PFME: A Paradigm Functional Morphology Engine

Raphael Finkel
raphael@cs.uky.edu
Dept. of Computer Science
University of Kentucky

Gregory Stump
gstump@pop.uky.edu
Dept. of English
University of Kentucky

January 8, 2013

1 Introduction

Gregory Stump introduced **Paradigm Functional Morphology (PFM)** as a way to express the morphology of natural languages. [1]. Raphael Finkel built **PFME**, a web-based engine that generate word forms from language theories expressed in PFM. This document describes the features of PFME, which is accessible at `http://www.cs.uky.edu/~raphael/linguistics/pfm.cgi`. This engine belongs to a suite of web-accessible tools for computational linguistics called *Cats Claw: Computer-Assisted Technology Service; Computational Linguist's Automated Workbench*, accessible at `http://www.cs.uky.edu/~raphael/linguistics/claw.html`.

In the following, we distinguish "must", "should", and "may" to denote whether an item is obligatory, conventional, or optional.

2 Getting started

PFME allows the user to enter a theory describing a language in three different ways.

- Uploading from a file
- Entering into a text area (preferably by copy and paste)
- Filling in a collection of forms

The easiest way to start is to download one of the PFM theories listed in the PFME web page and then upload it to PFME via the *submit file* button. These theories are intended to display PFME's function, not linguistic theories of the languages represented. Not all provide correct results or even reasonable

approaches to the problems they try to solve. However, they are generally syntactically correct input to the PFM calculator.

After receiving a PFM theory, PFME displays (1) the output generated by the theory, a filled-in text area containing the theory, which you can modify and re-submit with the *submit text* button, and (3) a filled-in form, which you can modify and re-submit with the *submit form* button.

PFM theories must be encoded in Unicode UTF-8. In particular, rules use the σ and "→" symbols (Unicode \u03c3 and Unicode \u2192). In addition, theories may use any Unicode characters in representing lexemes and derived forms, so it is straightforward to generate IPA or words in non-Latin alphabets.

3 Notation

We need to introduce some technical terms in order to describe the notation for PFM theories.

3.1 Expandables

Many components of a PFM theory are expressed by a shorthand that we call an **expandable**. Here is an example:

```
sg/pl 1/2/3 masc/fem
```

This example expands to a list containing the following pieces:

```
sg 1 masc
sg 1 fem
sg 2 masc
sg 2 fem
sg 3 masc
sg 3 fem
pl 1 masc
pl 1 fem
pl 2 masc
pl 2 fem
pl 3 masc
pl 3 fem
```

An expandable is a space-separated list of elements. Each element is a solidus ("/")-separated list of alternatives. Each alternative is either a parenthesized expandable, an identifier (such as `masc` or `3` in the example above), or a supertoken. Here are some sample supertokens:

```
AGR(subject):{1/2/3 masc/fem}
TENSE:{past/present/future}
```

Supertokens must have a name (conventionally in upper case), a colon, and braces surrounding an expandable; they may include a parenthesized subname.

3.2 Contexts

A **context** is a package containing information about a particular lexeme and a morphosyntactic property set (MPS). For instance, in English we might be interested in forming the lexeme "eat" in the third-person singular present. PFM theories represent a context by a pair in pointy brackets, such as `<L, σ>`, where `L` represents the lexeme and σ represents the MPS.

3.3 Matching

PFME often needs to match components of contexts against patterns. A component could be composed of a lexeme's syntactic category and inflection class, or it could be composed of an MPS. The pattern is an expandable. We say the component **matches** the pattern (or, equivalently, that the pattern **matches** the component) if at least one of the alternatives generated by the pattern has elements all of which appear in the component. The **strength** of the match is the number of such elements that appear.

3.4 Paṇīni precedence

When PFME must choose among alternative patterns in some context, it uses Paṇīni precedence, which follows these steps:

1. Determine which alternatives are **available**: those that the context matches.
2. Of the available alternatives, select the most **restrictive** alternative: those whose match has the highest strength.

For instance, in a context whose MPS is {1 sg past fem}, consider these alternatives:

```
a.  1 sg fut fem
b.  1 sg fem
c.  sg fem
d.  sg past
```

Of these four alternatives, only `b`, `c`, and `d` are available. Alternative `a` is unavailable because it requires the component `fut`, which is not present in the MPS. Of the available alternatives, `b` is the most restrictive, matching three of the four components of the MPS. Alternatives `c` and `d` are less restrictive, matching only two of the four components of the MPS.

4 PFM components

The components of a PFM theory may be presented in any order. PFME simply ignores any part of the PFM theory that does not correspond to a recognized component, so the theory can include comments without specially delimiting them. The special character "%" introduces a comment that continues to the end of the line. Lines may be terminated in the Unix fashion (with newline), in the Win32 fashion (carriage return, newline), or in the Macintosh fashion (carriage return).

4.1 Language name

A PFM theory should specify the language it represents by a line like this:

```
Language:  Name
```

4.2 Paradigm function

A PFM theory must specify a single paradigm function that is to apply to all lexemes. Here is a sample paradigm function:

```
Paradigm Function
   PF(<L,σ>) = person(tense(I(Stem(<L,σ>))))
```

The function must have a line saying `Paradigm Function`. This particular function says that the way to generate a surface form from the context `<L,σ>` is to first find the appropriate stem, then apply an appropriate rule from block `I`, then a rule from block `tense`, then a rule from block `person`. Any word may name a block, although it is conventional to name blocks either by Roman numerals (like `I`) or by names of morphosyntactic properties (like `person`).

4.3 Lexical entries

Each lexeme must be described by a lexical entry like the following:

```
Lexeme:  EAT
Meaning:  eat
Syntactic category:  V
Inflection class:  n
```

The lexeme name should be in upper case, and one should use `V`, `N`, and `A` for verbs, nouns, and adjectives. The `Lexeme` and `Syntactic category` must be a single word. The `Meaning` may be several words. The inflection class is an expandable.

4.4 Stem-selection rules

Each lexeme must have one or more associated stems. Stems are defined by syntax like the following.

```
Stem(<EAT, S:{past}>) = <ate, S>
Stem(<EAT, S:{perfect/futPerf}>) = eaten
Stem(<EAT, S:{}>) = eat
Stem(<CLIMB, σ:{}>) = climb
Stem(PERFORM) = perform
```

This example demonstrates a variety of acceptable formats. The first line shows the most general format, where both the left-hand side and the right-hand side use full context notation. The other examples use various acceptable shorthands. Stem formats obey these rules:

- The supertoken expandable denoting the MPS, if present, should either use the name `S` or σ.
- The same supertoken name must be used on the right-hand side, if present, as on the left-hand side.
- The lexeme component on the right-hand side must be a single word.

PFME uses Paṇīni precedence to select the stem appropriate to a context.

4.5 Paradigm Schemata

A PFM theory must have at least one paradigm schema; it may have several. A simple paradigm schema looks like this:

```
ParadigmSchema(V) = {
   present/past/perfect/future/futPerf sg/pl 1/2/3
}
```

The first line must have a parenthesized, non-empty expandable pattern that may refer to syntactic categories and inflection classes. The bracketed right-hand side is an expandable that generates a list of MPSs.

Given a lexeme, PFME finds all paradigm schemata whose pattern matches the syntactic category and inflection class of the lexeme. PFME generates all MPSs from those matching schemata.

A complex paradigm schema may expand to several paradigm schemata. Here is an example taken from a theory of nouns and adjectives in Noon (Niger-Congo, Senegal):

```
ParadigmSchema(N <\d>A) = {
   CLASS:{$1}
   NUM:{sg/pl}
   DEF:{plus/minus}
```

```
LOC:{1/2/3/noLoc}
POSS:{(1 sg)/(1 pl incl/excl)/(2/3 sg/pl)/noPoss}
REL:{noRel}
```

The presence of an expression bracketed by < and > in the pattern indicates expansion. The special characters \d represent any number $0 \ldots 9$. The later use of $1 in the right-hand side refers back to that bracketed expression. In this language, nouns have inflection classes 1A ... 6A. The paradigm schema allows each noun to gain an MPS supertoken called CLASS containing a number in $1 \ldots 6$.

4.6 Disallowed MPSs

The MPS list that PFME produces from the paradigm schemata may include some unwanted combinations. For instance, in Noon nouns, possession requires definiteness, so we don't want to generate MPSs that contain a POSS other than noPoss if we have DEF:{minus}. We indicate unacceptable combinations by disallow schemata, which follow the same rules as paradigm schemata. For instance, we can have:

```
Disallow(N) = {
   (POSS:{sg/pl} DEF:{minus}) /
   (LOC:{1/2/3} DEF:{minus})
}
```

This particular schema enforces the rules that possession requires definiteness, and location requires definiteness.

PFME must check every generated MPS against the list of disallowed MPSs, so where possible, it is better to use restrictive expandables in the paradigm schema instead of listing disallowed entries. In the example above, for instance, we have chosen to indicate

```
POSS:{(1 sg)/(1 pl incl/excl)/(2/3 sg/pl)/noPoss}
```

instead of allowing

```
POSS:{(1/2/3 sg/pl incl/excl/nocl)/noPoss}
```

and disallowing

```
POSS:{(sg incl/excl) / (2/3 pl incl/excl)}
```

4.7 Rules of exponence

The paradigm function indicates at least one block of rules of exponence. Individual rules can themselves refer to other blocks. A **block** of rules looks like the following:

```
Block I
   I, X,S:{3 sg present} → Xs
   I, X[weak],S:{perfect/past/futPerf} → Xed
```

Every block implicitly contains the **default rule**:

```
   blockName, X[],S:{} → X
```

The block must begin with a `Block` line and be followed by the rules appropriate to that block. Each **rule** must start with the block name, comma, and the letter `X`. The `X` refers to the argument to the rule, typically a partial surface form. Following the `X` is an optional bracketed expandable, which we call the **classifier**. If there is no bracketed expandable, the classifier is considered to be empty.

The left-hand side concludes with the **MPS component**, which is a supertoken named either `S` or σ. Within this supertoken is an expandable. Entire bracketed subparts may be abbreviated by a single Greek letter. For instance, the supertoken may look like this:

```
S:{transitive AGR(SUBJ):{τ} AGR(OBJ):{σ}}
```

The left-hand side and right-hand side are separated by →.

The right-hand side is composed of Unicode characters, which may contain special forms:

- The letter `X`, standing for the argument to the rule. `S` is the name of the supertoken on the left-hand side.
- An embedded expression, such as `(Negator(Mood(<X,S>)))`, which acts as a referral to a subordinate paradigm function, in this case, invoking first the block `Mood` and then the block `Negator`. Embedded expressions must be surrounded by parentheses, and references to `X` must be of the form `<X,S>`.
- Embedded expressions may introduce a new context and may refer to abbreviations from the left-hand side:

  ```
  (II(<X, τ>)) (IV(<X, σ>))
  ```

  In this example, the right-hand side invokes two blocks, each in a new context. The block `II` only uses the τ part of the MPS component from the left-hand side; the block `IV` only uses its σ part.

Parentheses and the letter `X` are not allowed in the right-hand side except as described here.

PFME evaluates blocks in an order determined by the paradigm function and referrals from rules. It evaluates a block with respect to a context, which includes the lexeme and a given MPS. In evaluating a block, PFME selects the appropriate rule as follows.

1. Discard those rules in the block that do not have highest Paṇīni precedence based on comparing their classifiers with the syntactic category and inflection class of the lexeme. It is permissible to retain multiple rules.

2. Of the retained rules, select that rule with the highest Paṇīni precedence based on MPS component. There must be exactly one such rule (possibly the default rule) or the PFM theory is erroneous.

5 Sandhi

A PFM theory may include rules of Sandhi, including shorthands for phonological classes. For example:

```
PhonologicalClass voiceless = f k p t


Sandhi {
   z → s / [voiceless]_
}
```

The theory may contain any number of `PhonologicalClass` commands; each must by on a line by itself. There may be only one `Sandhi` section, which must contain a bracketed set of rules. Each rule is of the form

original → *replacement* / *when*

Such a rule indicates that the string indicated by *original* is to be replaced by the string indicated by *replacement* under the situation indicated by *when*. The *when* string must have a single underscore (_), which represents the original string; to its left and right may be indicators specifying the environment surrounding the original string in order for the rule to apply. These indicators are **enhanced strings**, which are strings that may contain phonological class shorthands, which must be enclosed in square brackets. In the example above, the rule specifies that `z` converts to `s` if it is preceded by a voiceless letter, which is any of `f`, `k`, `p`, or `t`.

The *replacement* may be ∅ (Unicode \u00d8 or Unicode \u2205) to indicate that PFME should simply delete the *original* in the given environment.

If the `Sandhi` section includes multiple rules, they are applied in the order shown; later rules can therefore further modify forms that earlier rules have produced.

6 Truth

A PFM theory may include a section showing known forms for some lexemes and MPSs. For instance, we can say:

```
Truth = {
   EAT:{1 sg present} = I eat
   EAT:{3 sg present} = he eats
   EAT:{3 sg past} = he ate
   EAT:{3 sg perfect} = he has eaten
   EAT:{1 pl perfect} = we have eaten
   EAT:{3 sg futPerf} = he will have eaten
   CLIMB:{1 sg past} = I climbed
   CLIMB:{3 pl perfect} = they have climbed
}
```

Each entry must be on a single line by itself. It contains a lexeme name, a colon, a bracketed expandable (representing one or more MPSs), an equal sign, and the expected output. PFME checks all results that are expected by the `Truth` section and indicates if they are as expected or not.

If the theory contains the line

```
ShowOnlyTruth
```

then PFME will only run the lexemes and MPSs indicated by the `Truth` section.

7 Randomization

A PFM theory may include a line like

```
Random 10
```

to indicate that PFME is not to generate all possible MPSs specified by the paradigm schemata and not disallowed, but is rather to only randomly generate a limited number (in this case, ten) MPSs for each lexeme. This facility is especially useful if the number of possible MPSs is very large, because PFME operates under a time limit.

PFME ignores a randomization request if the theory also requests `ShowOnlyTruth`.

8 Output control

If the PFM theory contains this line:

```
notInteractive
```

then PFME suppresses interactive features, including a text box and a form in which the user can enter or modify the PFM theory. These features are experimental and incomplete in any case.

9 How PFME works

PFME first accepts the given PFM theory and parses it. For each lexeme, it consults the paradigm schemata and the disallow schemata (or, if the theory specifies `ShowOnlyTruth`, the `Truth` set) to produce MPSs (either all possible MPSs or a random selection, if the theory specifies randomization), which it packages with the lexeme into contexts that it calls **queries**. For each query, PFME applies the paradigm function, which invokes stem-selection rules and blocks. For each block that it applies, PFME selects the single best rule. If the block is ambiguous, that is, there are several best rules, but they all agree on their right-hand side, the ambiguity is innocuous and ignored; otherwise, PFME arbitrarily picks a best rule but flags the error. After it has finished applying the paradigm function to the context, PFME applies all sandhi rules to the result.

The output of PFME is a web page with one section per lexeme expressed as a table. The table section contains one line per query. The line has cells indicating the query, the chosen stem, and each block that is consulted. For each block, PFME either indicates `ditto` if it uses the default rule or the number of the rule it chooses. It then applies any sandhi rule to the result and displays it. Here are the cells in one sample line of output:

`present sg 1` (MPS of query)
`eat [strong]` (stem and inflection class)
`I: ditto` (result of block I)
`tense: ditto` (result of block tense)
`person 1: ↩ I eat` (result of block person)
`I eat` (result of sandhi)
✓(agrees with Truth)

If `notInteractive` is not set, PFME also displays a text box containing the PFM theory and a form containing the PFM theory. You may modify either the text box or the form and (re)submit it. However, not all features of PFME are available via the form, and neither the form nor the text box is a faithful reproduction of the PFM theory that PFME has analyzed. These features are only to help you get started.

References

[1] G.T. Stump. *Inflectional morphology: a theory of paradigm structure*, volume 93. Cambridge University Press, 2001.