun -
              Plural Ablative - Masculine [0]
```

13.2 Inflection Tables

Inflection tables can be generated in two formats, either as text or Latex source code. The text version is given below.

```
$ morpho_lat -tables

filius
Noun
Masculine
```

```
Singular Nominative: filius
Singular Vocative: fili
Singular Accusative: filium
Singular Genitive: fili
Singular Dative: filio
Singular Ablative: filio
Plural Nominative: filii
Plural Vocative: filii
Plural Accusative: filios
Plural Genitive: filiorum
Plural Dative: filiis
Plural Ablative: filiis
```

The generation of tables in Latex source code enables us to create nicer, formatted tables with the program `latex`.

```
$ morpho_lat -latex

filius, Noun Masculine
\begin{center}
\begin{tabular}{|l|l|}\hline
Singular Nominative & {\em filius} \\
Singular Vocative & {\em fili} \\
Singular Accusative & {\em filium} \\
Singular Genitive & {\em fili} \\
Singular Dative & {\em filio} \\
Singular Ablative & {\em filio} \\
Plural Nominative & {\em filii} \\
Plural Vocative & {\em filii} \\
Plural Accusative & {\em filios} \\
Plural Genitive & {\em filiorum} \\
Plural Dative & {\em filiis} \\
Plural Ablative & {\em filiis} \\
\hline
\end{tabular}
\end{center}
% \newpage
```

13.2.1 Grammatical framework (GF)

Grammatical Framework [4] is a multilingual grammar formalism, and because of the translation, we have a direct connection between a lexical resource and syntax, i.e. a GF grammar. GF also requires a type system, which is not exported from FM. The type system, which should corresponds to the type system of the FM implementation, must be in the file `types.latin.gf`.

We have actually cheated a bit with the generation — in our FM implementation we used the possibility to have pretty-printed versions of our types, through the function `prValue` in the class `Param`, where we have removed the constructor `NounForm`. For the generation to be correct, we needed to remove our `prValue` declarations, and recompile FM. It may be reasonable to extend FM to support two kinds of `Dictionary`:s, one with the pretty-printed types and one without, to avoid having to remove `prValue` declarations prior to GF generation.

```
$ morpho_lat -gf

include types.latin.gf ;

cat Noun;

fun filius_1 : Noun ;

lin filius_1 = {s = table {
  NounForm Singular Nominative => "filius" ;
  NounForm Singular Vocative   => "fili" ;
  NounForm Singular Accusative => "filium" ;
  NounForm Singular Genitive   => "fili" ;
  NounForm Singular Dative     => "filio" ;
  NounForm Singular Ablative   => "filio" ;
  NounForm Plural Nominative   => "filii" ;
  NounForm Plural Vocative     => "filii" ;
  NounForm Plural Accusative   => "filios" ;
  NounForm Plural Genitive     => "filiorum" ;
  NounForm Plural Dative       => "filiis" ;
  NounForm Plural Ablative     => "filiis" };
  h1 = Masculine
} ;
```

13.3 XML

The XML [5] format is a way of representing structured information in ASCII. It is the most verbose format of FM, which is typical to an XML representation. However, it is not as bad as it seems, since an XML file may be heavily compressed.

```
$ morpho_lat -xml

<lexicon_entry>
 <dictionary_form value="filius" />
 <inherent value="Masculine" />
 <inflection_table>
  <inflection_form pos="Singular Nominative">
   <variant word="filius" />
  </inflection_form>
  <inflection_form pos="Singular Vocative">
   <variant word="fili" />
  </inflection_form>
  <inflection_form pos="Singular Accusative">
   <variant word="filium" />
  </inflection_form>
  <inflection_form pos="Singular Genitive">
   <variant word="fili" />
  </inflection_form>
  <inflection_form pos="Singular Dative">
   <variant word="filio" />
  </inflection_form>
  <inflection_form pos="Singular Ablative">
   <variant word="filio" />
  </inflection_form>
  <inflection_form pos="Plural Nominative">
   <variant word="filii" />
  </inflection_form>
  <inflection_form pos="Plural Vocative">
   <variant word="filii" />
  </inflection_form>
  <inflection_form pos="Plural Accusative">
   <variant word="filios" />
  </inflection_form>
  <inflection_form pos="Plural Genitive">
   <variant word="filiorum" />
  </inflection_form>
  <inflection_form pos="Plural Dative">
   <variant word="filiis" />
  </inflection_form>
  <inflection_form pos="Plural Ablative">
   <variant word="filiis" />
  </inflection_form>
 </inflection_table>
</lexicon_entry>
```

13.4 XFST

XFST [2] (Xerox Finite State Transducer) source code defines a regular relation, i.e. a relation between two regular languages. A regular relation can be compiled into a finite state transducer, which is an automaton providing a compact and efficient structure for lexical resources. XFST source code is compiled by the XFST tool.

```
$ morpho_lat -xfst

[ {filius} %+Singular  %+Nominative  %+Masculine .x. {filius}]  |
[ {filius} %+Singular  %+Vocative  %+Masculine .x. {fili}]  |
[ {filius} %+Singular  %+Accusative  %+Masculine .x. {filium}]  |
[ {filius} %+Singular  %+Genitive  %+Masculine .x. {fili}]  |
[ {filius} %+Singular  %+Dative  %+Masculine .x. {filio}]  |
[ {filius} %+Singular  %+Ablative  %+Masculine .x. {filio}]  |
[ {filius} %+Plural  %+Nominative  %+Masculine .x. {filii}]  |
[ {filius} %+Plural  %+Vocative  %+Masculine .x. {filii}]  |
[ {filius} %+Plural  %+Accusative  %+Masculine .x. {filios}]  |
[ {filius} %+Plural  %+Genitive  %+Masculine .x. {filiorum}]  |
[ {filius} %+Plural  %+Dative  %+Masculine .x. {filiis}]  |
[ {filius} %+Plural  %+Ablative  %+Masculine .x. {filiis}]
```

13.5 LexC

LexC [2] source code is another, but more restricted, regular relation format designed by Xerox. The restrictions of the format enable the XFST tool to compile the regular relation to a finite state transducer faster and allow better optimizations to be done on the resulting finite state transducer.

```
$ morpho_lat -lexc

filius:filius+Singular+Nominative+Masculine # ;
fili:filius+Singular+Vocative+Masculine # ;
filium:filius+Singular+Accusative+Masculine # ;
fili:filius+Singular+Genitive+Masculine # ;
filio:filius+Singular+Dative+Masculine # ;
filio:filius+Singular+Ablative+Masculine # ;
filii:filius+Plural+Nominative+Masculine # ;
filii:filius+Plural+Vocative+Masculine # ;
filios:filius+Plural+Accusative+Masculine # ;
filiorum:filius+Plural+Genitive+Masculine # ;
filiis:filius+Plural+Dative+Masculine # ;
filiis:filius+Plural+Ablative+Masculine # ;
```

13.6 SQL

SQL, Structured Query Language [1], is a popular source format for defining databases. The first part of the generation creates a table `LEXICON` and defines the types of the elements in the table. We use integers here instead of word identifiers to identify words. The second part simply consists of insertions of data into the table.

```
$ morpho_lat -sql

CREATE TABLE LEXICON
(
ID INTEGER NOT NULL,
DICTIONARY VARCHAR(50) NOT NULL,
CLASS VARCHAR(50) NOT NULL,
WORD VARCHAR(50) NOT NULL,
POS VARCHAR(50) NOT NULL);

INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filius','Singular Nominative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','fili','Singular Vocative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filium','Singular Accusative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','fili','Singular Genitive - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filio','Singular Dative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filio','Singular Ablative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filii','Plural Nominative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filii','Plural Vocative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filios','Plural Accusative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiorum','Plural Genitive - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiis','Plural Dative - Masculine');
INSERT INTO LEXICON VALUES
 ('2','filius','Noun','filiis','Plural Ablative -Masculine');
```

14 Other Commands

14.1 Precompiled Dictionary

When we run FM we always rebuild our dictionary, and if it is large it may take some time. However, if we are only going to use the analyzer, there is a shortcut. A full form lexicon, i.e. a precompiled list of word forms, can be read with the following command.

```
 ./morpho_lat -f latin.fullform
```

The file `latin.fullform` was generated with the full form generation of FM, i.e. with the command `morpho_lat -lex`.

14.2 Print Paradigms

The paradigms of FM are printed with the `-p` flag. The result is similar to generating inflection tables, but with the crucial difference that every paradigm is only printed once and only for those paradigms that have been defined in the command map.

15 The Functional Morphology API

15.1 General.hs

The type for word forms. The type is a list to allow variants and missing word forms.

```
newtype Str = Str [String]
```

The type for polymorphic inflection tables.

```
type Table a = [(a, Str)]
```

The type for finite inflection functions.

```
type Finite a = a -> Str
```

The class for finite parameters.

```
class (Eq a, Show a) => Param a where
 values :: [a]
 value :: Int -> a
 value0 :: a
 prValue :: a -> String
```

The type for token: `W` for normal tokens, `P` for symbols and `D` for digits.

```
data Tok = W String
         | P String
         | D String
```

The attribute type for compounds.

```
type Attr = Int
```

The default attribute value (0).

```
noComp :: Attr
```

Promotes `String` to `Str`.

```
mkStr :: String -> Str
```

Sharing of `Str`:s, achieved by the use of a global hash table.

```
shareStr :: Str -> Str
```

Translate a `Str` to `[String]`.

```
unStr :: Str -> [String]
```

Promotes `[String]` to `Str`.

```
strings :: [String] -> Str
```

Apply function to `a` and promote the resulting `String` to `Str`.

```
mkStr1 :: (a -> String) -> a -> Str
```

Apply function to all variants in `Str`.

```
mapStr :: (String -> String) -> Str -> Str
```

The union of two `Str`.

```
unionStr :: Str -> Str -> Str
```

Prepend a string to all variants in `Str`.

```
(+*) :: String -> Str -> Str
```

Concatenation that marks the morpheme boundaries.

```
(+/) :: String -> String -> String
```

Variants listed in a string. Translated into a list of word by the function `words`.

```
mkStrWords :: String -> Str
```

Takes all but `Int` characters in the end of the string.

```
tk :: Int -> String -> String
```

Drops all but `Int` character in the end of the string.

```
dp :: Int -> String -> String
```

Gets the `Int`:th character from the end of String.

```
ch :: Int -> String -> String
```

Prevents duplication, e.g. `"mus" +? "s" = "mus"`.

```
(+?) :: String -> String -> String
```

Chooses suffix (second and third `String`) depending on the last letter of the first `String`.

```
ifEndThen :: (Char -> Bool) -> String ->
             String -> String -> String
```

Conditionally drops the last letter.

```
dropEndIf :: (Char -> Bool) -> String -> String
```

Apply substitution table to string.

```
changes :: [(String, String)] -> String -> String
```

Like `changes`, but applies only to the prefix.

```
changePref :: [(String, String)] -> String -> String
```

Single word form exception.

```
except :: Param a => Finite a -> [(a, String)] -> Finite a
```

Multiple word form exception.

```
excepts :: Param a => Finite a -> [(a, Str)] -> Finite a
```

Merge two paradigm functions.

```
combine :: Param a => Finite a -> Finite a -> Finite a
```

Missing forms exception.

```
missing :: Param a => Finite a -> [a] -> Finite a
```

Only exception, for highly degenerate paradigms.

```
only :: Param a => Finite a -> [a] -> Finite a
```

Single word form variant exception.

```
variant :: Param a => Finite a -> [(a, String)] -> Finite a
```

Multiple word form variants exception.

```
variants :: Param a => Finite a -> [(a, Str)] -> Finite a
```

Missing word form.

```
nonExist :: Str
```

Filters missing forms from inflection table.

```
existingForms :: Table a -> Table a
```

Translates a finite function to a table.

```
table :: Param a => (a -> Str) -> Table a
```

Used to define Param instances.

```
enum :: (Enum a, Bounded a) => [a]
```

A function with the same functionality as `fromEnum`, but for Param.

```
indexVal :: (Eq a, Param a) => a -> Int
```

Lookup in an inflection table.

```
appTable :: Param a => Table a -> a -> Str
```

Selects the first word form in an inflection table.

```
firstForm :: Param a => Table a -> Str
```

Creates a function from list of values (sensitive to order).

```
giveValues :: (Eq a, Param a) => [Str] -> a -> Str
```

Longest common prefix for a list of strings.

```
longestPrefix :: [String] -> String
```

Collects all word forms into a `Str`.

```
formsInTable :: Table a -> Str
```

Apply function to all word forms in table.

```
mapInTable :: (String -> String) -> Table a -> Table a
```

15.2 Dictionary.hs

An instance of the `Dict` class provides information on how to construct an entry for a given dictionary type. In particular, it associates a word class identifier to the dictionary type.

```
class Param a => Dict a where
 dictword :: (a -> Str) -> String
 category :: (a -> Str) -> String
 defaultAttr :: (a -> Str) -> Attr
 attrException :: (a -> Str) -> [(a, Attr)]
```

The type for dictionaries.

```
data Dictionary = D [Entry]
```

The type for a dictionary entry.

```
type Entry = (Dictionary_Word,
      Paradigm,
              Category,
              [Inherent],
              Inflection_Table,
              Extra)
```

The type for paradigm identifiers.

```
type Paradigm = String
```

Transforms a typed table to an untyped.

```
prTable :: Param a => Table a -> Table String
```

Removes attributes and extra information from a dictionary.

```
removeAttr :: Dictionary -> [EntryN]
```

The type for full form lexica: a list of word forms together with their analyses and compound attributes.

```
type FullFormLex = [(String, [(Attr, String)])]
```

Group a dictionary into categories; reverses the entries.

```
classifyDict :: Dictionary -> [(Category, [Entry])]
```

Removes attributes and extra information from `Entry`.

```
noAttr :: Entry -> EntryN
```

Translates an inflection function to an `Entry`.

```
entry :: Dict a => (a -> Str) -> Entry
```

Translates an inflection function with inherent information to an `Entry`.

```
entryI :: Dict a => (a -> Str) -> [Inherent] -> Entry
```

Translates an inflection function with extra information to an `Entry`.

```
entryWithInfo :: Dict a => (a -> (Str, Str)) -> Entry
```

Translates an inflection function with extra information and inherent information to an `Entry`.

```
entryWithInfoI :: Dict a => (a -> (Str, Str)) ->
                            [Inherent] -> Entry
```

Translates an inflection function with paradigm identifier to an `Entry`.

```
entryP :: Dict a => (a -> Str) -> Paradigm -> Entry
```

Translates an inflection function with inherent information and paradigm identifier to an `Entry`.

```
entryIP :: Dict a => (a -> Str) ->
                     [Inherent] -> Paradigm -> Entry
```

Translates an inflection function with extra information and paradigm identifier to an `Entry`.

```
entryWithInfoP :: Dict a => (a -> (Str, Str)) ->
                            Paradigm -> Entry
```

Translates an inflection function with extra information, inherent information, and paradigm identifier to an `Entry`.

```
entryWithInfoIP :: Dict a => (a -> (Str, Str)) ->
                             [Inherent] -> Paradigm -> Entry
```

An `Entry` without attributes and extra information.

```
type EntryN = (Dictionary_Word,
               Category,
               [Inherent],
               [(Untyped, Str)])
```

Creates a `Dictionary`.

```
dictionary :: [Entry] -> Dictionary
```

Translate a `Dictionary` to a list of `Entry`:s.

```
unDict :: Dictionary -> [Entry]
```

The number of entries in a dictionary.

```
size :: Dictionary -> Int
```

The number of word forms in a dictionary.

```
sizeW :: Dictionary -> Int
```

Concatenates two dictionaries.

```
unionDictionary :: Dictionary -> Dictionary -> Dictionary
```

Concatenates a list of Dictionaries.

```
unionDictionaries :: [Dictionary] -> Dictionary
```

An empty Dictionary.

```
emptyDict :: Dictionary
```

Translates a `Dictionary` to a `FullFormLex`.

```
dict2fullform :: Dictionary -> FullFormLex
```

A full form lexicon structured around the word identifier.

```
dict2idlex :: Dictionary -> FullFormLex
```

Performs sharing on the strings in the `Dictionary`.

```
shareDictionary :: Dictionary -> Dictionary
```

15.3 Print.hs

Prints word forms in `Str`, separated with '/'.

```
prStr :: Str -> String
```

Similar `prStr`, but outputs '*' for missing word forms.

```
prAlts :: Str -> String
```

Creates a constant table.

```
consTable :: Str -> Table String
```

Creates an attributed constant table.

```
consTableW :: Str -> [(String, (Attr, Str))]
```

Print a `show`:ed inflection function to standard output.

```
putFun0 :: Param a => (a -> Str) -> IO ()
```

Print an inflection function to standard output.

```
putFun :: Param a => (a -> Str) -> IO ()
```

Translate a `show`:ed parameter value to one without parenthesis.

```
prFlat :: String -> String
```

Shows all values for the first parameter.

```
prFirstForm :: Param a => Table a -> String
```

Shows one value for the first parameter.

```
prDictForm :: Param a => Table a -> String
```

Another `Str` printing function.

```
prDictStr :: Str -> String
```

Prints a dictionary, removing the attributes.

`prDictionary :: Dictionary -> String`

Prints a dictionary in a structured format.

`prNewDictionary :: Dictionary -> String`

Writes a full form lexicon to handle.

`prFullFormLex :: Handle -> FullFormLex -> IO ()`

Prints attribute to handle.

`prCompAttr :: Handle -> Attr -> IO ()`

Generates GF paradigm functions.

`prGFRes :: Dictionary -> String`

Prints GF source code.

`prGF :: Dictionary -> String`

Generates XML source code.

`prXML :: Dictionary -> String`

Prints LexC source code.

`prLEXC :: Dictionary -> String`

Prints XFST source code.

`prXFST :: Dictionary -> String`

Prints latex tables.

`prLatex :: Dictionary -> String`

Prints SQL Code.

`prSQL :: Dictionary -> String`

15.4 Frontend.hs

The runtime system class.

```
class Show a => Language a where
 name        :: a -> String
 dbaseName   :: a -> String
 composition :: a -> [Attr] -> Bool
 env         :: a -> String
 paradigms   :: a -> Commands
 internDict  :: a -> Dictionary
 tokenizer   :: a -> String -> [Tok]
 wordGuesser :: a -> String -> [String]
```

The type for command maps.

```
type Commands = Map String ([String], [String] -> Entry)
```

An empty command map.

```
emptyC :: Commands
```

Inserts a command into a set of commands.

```
insertCommand :: (String, [String], [String] -> Entry) ->
                 Commands -> Commands
```

Constructs a command map.

```
mkCommands :: [(String, [String], [String] -> Entry)] ->
              Commands
```

Creates a dictionary from the list of paradigms.

```
command_paradigms :: Language a => a -> Dictionary
```

Parses commands.

```
parseCommand :: Language a => a -> String -> Err Entry
```

Lists paradigm names.

```
paradigmNames :: Language a => a -> [String]
```

The number of paradigms.

```
paradigmCount :: Language a => a -> Int
```

Reading external lexicon. Creates empty lexicon if the file does not exist.

```
parseDict :: Language a => a -> FilePath ->
                           IO (Dictionary, Int)
```

Is input string a paradigm name?

```
isParadigm :: Language a => a -> String -> Bool
```

Reads external lexicon.

```
readdict :: Language a => a -> FilePath ->
                          IO ([Entry], Int)
```

Removes comments in String.

```
remove_comment :: String -> String
```

Wrapper functions for the command map.

```
app1 :: (String -> Entry) -> [String] -> Entry
app2 :: (String -> String -> Entry) -> [String] -> Entry
app3 :: (String -> String -> String ->
         Entry) -> [String] -> Entry
app4 :: (String -> String -> String ->
         String -> Entry) -> [String] -> Entry
app5 :: (String -> String -> String -> String ->
         String -> Entry) -> [String] -> Entry
app6 :: (String -> String -> String -> String ->
         String -> String -> Entry) -> [String] -> Entry
app7 :: (String -> String -> String -> String ->
         String -> String -> String -> Entry) ->
         [String] -> Entry
```

Prints to `stderr`.

```
prErr :: String -> IO ()
```

15.5 GeneralIO.hs

Outputs UTF8-encoded string.

```
putStrLnUTF8 :: String -> IO ()
```

Writes source format to file.

```
writeLex    :: FilePath -> Dictionary -> IO ()
writeTables :: FilePath -> Dictionary -> IO ()
writeGF     :: FilePath -> FilePath -> Dictionary -> IO ()
writeGFRes  :: FilePath -> FilePath -> Dictionary -> IO ()
writeXML    :: FilePath -> Dictionary -> IO ()
writeXFST   :: FilePath -> Dictionary -> IO ()
writeLEXC   :: FilePath -> Dictionary -> IO ()
writeLatex  :: FilePath -> Dictionary -> IO ()
writeSQL    :: FilePath -> Dictionary -> IO ()
```

The analysis function.

```
analysis :: ([Attr] -> Bool) -> String -> [[String]]
```

Lookup identifiers for a word form.

```
lookupId :: String -> [String]
```

The synthesiser function.

```
synthesiser :: Language a => a -> IO ()
```

The inflection mode function.

```
infMode :: Language a => a -> IO ()
```

The batch inflection mode function.

```
imode :: Language a => a -> IO ()
```

15.6 CommonMain.hs

The 'main' function of FM.

```
commonMain :: Language a => a -> IO ()
```

A type for statistics.

```
data Stats = Stats {totalWords :: Int,
                    coveredWords :: Int}
```

Empty statistics.

```
initStats :: Stats
```

15.7 CTrie.hs

Constructs a c-trie from a file containing a full form lexicon.

```
buildTrie :: FilePath -> Bool -> IO ()
```

Constructs a C-trie from a Dictionary ADT. Note that the trie is not handled in Haskell, it's a global object in C.

```
buildTrieDict :: Dictionary -> Bool -> IO ()
```

```
buildTrieDictSynt :: Dictionary -> Bool -> IO ()
```

Builds an undecorated trie.

```
buildTrieWordlist :: [String] -> Bool -> IO ()
```

```
trie_lookup :: String -> [(Attr, String)]
```

Is the string a member in the trie?

```
isInTrie :: String -> Bool
```

Function for compound analysis.

```
decompose :: ([Attr] -> Bool) -> String -> [[(Attr, String)]]
```

References

[1] *ISO/IEC 9075 Information Technology–Database Languages–SQL*. 1999.

[2] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI Publications, Stanford University, United States, 2003.

[3] G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005. `http://yquem.inria.fr/~huet/PUBLIC/tagger.pdf`.

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

[5] The World Wide Web Consortium. Extensible Markup Language (XML). `http://www.w3.org/XML/`, 2000.