User Guide: the *extract* tool (v1.0Beta4)

Markus Forsberg
Department of Computing Science
Chalmers University of Technology and the University of Gothenburg
SE-412 96 Gothenburg, Sweden
markus@cs.chalmers.se

October 11, 2006

Abstract

This document describes how to use the tool *extract* to automatically extract lemma-paradigm pairs for a target morphology. The tool was developed primarily for morphologies defined in Functional Morphology[1] (FM), but is useful for all similar systems that implement a word-and-paradigm description of a morphology.

The *extract* tool can be viewed as an advanced search tool, which uses regular expressions and propositional logic to classify words into paradigms.

1 Introduction

1.1 Functional Morphology

Functional Morphology is a system for developing morphologies. The FM system uses a full-fledged functional programming language as the development environment. The system requires that the morphology description denotes a (language-independent) `Dictionary` data structure (defined in FM), and by doing this, it can support translations to analyzers, synthesizers and a multiple of source formats for other system, such as SQL, GF, LexC, XFST and Latex.

Simply put, a morphology in FM consists of definitions of inflection tables, *paradigms*, defined as functions. The lexicon of the morphology is a listing of the target language's dictionary forms, *lemmas*, applied to the appropriate paradigm function.

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

An example of a paradigm is the Swedish first declension nouns, which is inflected in number, definiteness and case. Here illustrated with the word *lampa* (*Eng. lamp*).

	Singular	**Plural**
Indef Nom	*lampa*	*lampor*
Indef Gen	*lampas*	*lampors*
Def Nom	*lampan*	*lamporna*
Def Gen	*lampans*	*lampornas*

A possible description of the first declension paradigm in FM follows, here presented with the function `decl1`.

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

decl1 :: String -> Paradigm
decl1 word (N c d n) =
  let stem = tk 1 word in
    mkCase $
       case (d,n) of
        (Indef,Singular) -> stem ++ "a"
        (Def,Singular)   -> stem ++ "an"
        (Indef,Plural)   -> stem ++ "or"
        (Def,Plural)     -> stem ++ "orna"
```

Without going into details, one can notice that there is a way of representing the inflectional parameters and that the case distinction has been abstracted into a separate function.

Now that the inflection table of first declension Swedish nouns has been represented as a paradigm function, a small lexicon can be created by giving a list of lemmas applied to the paradigm function.

decl1	“älva”	(*Eng. fairy*)
decl1	“ånga”	(*Eng. steam*)
decl1	“ärta”	(*Eng. pea*)
decl1	“blåsa”	(*Eng. blister*)

1.2 Automatic Lexicon Extraction

Instead of manually list words in the respective paradigms, let us consider extracting the words automatically with the *extract* tool. The initial idea is simple: start with raw text data and a description of the forms in the paradigms with the varying parts, also known as the *technical stem*, represented with variables. In the *extract* tool syntax, one could write the following definition to describe the Swedish first declension paradigm.

```
paradigm decl1 =
      x+"a"
      {x+"a" & x+"as" & x+"an" & x+"ans" &
       x+"or" & x+"ors" & x+"orna" & x+"ornas"} ;
```

Given that all forms in the curly brackets, the **constraint**, are found for some substring `x`, the tool outputs the **head** `x+"a"` tagged with the name of paradigm. For example, if the following forms exist in the corpus: *ärta*, *ärtas*, *ärtan*, *ärtans*, *ärtor*, *ärtors*, *ärtorna* and *ärtornas*, the tool would output `decl ärta`.

We can now run the tool, with the definition above in a file `paradigm.para` and with some Swedish text data `Swedish_text`, by the command:

```
$ extract paradigm.para Swedish_text
```

However, while looking for all the forms in a paradigm, it becomes clear that it is actually quite uncommon for a lemma to appear in all forms and furthermore, it is seldom a simple task to select the forms which are the most representative for the particular paradigm. For example, in the definition above, it does not really matter if the word in nominative or genitive case and it is too restrictive just to search for one of the cases. To enable more fine-grained descriptions, the tool supports propositional logic in the constraint, described in the section 2.1.

Another drawback of the description above is the lack of control of what `x` might be. Section 2.2 describes how the tool improve this situation with regular expressions.

The technical stem in the description above is the same for all forms, which is not the case for many paradigms, e.g. paradigms with umlaut. Section 2.3 presents the tool's use of multiple variables as a solution to this problem.

2 Paradigm File Format

A paradigm file consists of two kinds of definitions: `regexp` and `paradigm`.

```
regexp NAME = RegEXP ;

paradigm NAME VarDef = HEAD CONSTRAINT ;
```

A `regexp` definition associates a name (`NAME`) with an regular expression. A `paradigm` definition consists of a name (`NAME`), a set of variable bindings (`VarDef`), a set of output constituents (`HEAD`) and search pattern (`CONSTRAINT`).

The basic unit in `HEAD` and `CONSTRAINT` is a *pattern* that describes a word form. A pattern consists of a sequence of variables and string literals glued together with the '+' operator. A previous example of a pattern was `x+"a"`.

The definitions above will be discussed in detail in the following sections.

2.1 Propositional Logic

Propositional logic is used in the constraint to enable a more fine-grained description of what word forms the tool should look for. The basic unit is a pattern, corresponding to a word form, which is combined with the *and operator* `&`, the *or operator* `|` and the *negation operator* `~`.

The syntax for is given below, where *Pattern* refers to one word form.

$$\begin{array}{lll}
\langle \textit{Logic} \rangle & ::= & \langle \textit{Logic} \rangle \ \texttt{\&} \ \langle \textit{Logic1} \rangle \\
 & | & \langle \textit{Logic} \rangle \mid \langle \textit{Logic} \rangle \\
 & | & \langle \textit{Logic} \rangle \\
 & | & \texttt{\~{}} \ \langle \textit{Logic} \rangle \\
 & | & \langle \textit{Pattern} \rangle \\
 & | & \texttt{(} \ \langle \textit{Logic} \rangle \ \texttt{)}
\end{array}$$

With the propositional logic, the paradigm in section 1.2 can be rewritten to reflect that it is sufficient to find a word form either in the nominative or genitive case by the use of the `|` operator.

```
paradigm decl1 =
      x+"a"
      {( x+"a"  | x+"an"  | x+"or"  | x+"orna") &
       ( x+"as" | x+"ans" | x+"ors" | x+"ornas")} ;
```

2.2 Regular Expressions

In section 1.2 it was mentioned that increased control over what (sub)strings the variables in a paradigm may be associated with was required to enable a more precise description. The solution that the tool provides is to enable the user to associate every variable with a regular expression, which describe which strings the variable actually can match. An unannotated variable can match any string, i.e. its regular expression is Kleene star over all characters.

Consider German nouns, which always start with an uppercase letter. This could be describe in the tool with the following definition.

```
 regexp UpperWord = upper letter*;

 paradigm n [x:UpperWord] = ... ;
```

The syntax of the tool's regular expression is given below, with the normal connectives: union, concatenation, set minus, Kleene's star, Kleene's plus and optionality. *eps* refers to the empty string, *digit* to $0-9$, *letter* to an alphabetic character, *lower* and *upper* to lowercase and uppercase. *char* refers to any character. A regular expression can also contain a double quoted string, which is interpreted as the concatenation of the characters in the string.

$$\begin{array}{lcl}
\langle \mathit{Reg} \rangle & ::= & \langle \mathit{Reg} \rangle \mid \langle \mathit{Reg} \rangle \\
 & | & \langle \mathit{Reg} \rangle - \langle \mathit{Reg} \rangle \\
 & | & \langle \mathit{Reg} \rangle\ \langle \mathit{Reg} \rangle \\
 & | & \langle \mathit{Reg} \rangle\ \texttt{*} \\
 & | & \langle \mathit{Reg} \rangle\ + \\
 & | & \langle \mathit{Reg} \rangle\ \texttt{?} \\
 & | & \texttt{eps} \\
 & | & \langle \mathit{Char} \rangle \\
 & | & \texttt{digit} \\
 & | & \texttt{letter} \\
 & | & \texttt{upper} \\
 & | & \texttt{lower} \\
 & | & \texttt{char} \\
 & | & \langle \mathit{String} \rangle \\
 & | & \texttt{(}\ \langle \mathit{Reg} \rangle\ \texttt{)}
\end{array}$$

2.3 Multiple Variables

Not all paradigms are as neatly as the initial example — phenomena like *umlaut* requires an increased control over the variable part. The solution the tool provides is to allow multiple variables, i.e. that a pattern can have more than one variable. This is best explained with an example, here with the paradigm of two German nouns.

```
regexp Pre = upper Whatever;

regexp Whatever = letter*;

paradigm n2 [F:Pre, ll:Whatever] =
  F+"a"+ll
    {F+"a"+ll & F+"ä"+ll+"e"} ;

paradigm n3 [W:Pre, rt:Whatever] =
   W+"o"+rt
    {W+"o"+rt & W+"ö"+rt+"er"} ;
```

The use of variables may greatly reduce the performance of the tool. The reason for this is that every possible variable binding is considered, and if there are many variables, the search space could be huge. To remedy this situation, use multiple variables with care and restrict them as much as possible with regular expressions.

It is not required that all variables occurs in every pattern, but the tool only perform an initial match with patterns that contains all variables. The reason for this is efficiency — the tool only consider one word at the time, and if the word matches one of the patterns, it searches for all other patterns with the variables instantiated by the initial match. By obvious reasons, an initial match is never performed under a negation (this would imply that the tool searches for something it does not want to find!).

2.4 Multiple Arguments

The head of a paradigm definition may also have multiple arguments. This to support more abstract paradigms. An example is Swedish nouns, where many nouns can be correctly classified by just analyzing the word forms in nominative singular and nominative plural. In the example below, the first and second declension is handled with the same paradigm function, and the head consists of two variables. The constraints are omitted.

```
paradigm regNoun =
        flick+"a" flick+"or"
        {...} ;

paradigm regNoun =
        pojk+"e" pojk+"ar"
        {...} ;
```

3 Adding Additional Information

There are in two senses on may be interested in adding additional information to the tool - adding preprocessing information to the words (such as POS tags or context information) and adding information to the output, such as the word class, to simplify the translation into another format that requires this extra information. Adding information to the output is simply done by adding an extra head with information. For example:

```
paradigm regNoun =
        flick+"a" flick+"or" "Noun"
        {...} ;
```

Adding preprocessing information is rather ad hoc in the current version. Let us consider how to incorporate the result of a POS tagger in the `decl1` rule. A solution is just glueing the information to the word strings and rewrite the rule to reflect this addition.

```
paradigm decl1 =
      x+"a"
      {(x+"a-N"  | x+"as-N") &  (x+"an-N"   | x+"ans-N") &
       (x+"or-N" | x+"ors-N") & (x+"orna-N" | x+"ornas-N")} ;
```

This rule describes strings like `flicka-N`. This solution does the job, but the description is not as easy to read as before.

4 Command-line Options

This section describes the command-line options of the tool.

```
extract [Option(s)] paradigm_file  Corpus_File
```

The `Option(s)` is given below.

-h Display the help message.

-utf8 Use UTF-8 encoding. Both the corpus file and the paradigm file must be encoded in UTF-8.

-nobad Only the analyzed words are output.

-uncap Transforms the whole input corpus to lowercase.

-nocap Throw away all capitalized words.

-e Output the evidence of a constraint (the word forms) as a comment after the result.

-u Do not output duplicates, i.e. a history of previous results is kept.

-id Label all output with an integer that corresponds to a particular paradigm. Used to distinguish rules with the same name.

-r Reverse lexicon. May give space and/or time improvement for some suffix-heavy languages.