Functional Pronunciation Dictionaries

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

September 2, 2007

1 Introduction

This document describes the system **Functional Pronunciation Dictionaries** (FPD), a language-independent system for defining pronunciation lexicons. The starting point of this system is Functional Morphology (FM) [3, 2], where we asked us the question — given that we already have defined a lexical resource in FM, what would be an efficient approach to extending the lexicon with high-quality pronunciation information?

We will use Swedish to illustrate the ideas of FPD.

The main idea of FPD is based on the assumption that we can get high correctness by using automatic transcription, and if a transcription is erroneous, it is normally not completely wrong — it may be one or two sounds that are incorrect. Given that this assumption is true, then automatic transcription is a reasonable approach as the first step in the creation of a high-quality pronunciation dictionary. The lexicographer's job would be to adjust the incorrect transcriptions.

M. Uneson [8] reports a precision of 95.7% in his automatic transcription system of Swedish, i.e. approximately every twentieth word is transcribed wrongly. Uneson gives a list of the most problematic cases: compounds; lexical stress; lexical word accent; loan words; pronunciation that is not derivable by orthography; occasional exceptions to normal orthographical markings. Moreover, he reports that the main problem is compound resolution (the compound boundaries are unmarked in Swedish). The compound resolution is, of course, connected to the problem of assigning the lexical stress.

Learning from Uneson's experiences, we only allow non-compound in our lexicon and the pronunciation of compounds are instead resolved in FM:s compound analysis.

Even though we are describing how the automatic transcription can be done by manual rules within FPD, we could just as well plug in an external transcription system, such as Uneson's transcription system.

2 Representation of pronunciation in FPD

The transcription of pronunciation has been standardized by International Phonetic Alphabet [7, 1], abbreviated IPA. Before IPA was invented, all phoneticians invented their own description alphabet, and comparisons between works were difficult, sometimes close to impossible.

A transcription can be accomplished on different levels of detail. The transcription with least details are referred to as *phonemic*, indicated with slants (/../). *Phonetic* transcription, indicated with brackets ([..]), are divided into two types, *narrow* and *broad*, where broad, in contrast to narrow, contains the correct phones but few diacritics.

In ordinary dictionaries, it is common to use a phonemic transcription with symbols for vowel length (ː), primary stress (ˈ) and secondary stress (ˌ), and diacritics for syllabic consonants, i.e. a consonant that form a syllable by itself or acts as a nucleus in a syllable.

2.1 Algebraic Representation of IPA

We use an abstract representation of IPA, instead of using some concrete encoding scheme. The IPA alphabet is represented with an algebraic data type in the programming language Haskell [6], the description language of FM. The motivation for an abstract representation of the IPA alphabet, instead of using a particular encoding (see Sec. 6 about encodings), is because most encodings do not cover the whole IPA. Moreover, it is easier to generate an encoding from an abstract representation than from another encoding.

The `IPA` type contains five constructors, corresponding to consonants and vowels, with optional diacritics, primary and secondary stress, and unknowns. Unknowns are used to mark letter for which no analysis was possible. The reason for this could be that no mapping is yet defined, or that the input letter is not in the alphabet of the target language. The definition of consonants and vowels are faithful reflections of the IPA tables: consonants are defined in terms of manner, place, voice, and quantity; and vowels in terms of orientation, openness, roundness, and quantity.

```
data IPA = C Consonant [Diacritics]    |
           V Vowel     [Diacritics]    |
           PrimaryStress               |
           SecondaryStress             |
           Unknown

data Consonant = CC (Manner,Place,Voice,Quantity) |
                 HookTopHeng

data Vowel = VC (Orientation,Openness,Roundness,Quantity)

data Diacritics = Voiceless | Voiced | ...
```

The normal use is to define a set of names for a subset of IPA, representing the sound set of the target language, here with four examples from Swedish: the primary stress marker ˈ, the consonant b, the vowel ɑː, and the retroflexed consonant ɖ.

```
stress :: IPA
stress  = PrimaryStress

b :: IPA
b     = consonant (Plosive,Bilabial,Voiced,Short)  []

aa :: IPA
aa = vowel (Front,Open,UnRounded,Long) []

rd :: IPA
rd = consonant (Plosive,Retroflex,Voiced,Short) []
```

Transcription of the Swedish word *bard* (Eng. 'bard') would be the Haskell list `[stress,b,aa,rd]` corresponding to [ˈbɑːɖ].

3 Automatic transcription of Swedish in FPD

We will now present an example on how an automatic transcription of Swedish may look like FPD. The goal is to illustrate the method, not to present the best transcription strategy.

Our starting point is an input non-compound word that has its stem marked. The first step is stress marking, where special symbols are inserted, representing primary and secondary stress. This step is much simpler than assigning stress to an arbitrary Swedish word, since we know where the stem is, and moreover, that it is not an compound.

Letter	Pronunciation	Letter	Pronunciation
a	ɑ	*b*	b
c	s	*d*	d
e	e	*f*	f
g	g	*h*	h
i	i	*j*	j
k	k	*l*	l
m	m	*n*	n
o	u	*p*	p
q	k~~ʉ~~	*r*	r
s	s	*t*	t
u	~~ʉ~~	*v*	v
x	ks	*y*	y
z	s	*å*	o
ä	ɛ	*ö*	ø

Figure 1: Letter-to-pronunciation table

The next step is a naive letter-to-pronunciation translation, i.e. a mapping of every Swedish letter to a pronunciation, illustrated in Fig. 3.

The letter-to-pronunciation translation is done by a function `letter_to_ipa`, easily derivable from the table in Fig. 3. The result of the function is a list of `InterIPA`. An `InterIPA` is a pair of a `String`, which is the sequence of letters corresponding to the pronunciation, and an `IPA`, the algebraic representation of the pronunciation.

```
type InterIPA = (String,IPA)

letter_to_ipa :: String -> [InterIPA]
```

To improve the result, stepwise refinements are applied to the basic transcription. In Sec. 3.1 to Sec. 3.8, we present a couple of rules, ordered as they are applied. The rules have been derived from the description of Swedish phonology in Garlén's book [5].

A transcription system external to FPD could, as well, be plugged into `letter_to_ipa`. The work required is the translation of the output of the system into a list of `InterIPA`.

3.1 Detection of ʃ

The rule maps every substring of the form *sj* and *stj* to ʃ. Note that there are many other substring that is also mappable to ʃ, e.g. *sk*, *sch* and *sh*.

Here we use a helper function `replace_refine`, which perform the actual mapping. We use functional composition (`.`) to sequentialize the replacement. Functional composition starts with the last element, i.e. the refinement of *sj*, and goes backwards.

```
sje_refine :: [InterIPA] -> [InterIPA]
sje_refine = replace_refine ("stj",Sw.sj) .
             replace_refine ("sj",Sw.sj)
```

3.2 Detection of ç

The rule maps every substring of the form *tj* and *ch* to ç. The rule is defined in the same manner as `sje_refine`.

3.3 Detection of ə

The rule maps every occurrence of *e* to ə, given the following conditions: it occurs in a non-stressed final position, or it occurs in a non-stressed syllable preceding either *l* or *n*.

```
schwa_refine :: [InterIPA] -> [InterIPA]
schwa_refine is = e_refine is False
  where
    e_refine          [] _    = []
    e_refine (c@("[",_):xs) _ = c : e_refine xs True
    e_refine (c@("]",_):xs) _ = c : e_refine xs True
    e_refine (c@("e",_):xs) stressed =
        case xs of
 []        | not stressed -> [("e",Sw.schwa)]
 [("l",_)] | not stressed -> (("e",Sw.schwa):xs)
 [("n",_)] | not stressed -> (("e",Sw.schwa):xs)
 _                        -> c:e_refine xs False
    e_refine (x:xs) stressed
     | stressed && sw_vowel x = x:e_refine xs False
     | otherwise              = x:e_refine xs stressed
```

3.4 Refinement of g

The rule maps every *g* to the sound j if it occurs in a stressed position before a soft vowel (e, i, y, ɛ, ø).

```
g_to_j_refine :: [InterIPA] -> [InterIPA]
g_to_j_refine is = j_refine is False
  where
    j_refine             [] _ = []
    j_refine (c@("[",_):xs) _ = c :j_refine xs True
    j_refine (c@("]",_):xs) _ = c :j_refine xs True
    j_refine (c@("g",_):c1:xs) stressed
| stressed && soft_wovel c1  = ("g",Sw.j):c1:xs
| otherwise  = c:j_refine (c1:xs) stressed
    j_refine (x:xs) stressed
     | stressed && sw_vowel x = x:j_refine xs False
     | otherwise              = x:j_refine xs stressed
```

3.5 Refinement of ä and ö

The rule maps every occurrence of *ä* and *ö* preceding *r* to the sounds æ and œ respectively, i.e. the vowels becomes more open in front of an *r*.

```
öä_refine :: [InterIPA] -> [InterIPA]
öä_refine [] = []
öä_refine (("ä",_):c@("r",_):xs) = ("ä",Sw.ä2):c:öä_refine xs
öä_refine (("ö",_):c@("r",_):xs) = ("ö",Sw.ö2):c:öä_refine xs
öä_refine (x:xs) = x:öä_refine xs
```

3.6 Double consonants

The rule locates and merges doubled consonant (and the letter combination *ck*) to a single long consonant. The implementation is a simple traversal of the input, where two helper functions are used: `is_consonant`, which identifies a consonant, and `change_quantity`, which is used to mark consonants as long.

```
double_consonant_refine :: [InterIPA] -> [InterIPA]
double_consonant_refine  [] = []
double_consonant_refine [c] = [c]
double_consonant_refine (("c",_):("k",_):xs) =
  ("ck",Sw.kk):double_consonant_refine xs
double_consonant_refine (c1@(s1,cons1):c2@(s2,cons2):xs)
 | is_consonant cons1 && cons1 == cons2  =
    (s1++s2,change_quantity Long cons1) : double_consonant_refine xs
 | otherwise    = c1 : double_consonant_refine (c2:xs)
```

Letter combination	Pronunciation
rl	ɭ
rn	ɳ
rt	ʈ
rd	ɖ
rs	ʂ
ng	ŋ
gn	ŋ
lj	j
hj	j
gj	j

Figure 2: Assimilation table

3.7 Assimilation

The rule performs assimilation, i.e. two sounds are melted into one. The assimilation table is given in Fig. 3.7. Note that there are exceptions to this transformation where no assimilation occurs, e.g. *sälja* (Eng. 'to sell') and *tälja* (Eng. 'to carve'). The function `assimilation_refine` dealing with the assimilation is defined in the same manner as `sje_refine`.

3.8 Vowel quantity

The rule decides on the vowel quantity, where the default is a short vowel. A vowel becomes long if it is stressed and it is followed by a short consonant, or if it is stressed in a final position. Note that this rule requires that we have performed the double consonants refinement. The function `vowel_quantity_refine` may look a bit complicated, but what it does is essentially just lexical stress book keeping.

```
vowel_quantity_refine :: [InterIPA] -> [InterIPA]
vowel_quantity_refine is = v_refine is False
 where
  v_refine []              _ = []
  v_refine (c@("[",_):xs) _ = c : v_refine xs True
  v_refine (c@("]",_):xs) _ = c : v_refine xs True
  v_refine ([c1@(s1,v1)]) stressed
      | stressed && is_vowel v1 = [(s1,toLongVowel s1 v1)]
      | otherwise               = [c1]
  v_refine (c1@(s1,v1):c2@(_,cons1):xs) stressed
      | stressed && is_vowel v1 &&
```

```
      is_consonant cons1 && is_short cons1 =
  (s1,toLongVowel s1 v1) : c2 : v_refine xs False
      | is_vowel v1 = c1:v_refine (c2:xs) False
      | otherwise   = c1:v_refine (c2:xs) stressed
  v_refine (c1:xs) stressed = c1:v_refine xs stressed
```

4 Paradigm description

The paradigm description of FPD is essentially the same as FM, with the crucial difference that the stem is marked. However, this additional annotation does not require any substantial work. Let us see how we could rewrite the third declension function `decl3`, given below.

```
decl3 :: String -> Noun
decl3 sak = mkNoun sak (sak ++ "en") (sak ++ "er") (sak++"erna")
```

It is the variable `sak` that we want to be marked with stem annotations. An easy way to do the marking, as done in `decl3_marked`, is to change the input variable name to `sak'`, and define `sak`, as `sak'` with stem annotations, done with the helper function `stem`. It does not really matter how `stem` is defined, as long as we are consistent, here we use curly braces for the stem marking.

```
decl3_marked :: String -> Noun
decl3_marked sak' = mkNoun sak (sak ++ "en") (sak ++ "er") (sak++"erna")
  where sak = stem sak'

stem :: String -> String
stem s = "{" ++ s ++ "}"
```

5 A new dictionary language

The pronunciation adjustments are done in the lexicon of FPD. The words that obtain the incorrect transcription are marked with a set of terms that describes how the pronunciation should be adjusted. It is up to the developer to define a term language parser of the terms describing all possible transformations.

⟨*Dict* ⟩ ::= ⟨*ListEntry* ⟩
⟨*Entry* ⟩ ::= ⟨*Ident* ⟩ ⟨*ListString* ⟩ { ⟨*ListTerm* ⟩ }
⟨*Term* ⟩ ::= ⟨*Ident* ⟩ (⟨*ListTerm* ⟩)
| ⟨*Ident* ⟩
| ⟨*String* ⟩
| ⟨*Integer* ⟩
⟨*ListEntry* ⟩ ::= ϵ
| ⟨*Entry* ⟩ ; ⟨*ListEntry* ⟩
⟨*ListString* ⟩ ::= ⟨*String* ⟩
| ⟨*String* ⟩ ⟨*ListString* ⟩
⟨*ListTerm* ⟩ ::= ⟨*Term* ⟩
| ⟨*Term* ⟩ , ⟨*ListTerm* ⟩

Figure 3: Dictionary language

An example on how the dictionary may look like is given here. The first word, *sol* (Eng. 'sun'), is transcribed correctly, but the second word, *kol* (Eng. 'coal'), is not. The term in the curly brackets describes that o in the stem should be map to u.

```
d2 "sol" ;
d2 "kol" {aa_to_o(stem)} ;
```

This new language of lexicons is given in Fig. 5 in BNF notation. A term is arbitrarily complex and may be built up from other terms, atoms, integers and strings.

6 Encodings

There are many different encodings of the IPA table — some are for displaying transcriptions, such as *TIPA* [4], which is a package for LaTeX, supporting IPA fonts, and *HTML Unicode*. Moreover, some encodings are in ASCII, created to simplify processing and editing on a computer, such as: *ARPABET 1 & 2*, developed by ARPA, and *SAMPA*, a language dependent[1] encoding, and *X-SAMPA* [9], a language independent encoding.

Adding a new encoding to FPD involves describing a mapping of the `IPA` type to the target encoding. This mapping may differ a bit, depending on how the encoding symbols are built up, but it is typically straight-forward.

[1] A language with SAMPA encoding has its own set of SAMPA characters that cover the sounds of that language.

Here is an example of the translation of some consonants to TIPA. The quantity part, which actually just adds a colon (`:`) to the end of the character, is abstracted with the function `prQuantity`.

```
prConsonant :: Consonant -> String
prConsonant HookTopHeng = "\\texththeng"
prConsonant (C (manner,place,voice,quantity)) =
    prQuantity quantity $
      case (manner,place,voice) of
       (Plosive,Bilabial,Unvoiced)       -> "p"
       (Plosive,Bilabial,Voiced)         -> "b"
       (Plosive,Alveolar,Unvoiced)       -> "t"
       (Plosive,Alveolar,Voiced)         -> "d"
       (Plosive,Retroflex,Unvoiced)      -> "\\textrtailt"
       (Plosive,Retroflex,Voiced)        -> "\\textrtaild"
       ...
```

The encoding is not required to be complete, we just define those sounds covered by the encoding, and maps all else to a dummy symbol. As long as the transcription is within the encoding, everything is ok, and if it is outside it, then the user will be notified by the occurring dummy symbol.

7 Dealing with Compounding

As mentioned previously, the pronunciation of compounds is done in the compound analysis of FPD, i.e. if an input word is analyzed as a compound, we adjust the lexical stress markers to match the pronunciation of Swedish compounds. The rule of lexical stress of Swedish compounds is: *the first word of a Swedish compound word has a primary accent and the following words has a secondary stress.*

This rule is not without exceptions, e.g. the word *Kolsvart* (Eng. 'pitch black') has two primary accents: ['koːl 'svaʈ]. However, these exceptions are so few that they can be treated with an exception list.

8 Changes in the FM implementation

The changes necessary are rather conservative:

- Extend the class `Frontend` with five member functions: `word_annotations`, which removes all special symbols occurring in word forms; `preprocess`, which adds symbols for stress and similar markings; `transcription`,

which performs the automatic transcription; `term_parser`, which parses the terms and performs the operations derived from them;
and `compound_pronunciation`, dealing with the pronunciation of compounds. All these functions may be defined as identity functions as default, i.e. they are conservative additions to FM.

- Change the lexicon format — this would be a good thing for FM, since the current lexicon format is rather primitive. The term language should by no means be restricted for use in the transcription process, instead it should be a general facility to add and adjust information.

- Extend the command language with flags for controlling the encodings of the output transcription.

9 Final comments

This system aims at providing an answer to the question: given that we have an automatic transcription system, then how do we systematically deal with the exceptions without having to resort to an exceptions list where the whole words are transcribed?

The automatic transcription described here is a part of a CGI application[2], and not yet fully implemented in FM. The details on the additions necessary are presented in Sec. 8.

The major step is to perform an evaluation of Swedish transcription to demonstrate the feasibility of the approach. For this we need a reference material of good quality, but to my knowledge, there is no such source publicly available[3].

References

[1] I. P. Association, editor. *Handbook of the International Phonetic Association*. Cambridge University Press, Cambridge, UK, 1999.

[2] M. Forsberg and A. Ranta. Functional Morphology. *Proceedings of the Ninth ACM SIGPLAN International Conference of Functional Programming, Snowbird, Utah*, pages 213–223, 2004.

[2] Available at: `http://www.cs.chalmers.se~/markus/transcription`

[3] The reference material used by M. Uneson, Hedelin's pronunciation lexicon, is not publicly available.

[3] M. Forsberg and A. Ranta. Functional morphology. `http://www.cs.chalmers.se/~markus/FM`, 2007.

[4] R. Fuki. TIPA Manual Version 1.3. *Graduate School of Humanities and Sociology, The University of Tokyo*, 2004.

[5] C. Garlén. *Svenskans Fonologi*. Studentlitteratur, Lund, Sweden, 1988.

[6] S. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available at `http://www.haskell.org`, February 1999.

[7] The International Phonetic Association, 2005. `http://www.arts.gla.ac.uk/IPA`.

[8] M. Uneson. Letter-To-Sound Conversion for Swedish with FST and TBL. *Term Paper: Statistical Methods, GSLT, Sweden*, 2005.

[9] J. Wells. Computer-coding the IPA: a proposed extension of SAMPA, 1995.