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

July 4, 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 version 2, which is accessible at `http://www.cs.uky.edu/~raphael/linguistics/pfm2.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 the morphological structure of a language in two ways.

- Uploading from a file
- Entering into a text area (preferably by copy and paste)

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)[1]. 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}
```

[1] Instead of Unicode \u2192, one may use either Unicode \u279d or the two-character combination -> .

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

3.2 Bracketed expandables

A **bracketed expandable** is an expandable surrounded by brackets, as in {`1/2 fem pl`} or `[noun aStem]`. The first is an example of a {-bracketed expandable, and the second is a `[`-bracketed expandable.

PFME uses {-bracketed expandables to refer to MPSs that are supersets of any member of the expanded list. So {`1/2 fem pl`} matches any MPS that includes, for instance, {`1 fem pl`}. PFME uses `[`-bracketed expandables to refer to combinations of syntactic category and inflection class. So `[noun aStem]` matches any noun that is in inflection class `aStem`.

3.3 Named expandables

A **named expandable** is formed by a name, a colon, and a {-bracketed expandable, such as `S:`{`1 sg`} or σ`:`{`dat pl`}. The name can be any word, but `S` and σ are the most common names.

3.4 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`:`{`1 pl`}`>`, where `L` represents the lexeme and σ`:`{`1 pl`}`>` represents the MPS. The MPS may be a name, a {-bracketed expandable, or a named expandable.

3.5 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.

3.6 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 are patterns that match the component specified by the context.
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
```

Each of these represents a pattern to be matched against the MPS component of the context. Alternative `a` is unavailable because it requires the component `fut`, which is not present in the MPS. On the other hand, `b`, `c`, and `d` are available. 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 then newline), or in the Macintosh fashion (carriage return).

4.1 Identification

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

```
Language:  Name
```

. The language name contains everything on the line following `Language:`.
It should also specify the author of the theory:

```
Author:  Name
```

. The author name contains everything on the line following `Author:`.

4.2 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 may be multiple words.

4.3 Root-selection rules

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

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

This example demonstrates a variety of acceptable formats. The first two lines show the most general format, where the left-hand side is in full context notation. The other examples use various acceptable shorthands. Root formats obey these rules:

- The MPS, if present, may be either a {-bracketed expandable or a named expandable. The name, if present, is immaterial.
- The lexeme component on the right-hand side must be a single word.

PFME uses Paṇīni precedence to select the stem whose MPS pattern is the best match to the MPS in the context.

4.4 Stems

If all stems are simply roots, one may omit any direct mention of stems. For backward compatibility with PFM1, one may use the `Root` syntax above, replacing the word `Root` with `Stem`.

In some languages, however, stems are formed from roots by morphophonological operations. An example comes from Hua (dialect of Yagaria, Trans-New Guinea):

```
Stem(L:front) = front(Root(L))
Stem(L:back) = back(Root(L))
Stem(L:diag) = low(Root(L))
Morphophonological operations = {
  front(Pu) = Pi
  front(Po) = Pe
  low(Po) = Pa
  low(Pu) = Pa
  back(Pi) = Pu
  back(Pe) = Po
}
```

In this example, `front`, `back`, and `diag` are **lexeme modifiers**. They are also names of morphophonological operations. The definitions of the operations use `P` to represent arbitrary phonemes. The two rules for `front` say that a root ending with `u` should have that ending changed to `i`, whereas a root ending with `o` should have that ending changed to `e`. A root that satisfies neither of these situations remains unchanged.

Lexeme modifiers may be any word; likewise, the name of morphophonological operations may be any word.

4.5 Content paradigm

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

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

For backward compatibility with PFM1, one may use the word `ParadigmSchema` instead of `Content paradigm schema`.

The first line may have non-empty `(`-bracketed expandable pattern (here, `(V)`) that matches syntactic categories and inflection classes. The rest of the schema is a {-bracketed expandable that generates a list of MPSs.

Given a lexeme, PFME finds all content-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):

```
Content paradigm schema(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`...`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`...`6`.

4.6 Disallowed MPSs

The MPS list that PFME produces from the content-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 Converting from content to form paradigm

In most theories, the content paradigm is the same as the form paradigm. In these cases, there is no need to specify a correspondence. When the form paradigm differs from the content paradigm, we express their correspondence by a `Corr` function. Here is an example from Noon (Niger-Congo; Senegal).

```
Corr(<L[like], σ>) = <Stem(L), objPos(σ)>
Corr(<L[balaa], σ>) = <Stem(L), objRel(σ)>
Corr(L) = <Stem(L), σ> % default rule; unnecessary
Property mapping objPos = {
   (INFL:{obj}) → (INFL:{poss})
}
Property mapping objRel = {
   (INFL:{obj}) → (INFL:{rel})
}
```

The left-hand side of each `Corr` rule specifies a context, including (optionally, in brackets) a pattern to match the syntactic category and inflection class and (optionally) an MPS, either {-bracketed or named, to match the content paradigm. There are two acceptable right-hand side forms:

1. a context specifying both the stem and the form paradigm. If the left-hand side is named, the same name must appear on the right-hand side; if the MPS is not named, it is taken to be σ. This form paradigm may indicate a modification of the MPS by naming a property mapping. Each property mapping must be defined with one or more rules. Each rule has a left-hand side pattern to match the MPS and a right-hand side showing how that part of the MPS is to change. The two sides are separated by →. The best property mapping is chosen based on Paṇīni precedence of matches with the MPS. If no property mapping applies, the mapping is the identity function.

2. a referral to another `Corr` rule, such as `Corr(<L, pm(σ)>))`, which passes the lexeme (and its syntactic category and inflection class) along with an MPS modified by a property mapping.

Given a lexeme and a content paradigm, Paṇīni precedence determines the best `Corr` rule, which then computes the form paradigm. If no `Corr` rule applies, the form paradigm is the same as the content paradigm.

4.8 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(<X,σ>) = [person:  [tense:  [I: <X,σ>]]]]
```

The line saying `Paradigm function` is optional; the line with `PF` is required.

By convention, `<X,σ>` refers to the context formed by the stem (which may be a modification of the root, as shown earlier) and the form paradigm (which may be a modification of the content paradigm, as shown earlier).

This particular function says that the way to generate a surface form from the context `<X, σ>` is to apply rules of exponence (described below), first choosing 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`). The paradigm function must include at least one block of rules of exponence.

For backward compatibility with PFME version 1, one may also write a paradigm function using this syntax:

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

This syntax explicitly refers to the lexeme and its stem.

4.9 Rules of exponence

Rules of exponence are organized in named blocks. 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],{perfect/past/futPerf} → Xed
```

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

```
blockName, X[],{} → 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 input 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 classifier, the classifier is taken to be empty.

The left-hand side concludes with the **MPS component**, which is either a {-bracketed expandable or a named expandable. The name, if present, is immaterial. The components of supertokens in the MPS component may be abbreviated by a single Greek letter. For instance, the MPS component may look like this:

```
{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 arbitrary characters and may contain special forms:

- The letter `X`, standing for the input to the rule.

- An embedded expression, such as `[Negator:[Mood:<X,S>]]`, which refers to a subordinate paradigm function, in this case, invoking first the block `Mood` and then the block `Negator`. For backward compatibility with PFME version 1, embedded expressions may have the form `(Negator(Mood(<X,S>)))`, which must be surrounded by parentheses.

- Embedded expressions must refer to `X` by a full context, such as `<X,S>`. The context may be modified from the input context, and it 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.

- A reference to a stem based on an updated context:

  ```
  [Stem:  <X,S:{prefixed}>]
  ```

  For backward compatibility with PFME version 1, you may also write such a reference this way:

  ```
  (Stem(<X,S:{prefixed}>))
  ```

  Such references must be parenthesized as shown. In this example, the MPS of the context is enhanced by adding the element `prefixed`.

- A reference to a morphophonological operation, such as `(!back(X))`.

Parentheses, brackets, 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 braced 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. Whenever a rule applies, however, all the rules are tried again, starting from the first sandhi rule.

6 Truth

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

```
Truth = {
  PF(<EAT, σ:{1 sg present}>) = I eat
  PF(<EAT, σ:{3 sg present}>) = he eats
  PF(<EAT, σ:{3 sg past}>) = he ate
  PF(<EAT, σ:{3 sg perfect}>) = he has eaten
  PF(<EAT, σ:{1 pl perfect}>) = we have eaten
  PF(<EAT, σ:{3 sg futPerf}>) = he will have eaten
  PF(<CLIMB, σ:{1 sg past}>) = I climbed
```

```
    PF(<CLIMB, σ:{3 pl perfect}>) = they have climbed
}
```

Each entry must be on a single line by itself. The MPS name should be σ or `S`. The MPS itself is a {-bracketed expandable or a named expandable. For backward compatibility with PFME version 1, one may also express a known form in this syntax:

```
    CLIMB:{3 pl perfect} = they have climbed
```

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 in which the user can enter or modify the PFM theory.

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 content-paradigm 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**. It then finds the appropriate root for the lexeme, from which it builds the stem. It applies the most applicable `Corr` rule to convert the content paradigm in the query to a form paradigm. 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:` $\hookleftarrow$ `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 in which you may paste a new theory and submit it.

References

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