Text to speech — a rewriting system approach

José João Almeida
jj@di.uminho.pt

Alberto Manuel Simões
albie@alfarrabio.di.uminho.pt

Abstract In this document we present an open source text-to-speech for Portuguese. Our first goal is to provide a flexible way to extend it, using a generic way to convert Portuguese words into SAMPA phonemes, and consult dictionaries only on exceptions.

The Text-to-Speech is composed of five layers, each one based on simple rules in a way that can be easily tuned. In order to do that, we wrote a generic text rewriting system that is presented the section 2.

The result of this work is a tool that can be used as an independent Text-to-Speech system or as a Natural Language Processing library for various tasks. We present some examples on how they can be used in the *Applications* section.

1 Introduction

Text-to-Speech(TTS) is, as we know, a difficult area. Romance languages, like Portuguese, are very hard to transform into sound because of the great amount of exceptions.

We intended to make the most generic Perl[8, 4] module to convert Portuguese texts into sound, using rules to transform words into SAMPA [7] phonemes and dictionaries for exceptions.

Our approach is based on rewriting rules. We take a text, divide it into sentences and, based on the punctuation, classify the sentence as exclamative, interrogative or other. This classification will be used later, by the prosodic transformer in order to make sentences more understandable.

Each sentence is divided into words to be, each of them, transformed into SAMPA. This process is based on dictionary search and on rule transformations. Later, the words are again joined and compared with rules to make better word junctions.

The SAMPA sentence formed thus is passed to a prosodic transformer to make the phrase sound more human (transform a same frequency sound into a melodic one). This is done with rewriting rules, too.

Scheme 1 tries to explain this cycle.

All these rewriting systems and functions can be used as a complete program, or can be called independently. In the later case, we are talking about a Perl module or library.

This Perl module has many functions that can be helpful whithout the full TTS system. We can name some of them, like the text to sampa conversor, the word to sampa (different from the previous one in the data type), or text to MBrola[6] system file format.

Other ones, not so connected with the TTS system, can be helpful for other purposes, like the number to text conversion, or the e-mail and internet URLs conversor to text.

Looking to this module as an application, we can get a program to read text from the standard input, or to create a wave file to play later.

This system was not built alone, but in conjuntion with a `pt::pln` perl module (Portuguese natural language processing module) that implements some basic functions like sentence tokenize to words, words division by syllable, tonic syllable search and so on.

2 Rewriting system

To this and other purposes, we built a rewriting system. What we intend about this is a system that, given a set of rules, parses a text and rewrites all matching patterns.

Each set of rules, after the compilation takes place, generates a function that accepts the text to be rewritten, and returns it. These

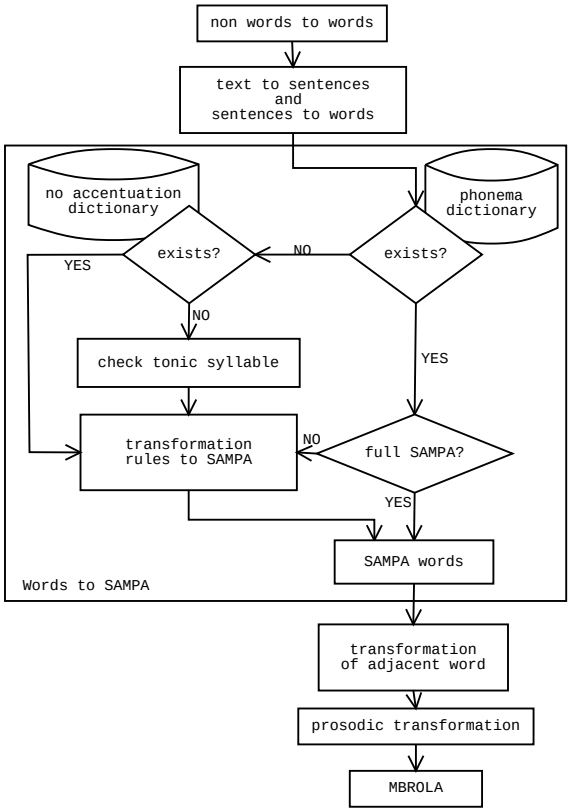

Figure 1: pt::speaker structure

functions can easily be composed so that we will have composed rewriting systems.

There are various kinds of rules, from simple substitutions rules, rules that evaluate the string they will replace to rules that are evaluated only when starting the system, rules that evaluate if there is some context conditions and rules that make the system quit.

If there is any of the rules that make the system exit, the system will process the text until no rule pattern matches.

In this particular system, we should define rules on a file, and compile it using the `mktextrr` command. This transforms rules to a Perl script, that does the real job.

The source file is a perl file that, between `RULES` and `ENDRULES` string, accepts rule definitions to construct a function.

Rules have the following syntax:

```
left hand side ==> right hand side
left hand side =e=> right hand side
left hand side ==> right hand  !! condiction
```

Because of the column width of the article, we will use a style similar to LaTeX..

A first example: a rewriting system that expands an e-mail to a HTML link to that e-mail:

```
RULES email_expand
(\w+(\.\w+)+\@\w+(\.\w+)+)
⇓
<a href="mailto:$1">$1</a>
```

Note that regular expression matching is pure perl code.

Saving this file under the name `email_expand` and processing it with

`mktextrr email_expand email_expand.pl`,

we get a function, named `email_expand` that accepts a text and does the transformations needed.

You will notice that we didn't use the `ENDRULES` command. This is because there is nothing more after the rules set. Now, suppose we will edit the file, and make another function, this one expand `http URL`s:

```
RULES email_expand
(\w+(\.\w+)+\@\w+(\.\w+)+)
⇓
<a href="mailto:$1">$1</a>


RULES http_expand
(http://\w+(\.\w+)+)
⇓
<a href="$1">$1</a>
```

Once more, we didn't use the `ENDRULES`, because we are defining another rule set. Compiling the rules set, and using `http_expand(email_expand($text))` we can replace all emails and URL's.

Because `RULES` define a set, we can replace the example with the following code:

```
RULES expand
(\w+(\.\w+)+\@\w+(\.\w+)+)
⇓
<a href="mailto:$1">$1</a>


(http://\w+(\.\w+)+)
```

$$\Downarrow$$

```
<a href="$1">$1</a>
```

and, calling once this function, all emails and URLs will be replaced.

Sometimes, we want to evaluate the right side of the substitution. We can do such a thing using the rewriting system `=e=>` arrow:

```
RULES arithmetic
\s*\d+\s*[+*-/]\s*\d+\s*
```

$$\Downarrow_e$$

```
$&
```

This simple example, parses a text file and all formulae found, are evaluated. This is, indeed, a fast and easy way to change texts.

The `=b=>` arrow does not have a left hand side, and evaluates the right hand side. So, if we need a dot at the end of the string, we can make:

$$\Downarrow_b$$

```
$_.="."
```

Note that this is different from

```
$==>.
```

because there will be a endless loop. Meanwhile, it will work fine if you write:

```
[^.]$==>.
```

Finally, we can impose conditions to each rule, along with the matching pattern. For an example purpose, we have defined a user hash (`%user`) that associates user names to their full names. We can write:

```
\b(\w+)\b==>$user{$1} !! defined($user{$1})
```

This way, each word is checked, but only the ones that match with a user name will be substituted.

3 Text to words

To read a text, we must divide it into smaller tokens. These token can be, at first, sentences and, later, words.

Text can't be divided straight away into words because we need sentence delimiters to check the sentence type (interrogative, exclamative or imperative) and make prosody works.

The sentence text is, then, divided by spaces or commas into words. These words are lower cased and checked on a phoneme dictionary. This dictionary can contain full translations of Portuguese to SAMPA or semi-translated ones that will be transformed, again, with the transformation rules.

This dictionary has the following syntax:

```
dic :: line dic
line :: word '=' SAMPA
      | word '=' '!' SEMI-SAMPA
      | prefix '*' '=' SEMI-SAMPA '*'
```

We explain how this dictionary is used at the next section.

4 Words to SAMPA

To translate words to SAMPA, we need a word. Taking this word, we will check if it exists on the dictionary file:

1. If it finds a word and its full SAMPA translation (first case), its translation is returned;
2. If it exists, but it has an exclamation mark (second case), it is substituted by the SEMI-SAMPA code and the process continues;
3. If none of them exists, the last letter is substituted with an asterisk (*) and checked on the dictionary. If it does exists, the prefix text (before the asterisk) is substituted by SEMI-SAMPA and the rest of the word is concatenated and the process continues with the rewrite rules. But, if it does not exists, we take off the last letter before the asterisk, and check it again, until the word disappear.

   These asterisks are used to signal that there is an exception for words starting that way. So, with only one of these rules we can process a lot of words (verbal constructs, and so on).
4. If there isn't a prefix for the word, the process continues to the rewrite rules.

This rewriting system uses two functions. The first one, tries to convert simple letters sequences to it's respective phonemes

(SAMPA and some pseudo-SAMPA ones) and the second one, tries to convert it all to SAMPA. Some pseudo-SAMPA is left so it will be possible to make some sounds take some more time.

Some letters have two or more different sounds if they appear between specific letters. So, we have rules like

```
($vg)x($vg)==>$1z$2
```

where the `$vg` variable contains all the letters and SAMPA phonemes that should be considered vowels.

There are other cases where some letters appearing in the beginning of words should be read differently from the cases where it appears in the middle of some others.

These rewriting rules also try to find the tonic syllable. This is done checking if certain sequences of letters are found at the end of the word.

5 *Transformation of adjacent words*

When reading, people tend to join some letters. As in English we can write "aren't" instead of "are not", Portuguese speakers[1] do several oral contractions.

For this purpose, we decided to make a new set of rules to rewrite sentences joining some word vowels, like

este elefante ⟶ est'elefante

↓

eSt@ elefant@ ⟶ eStelefant@

or concatenating some words

és esperto ⟶ ézesperto

↓

ES 6jSpertu ⟶ Ez6jSpertu

These rules have a slash delimiting words, so the two examples showed before, would be:

```
@/([ea])==>$1
S/6==>z/e
```

[1]specially speakers of Northern dialects

We should make it clear that these rules should join SAMPA words, and not Portuguese words. This is the reason we use an uppercase `S` on the second rule, instead of a lowercase one.

6 *Prosodic transformer*

This is another rewriting system. This is, probably, the most complicated one.

First, we define a set of letters and its respective duration. Then we match the tonic syllable, to make sound frequency go up or down some time later. For this, we define a set of commands, like `=Sub`, `=Sup` and `=Pause` to make the frequency go down, go up or pause the sound for some time.

Here are some examples of frequency variations for interrogative sentences:

```
($vg):\? ==>$1=Sub=Sup=Pausa500
($vg):($vg)\?==>$1=Sub $2=Sup=Pausa500
($vg):($con)($vg)\?==>$1=Sub $2 $3=Sup=Pausa500
```

Note that these colons symbolize the first vowel from the tonic syllable. Thus, there are some rules that make sound frequency go up, at the colon, and go down slighty, letter by letter, to the end of the word:

```
($vg): ==>$1=Acen
($vg)=Acen ==> \n$1-dur=$durac{$1}-30-130
```

At the end, we replace these commands with their respective frequency numbers and send them to the MBROLA[6] phoneme file for later conversion to wave and playback.

The monochordic version of this system was hard to understand. After applying a noisy transformation, making sound vary randomly, it was better understandable. Finally, using our simple prosodic transformer, we can understand even the differences between interrogative and imperative sentences. Of course, this system is very incomplete, but futher alterations will make it work better.

7 *Non words to words*

Before tokenizing text to words, we thought it would be useful to translate numbers, emails and URLs to a readable form. This is an addiction to the basic Text-to-Speech system that make it more sophisticated for real use.

7.1 Numbers to Words

The first one, for numbers, takes a number, decomposes it into the various components (unities, decimal, and so on) and translates each of them to the corresponding text form. This example will show only a small piece of the rewritng system because it is very long:

```
RULES number
10==>dez
11==>onze
12==>doze
 [...]
18==>dezoito
19==>dezanove
20==>vinte
2(\d)==>vinte e $1
30==>trinta
3(\d)==>trinta e $1
 [...]
70==>setenta
7(\d)==>setenta e $1
80==>oitenta
8(\d)==>oitenta e $1
90==>noventa
9(\d)==>noventa e $1
1==>um
2==>dois
 [...]
8==>oito
9==>nove
0$==>zero
```

The complete set of rules to translate numbers from zero to 999 999 uses about of 80 lines.

7.2 URLs to Words

The second rewriting system takes emails and URL's and *textifies* them. On emails, words smaller than four letters are spelt, and others are read normally. The *at* symbol is read, as well as the dots. Meanwhile, we put some pauses after each dot to make it more understandable.

This example is a bit more complicated. Because we want to translate the email to words, we will, probably, have endless loops. We want words of three or less characters to be spelt and bigger words read normally.

For this, we defined a associative array (we could make another rewriting system for this) that associates each letter to it's pronunciation:

```
%letters = {
            'a' => 'á',
            'b' => 'bê',
            'c' => 'cê',
            ...
            'z' => 'zê' }
```

If you make a system like,

```
[a-zA-Z]{1,3}?\b
```

$$\Downarrow_e$$

```
join("",map {$letters{lc($_)}} split(//,$1))
```

after substituting letters for their pronunciation, they will be replaced again and again, forever.

The solution we encountered to make this work, was to place a token that will go from the beginning to the end of the string. The result will be:

```
=b=> $_ = "_$_"
_\. ==> , ponto _
_\@ ==> , arroba _
_([a-zA-Z]{1,3}?)\b =e=>
    join("", map {$letters{$lc($_)}}
                  split (//,$1))."_"
_(.+?)\b ==> $1 _
_$==>
```

Explaining these rules:

- Place the token underscore (_) at the beginning of the text (this token is not the best one, because e-mails can contain them, but it makes easier to explain);
- Replace underscores followed by a dot by the word *ponto* followed by the underscore;
- Make the same thing with the *at* symbol;
- Words with one to three letters preceded by underscore, are translated, each letter to its word form, joined together and an underscore placed after the expression so it won't be processed again.
- Place the token after words with more than three letters;
- If the token is at the end of the string, remove it!

For a better understanding, we present this example:

```
 1. cj@di.servidor.pt
 2. _cj@di.servidor.pt
 3. cê jota _@di.servidor.pt
 4. cê jota,arroba _di.servidor.pt
 5. cê jota,arroba dê í _.servidor.pt
 6. cê jota,arroba dê í,ponto _servidor.pt
 7. cê jota,arroba dê í,ponto servidor _.pt
 8. cê jota,arroba dê í,ponto servidor,ponto _pt
 9. cê jota,arroba dê í,ponto servidor,ponto pê tê_
10. cê jota,arroba dê í,ponto servidor,ponto pê tê
```

8 Application Examples

In this section we provide two simple examples of how to use `pt::speaker` for real applications.

8.1 Telephone Numbers

Now that we are in the era of mobile phones that recognize our voice and connects directly to the person we want, we can make the opposite thing using a simple system. Imagine a database file with nicknames, full names and the respective telephone numbers. We want that, given a nickname, the program read the full name and the telephone number.

Look to a simple database file:

```
maria:Maria Alice:999222323
manuel:Manuel João:999323222
```

The perl program, will be something like this loads the dictionary, searches the nickname we want, and reads it:

```
# Charge dictionary
open DIC, "dic";
while(<DIC>) {
 ($nick,$name,$num)=split /:/;
 $dic{$nick}=[$name,$num];
}
close DIC;

# Read a nick
$nick = <>;
if (defined($n = $dic{$nick})) {
  # build the sentence to read
  # "the telephone number of XXX is xxx
  $s = "O telefone do $name é ";
  # split the number by three digits
  # to be more easily understandable
  $n->[1]=~/^(...)(...)(...)$/;
  $s.="$1 $2 $3";
  pt::speaker::speak($s);
} else {
  # say that we didn't find it
  pt::speaker::speak("não encontrei")
}
```

8.2 HTML Table Of Contents

Let's look to yet another example of usability for this module. We have an XHTML[2] file and want to read the headings:

```
use XML::DT;
use pt::speaker;

%handler = (
  '-default' => sub{},
  'h1' => sub {
    $h2=0;
    $h1++;
    # Say that it is a chapter
    pt::speaker::speak("Capítulo $h1: $c");
  }
  'h2' => sub {
    $h2++;
    # Say that it is a section
    pt::speaker::speak("Secção $h1 ponto $h2: $c");
  }
);

dt(shift,%handler);
```

This example can be a little weird at the first look, but it's easy to understand the idea: read each heading 1 as chapters and each heading 2 as sections, number them and read the table of contents.

9 Conclusions

We can conclude that the transformation of text to sound can be done with simple substitutions, and a little of language processing techniques.

This framework can be enlarged and made more powerful. The simple act of adding a rule on any of the rewriting systems make a real difference on the sound generated. There is the possibility to make an application to check, accordingly to a phonetic dictionary, the percentage of words we match correctly. This can help any of us to add or remove rules from the rewriting system, knowing the level of changes that operation will bring.

The possibility to add some functions to translate numbers, email addresses, URLs, time of day or acronyms. It's simple to add a XML[1] parser to make various types of transformations according to the tag we are looking into. For example, we can make some tags to be spelt, other to be emphasized with higher frequency, telephone numbers to be read by two or three numbers set, and so on.

[2] not HTML so we can use a XML tool like XML::DT[2]

We can check that the main power of this framework is the rewriting system that makes almost all the text-to-speech.

Futher development may include an adaptation of the system to the Festival[3] Speech Syntethis System that have more power than the MBrola system.

The simple transformation from words to phoneme symbols can be rewriten, again, to LaTeX[5], making the habitual phonema syntax we are used to under dictionaries.

References

[1] *eXtended Markup Language (XML) version 1.0 recommendation.* World Wide Web Consortium, 10 February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.html/.

[2] Almeida, José João & Ramalho, José Carlos. Xml::dt. 1998.

[3] Black, Alan W. & Taylor, Paul & Caley, Richard. *The Festival Speech Synthesis System.* 1999. Edition 1.4.

[4] Christiansen, Tom & Torkington, Nathan. *Perl Cookbook.* O'Reilly & Associates, Inc., 1999.

[5] Goossens, Michel & Mittelbach, Frank & Samarin, Alexander. *The LaTeX Companion.* Addison-Wesley, 1999.

[6] *The MBROLA Project: Towards a Freely Available Multilingual Speech Synthesizer.* http://tcts.fpms.ac.be/synthesis/mbrola.html.

[7] *SAMPA: computer readable phonetic alphabet (ASCII codes).* http://www.phon.ucl.ac.uk/home/sampa/home.htm.

[8] Wall, Larry & Christiansen, Tom & Schuartz, Randal. *Programming Perl.* O'Reilly & Associates, Inc.