Extension: Research report on Natural Language Processor (NLP) for Khmer TTS

Abstract

This report is an extension of phase 1.2 research report on natural language processing for Khmer Text-To-Speech System. The study includes text to sound conversion, text normalization, syllabification algorithm and diphone database generation module. In text to sound conversion section, we present some statistical based text to sound conversion for Khmer language and also some information in Lexicon. In text normalization; we have identified rules to transform non standard word such as year, number, date and abbreviation to full text. The implementation to test whether these rules can cover all non standard words is also presented. For syllabification, a system created to evaluate the syllabification algorithm is presented in this section. The next section of this report will be a presentation of the diphone database module and the last one will be the general conclusion of this research activities.

I. Introduction

There are two main modules for Text-To-Speech System: Natural Language Processing (NLP) and Digital Signal Processing (DSP). The NLP module converts words in input text to their corresponding narrowed phonetic transcriptions. The DSP module takes the words in narrowed phonetic transcription and converts them into speech wave form. Before the NLP can start the study of phonetic and phonological need to be done.
The current phase (2.1) is the continuation of phase 1.2 that is about natural language processor. The activities in this phase include the study in detail of text normalization, text to sound conversion by using the statistical approach, evaluation of the syllabification algorithm and diphone database module.

II. Text to Sound Conversion

II.1. Introduction

Speech synthesis systems use two basic approaches to determine the pronunciation of a word based on its spelling, a process which is often called text-to-phoneme, text-to-sound or grapheme-to-phoneme conversion. The simplest approach to text-to-phoneme conversion is the dictionary-based approach, where a large dictionary, also called lexicon, containing all the words of a language and their correct pronunciations is stored by the program. Determining the correct pronunciation of each word is a matter of looking up each word in the dictionary and replacing the spelling with the pronunciation specified in the dictionary. The other approach is rule-based, in which pronunciation rules are applied to words to determine their pronunciations based on their spellings.

Each approach has advantages and drawbacks. The dictionary-based approach is quick and accurate, but completely fails if it is given a word which is not in its dictionary. As

dictionary size grows, so too does the memory space requirements of the synthesis system. On the other hand, the rule-based approach works on any input, but the complexity of the rules grows substantially as the system takes into account irregular spellings or pronunciations. As a result, nearly all speech synthesis systems use a combination of these approaches.
Some languages, like Spanish, have a very regular writing system, and the prediction of the pronunciation of words based on their spellings is quite successful. Speech synthesis systems for such languages often use the rule-based method extensively, resorting to dictionaries only for those few words, like foreign names and borrowings, whose pronunciations are not obvious from their spellings. On the other hand, speech synthesis systems for languages like English, which have extremely irregular spelling systems, are more likely to rely on dictionaries, and to use rule-based methods only for unusual words, or words that aren't in their dictionaries. [1]
Like English, our future Khmer Text To Speech System will rely lexicon and use rule-based methods only for unusual words, or words that are not present in the lexicon.
Here we describe the process of building letter to sound model for Khmer language from Khmer pronunciation lexicon using Festival.

II.2. Background

The Cambodian script (called *Khmer* letters) are all probably derived from various forms of the ancient Brahmi script of South India. The Cambodian script has symbols for thirty-three consonants, twenty-four dependent vowels, twelve independent vowels, and several diacritic symbols. Most consonants have reduced or modified forms, called sub-consonants or subscript, when they occur as the second member of a consonant cluster. Vowels may be written before, after, over, or under a consonant symbol. [1]
The consonants are divided into 2 groups or series: ɑ-series and ɔ-series (see fig. 1 for the group of consonants along with their pronunciation). Most of consonants (in ɑ-series resp. ɔ-series) have their counterparts (in ɔ-series resp. ɑ-series). For consonants in one group that don't have their counterpart in another group, diacritic mark ◌៉ (MUUSIKATOAN) and ◌៊ (TRIISAP) are used to produce their counterpart. ◌៉ changes the consonant to ɑ-series while ◌៊ changes the consonant to ɔ-series. Among the 24 vowels there is 1 abstract (inherent) vowel which is embedded in a consonant. The pronunciation of a vowel, including the inherent vowel, is determined by the series of the initial consonant or consonant cluster that it accompanies. Khmer syllable in written form can contain 1 or 2 vowels. In the later case, last vowel is a vowel that has embedded consonant (see fig. 2 for the list of vowels and sequences of vowels that come together along with their pronunciation for each group). Independent vowels incorporate both an initial consonant and a vowel (see fig. 3 for list of independent vowels along with their pronunciations).

Consonants and subscripts				
Letter				Sound
ɑ-series		ɔ-series		
Cons.	Sub.	Cons.	Sub.	
ក	◌្ក	គ	◌្គ	k
ខ	◌្ខ	ឃ	◌្ឃ	k^h
ង៉	◌្ង	ង	◌្ង	ŋ
ច	◌្ច	ជ	◌្ជ	c
ឆ	◌្ឆ	ឈ	◌្ឈ	c^h
ញ៉	◌្ញ	ញ	◌្ញ	ɲ
ដ	◌្ដ	ឌ	◌្ឌ	d
ឋ, ថ	◌្ឋ, ◌្ថ	ឍ, ធ	◌្ឍ, ◌្ធ	t^h
ណ	◌្ណ	ន	◌្ន	n
ត	◌្ត	ទ	◌្ទ	t
ប	◌្ប	ប៊	◌្ប	b
ផ	◌្ផ	ភ	◌្ភ	p^h
ប៉	◌្ព	ព	◌្ព	p
ម៉	◌្ម	ម	◌្ម	m
យ៉	◌្យ	យ	◌្យ	j
រ៉	◌្រ	រ	◌្រ	r
ឡ	◌្ឡ	ល	◌្ល	l
វ៉	◌្វ	វ	◌្វ	w
ស	◌្ស	ស៊	◌្ស	s
ហ	◌្ហ	ហ៊	◌្ហ	h
អ	◌្អ	អ៊	◌្អ	ʔ

Figure 1. Text to sound mapping for consonants and subscripts

Vowels		
Letter	Sound	
	ɑ-series	ɔ-series
Inherent vowel	ɑ:	ɔ:
◌ា	a:	i:ə
◌ិ	e	i
◌ី	ej	i:
◌ឹ	ə	ɨ
◌ឺ	œ:	ɨ:
◌ុ	o	u
◌ូ	ɔ:o	u:
◌ួ	u:ə	u:ə
◌ើ	a:ə	ə:
◌ឿ	ɨ:ə	ɨ:ə
◌ៀ	i:ɜ	i:ɜ
◌េ	e:	e̞:
◌ែ	a:ɛ	ɛ:
◌ៃ	aj	e̞j
◌ោ	a:o	o:
◌ៅ	aw	əw
◌ុំ	om	um
◌ំ	ɑm	um
◌ាំ	am	oam
◌ះ	ah	ɛah
◌ិះ	eh	ih
◌ឹះ	əh	ɨh
◌ុះ	oh	uh
◌េះ	eh	ih
◌ោះ	ɑh	uəh

Figure 2. Text to sound mapping for vowels and sequence of vowels

Independent	
Letter	Sound
អា	ʔa:
ឥ	ʔe
ឦ	ʔej
ឧ	ʔu
ឩ	ʔu:
ឪ	ʔo:w
ឫ	ʔɨ
ឬ	ʔɨ:
ឭ	lɨ
ឮ	lɨ:
ឯ	ʔa:ɛ
ឰ	ʔaj
ឱ	ʔa:o
ឲ	ʔa:o
ឳ	ʔaw

Figure 3. Text to sound mapping for independent vowels

II. 3. Letter to Sound (LTS) Building Process

The process requires the installation of Festival, Speech Tools and Festvox. And if the lexicon is in Unicode uft8 encoding, the latest version of Festival and Speech Tools from the website http://www.speech.cs.cmu.edu/15-492/assignments/hw2/packed/ (we have accessed this website on 26/03/2009) are required to do the preprocessing step (setup script). At the time of writing, Festival at the official website does not have the the correct setup script for Unicode utf8.
Before proceeding, we need to setup the 2 environment variables ESTDIR and FESTVOXDIR to the Speech Tools and Festvox folder respectively.

```
export ESTDIR=/home/seangmeng/soft/festival/speech_tools
export FESTVOXDIR=/home/seangmeng/soft/festival/festvox
```

Building letter to sound model in Festival involves the following steps:

- Preprocessing lexicon
- Defining the set of allowable pairing of letters to phones
- Constructing the probabilities of each letter/phone pair
- Aligning letters to an equal set of phones/_epsilons_
- Split the data into training set and test set
- Building CART models for predicting phone from letters (and context)

II.3.1. Preprocessing lexicon

The first step is to preprocess the lexicon to suit the training process.

- Add some non sense words
  - As mentioned in section 2, for consonants in a group that do not have their counterpart in another group, the diacritic marks ◌៉ and ◌៊ are used to change the group of those consonants to produce their counterpart. Some forms of C'V, where C' is a consonant followed by ◌៉ or ◌៊ and V is a vowel, do not appear in the lexicon. So we add some non sense words containing all possible forms of C'V so that the model can predict words containing ◌៉ and ◌៊ more accurately.
  - In our lexicon, only one word contains the independent vowel ឰ. If this word is removed from the training set, then the model will not know how to pronounce this vowel. So we add a few non sense words containing the independent vowel ឰ to make sure that this vowel exists in the training set.
- Remove all words containing diacritic mark LEK TOO ៗ
  - Diacritic mark LEK TOO ៗ is used to repeat the word or phrase that precedes it. In fact this diacritic mark are treated in Text Normalization phase. So we do not need to treat it here.
- For lexicon in Unicode uft8 encoding use setup script
  $FESTVOXDIR/src/lts/build_lts setup lex_file.scm utf8, where lex_file.scm is the file containing the compiled lexicon with the entry in the following format
  ("LLLLLL" nil (((p p) 0)((p p p) 0) ((p p) 0)))
  Example: ("សាលា" nil (((s a:) 0) ((l a:) 1)))

After preprocessing, the output will look like this:

```
( ("ស", "◌ា", "ល", "◌ា") nil (s a: l a:) )
```

II.3.2. Defining the set of allowable pairing of letters to phones

This step is to define all possible pairings of letters to phones. This definition is essential so that orthography can be aligned to transcriptions and ultimately probabilities can be extracted for each letter-to-phone pairing. It’s worth noting that the set of allowable letter to phone mappings is irrespective of context.[2]
The last version of festival and speech tools, as mention above, has script to create allowable paring of letters to phones automatically by running the script:

$FESTVOXDIR/src/lts/build_lts make_allowables

But the produced allowable is not good. One letter can have so many possible sound. And using this allowable make the subsequent tasks very long time to complete. It even shows error message “out of memory”. So we define the set of allowable pairing ourself. Initially this file should be of the form:

```
(require 'lts_build)
(%%stack-limit 10000000 nil)
(setq allowables
   ’(("ក៏" _epsilon_)
    ("ខ" _epsilon_)
    ...
    ("ៗលៗ" _epsilon_)
    (# #)))
```

where each letter can at least map to nothing. This set of allowables can then be modified by hand and built incrementally by running festival with the file and typing the following:

$FESTVOXDIR/src/lts/build_lts cummulate

This will prints out any lexical entries from the training set that couldn’t be aligned, illustrating which new letter to phone mappings should be added to the allowable ones. To allow multi-phone mappings where a letter gets paired with more than one phone, the syntax “phone1-phone2” is used. Of course, there are many entries such as words with irregular pronunciation which are contained in the training set and can never be aligned with their pronunciations.[2]
For phonetic symbols, we used ASCII representation of IPA. The following figures show the mapping between IPA and ASCII for Khmer phonetic that we use in Festival.

IPA	k	ŋ	c	ɲ	d	t	n	b	p	m	j	r	l	w	s	h	k^h	c^h	t^h	p^h	ʔ
ASCII	k	N	c	n.	d	t	n	b	p	m	j	r	l	w	s	h	kh	ch	th	ph	?

Figure 4. IPA to ASCII Mapping for consonants

IPA	ɑ	ɔ	a	e	ə	œ	o	u	ɨ	i	ɜ	ẹ̜	ɛ
ASCII	A	O	a	e	@	W	o	u	ix	i	Vx	e.	E

Figure 5. IPA to ASCII Mapping for vowels

In Khmer language, for words especially pali words, one letter can be mapped to a sound with three phones for example in the word ចរិត [*c a ʔ . r ə t*] (character), the consonant ច maps to [*c a ʔ*]. The current version of Festival does not support this. We can put this mapping, but the alignment process always fails. Our solution is as follow: we treat the combination of the two phones for example [a] and [*ʔ*] as one phone [*aʔ*]. After the letter to sound conversion, we rewrite the phone [*aʔ*] into two phones [*a*] and [*ʔ*].

Here is the allowable pairing of letters to phones that we have defined:

```
(require 'lts_build)
(%%stack-limit 10000000 nil)
(setq allowables '(
(# #)
( "!" _epsilon_ )
( "-"_epsilon_ )
( "ក" _epsilon_ k k-A: k-A k-O: k-O ? c k-a? kh k-a k-Ea? k-Ea k-u k-oa )
( "ខ" _epsilon_ kh kh-A: kh-A k kh-a kh-a? kh-Ea? kh-Ea kh-u kh-oa )
( "គ" _epsilon_ k k-A: k-A k-O: k-O k-Ea? k-a k-a? kh k-Ea k-u k-oa )
( "ឃ" _epsilon_ kh kh-A: kh-A kh-O: kh-O k kh-Ea? kh-a? kh-a kh-Ea kh-u kh-oa )
( "ង" _epsilon_ N N-A: N-A N-O: N-O n. N-Ea? N-a? N-Ea N-a N-u N-oa )
( "ច" _epsilon_ c c-A: c-A c-O: c-O c-a? c-a c-Ea? ch c-Ea c-u c-oa )
( "ឆ" _epsilon_ ch ch-A: ch-A ch-O: ch-O c ch-a? ch-a ch-Ea? ch-Ea ch-u ch-oa )
( "ជ" _epsilon_ c c-A: c-A c-O: c-O c-a? c-a c-Ea? ch c-Ea c-u c-oa )
( "ឈ" _epsilon_ ch ch-A: ch-A ch-O: ch-O c ch-a? ch-a ch-Ea? ch-Ea ch-u ch-oa )
( "ញ" _epsilon_ n. n.-A: n.-A n.-O: n.-O n.-Ea? n.-a n.-a? n.-Ea n.-u n.-oa )
( "ដ" _epsilon_ d d-A: d-A d-O: d-O d-a? d-a d-Ea? d-Ea d-u d-oa )
( "ឋ" _epsilon_ th th-A: th-A th-O: th-O t th-a? th-a th-Ea? th-Ea th-u th-oa )
( "ឌ" _epsilon_ d d-A: d-A d-O: d-O d-Ea? d-a d-a? d-Ea d-u d-oa )
( "ឍ" _epsilon_ th th-A: th-A th-O: th-O t th-Ea? th-a? th-Ea th-a th-u th-oa )
( "ណ" _epsilon_ n n-A: n-A n-O: n-O n-a? n-a n-Ea? n-Ea n-u n-oa )
( "ត" _epsilon_ t t-A: t-A t-O: t-O t-a? t-a t-Ea? th t-Ea t-u t-oa d )
( "ថ" _epsilon_ th th-A: th-A th-O: th-O t th-a? th-a th-Ea? th-Ea th-u th-oa )
( "ទ" _epsilon_ t t-A: t-A t-O: t-O t-Ea? t-a t-a? th t-Ea t-u t-oa )
( "ធ" _epsilon_ th th-A: th-A th-O: th-O t th-Ea? th-a? th-Ea th-a th-u th-oa )
( "ន" _epsilon_ n n-A: n-A n-O: n-O n-Ea? n-a? n-a n-Ea n-Ea? n-u n-oa )
( "ប" _epsilon_ b b-A: b-A b-O: b-O p-A: p-A p-O: p-O p p-a? p-a b-a? b-a p-Ea? b-Ea? p-Ea b-Ea b-u b-
oa )
( "ផ" _epsilon_ ph ph-A: ph-A ph-O: ph-O p ph-a? ph-Ea? p ph-a ph-Ea ph-u ph-oa )
( "ព" _epsilon_ p p-A: p-A p-O: p-O p-oa p-Ea? p-a? ph p-a p-Ea p-u p-oa )
( "ភ" _epsilon_ ph ph-A: ph-A ph-O: ph-O p ph-Ea? ph-a? ph-Ea ph-a ph-u ph-oa )
( "ម" _epsilon_ m m-A: m-A m-O: m-O m-Ea? m-a? m-Ea m-a m-u m-oa )
( "យ" _epsilon_ j j-O: j-O j-A: j-A j-a? j-a j-Ea? j-Ea j-u j-oa )
( "រ" _epsilon_ r r-A: r-A r-O: r-O r-Ea? r-a r-a? r-Ea r-u l r-oa )
( "ល" _epsilon_ l l-A: l-A l-O: l-O l-l l-a? l-Ea? l-Ea l-a l-u l-oa )
( "វ" _epsilon_ w w-A: w-A w-O: w-O w-Ea? w-a? w-Ea w-a w-u w-oa p p-Ea p-a p-Ea? p-a? )
( "ស" _epsilon_ s s-A: s-A s-O: s-O h s-a s-a? s-Ea? s-Ea s-u s-oa h-s )
( "ហ" _epsilon_ h h-A: h-A h-O: h-O h-a? h-a h-Ea? h-Ea h-u h-oa )
( "ឡ" _epsilon_ l l-A: l-A l-O: l-O l-a l-a? l-Ea? l-Ea l-u l-oa )
( "អ" _epsilon_ ? ?-A: ?-A ?-O: ?-O ?-a? ?-Ea? ?-a ?-Ea ?-u ?-oa )
( "ឥ" _epsilon_ ?-e ?-i )
( "ឦ" _epsilon_ ?-e-j ?-i: )
( "ឧ" _epsilon_ ?-u )
( "ឪ" _epsilon_ ?-u: )
( "ឫ" _epsilon_ r-ix )
( "ឬ" _epsilon_ r-ix: )
( "ឭ" _epsilon_ l-ix )
( "ឮ" _epsilon_ l-ix: )
( "ឯ" _epsilon_ ?-aE )
( "ឰ" _epsilon_ ?-a-j )
( "ឱ" _epsilon_ ?-ao )
( "ឳ" _epsilon_ ?-a-w )
( "ា" _epsilon_ a: a i@ Ea oa )
( "ិ" _epsilon_ e i e: @ ix )
( "ី" _epsilon_ e-j i: @-j )
( "ឹ" _epsilon_  @ ix )
( "ឺ" _epsilon_ W W: ix: W: )
( "ុ" _epsilon_ o u )
( "ូ" _epsilon_ Oo u: o )
( "ួ" _epsilon_ u@ )
( "ើ" _epsilon_ a@  @: )
( "ឿ" _epsilon_ ix@ )
( "ៀ" _epsilon_ iVx )
( "េ" _epsilon_ e.: e: e. e aE )
( "ែ" _epsilon_ aE E: i E )
( "ៃ" _epsilon_ a-j e.-j )
( "ោ" _epsilon_ ao o: u@ )
( "ៅ" _epsilon_ a-w @-w )
( "ំ" _epsilon_ a-m u-m m )
( "ះ" _epsilon_ a-h Ea-h h )
( "ៈ" _epsilon_ a? Ea? Ea-h h )
( "៉" _epsilon_ )
( "់" _epsilon_ )
( "៊" _epsilon_ )
( "៌" _epsilon_ )
( "៍" _epsilon_ )
("៎" _epsilon_ )
( "៏" _epsilon_ )
( "័" _epsilon_ )
( "៓" _epsilon_ )
( "ៗ" _epsilon_ )
( "។ល។" _epsilon_ l-a? )
( "" _epsilon_ )
( "" _epsilon_ ) ))
```

II.3.3. Constructing the probabilities of each letter/phone pair

Once the number of failed alignments is satisfactorily low, *$FESTVOXDIR/src/lts/build_lts cummulate* can be allowed to run to completion. This permits the function to count how many times each letter-to-phone mapping occurs in allowable alignments, thus supplying the necessary information for calculating probabilities. The script *$FESTVOXDIR/src/lts/build_lts cummulate* also calculates the probabilities and save them to a file.

II.3.4. Aligning letters to an equal set of phones/_epsilons_

The next step is to find the best alignment of the number of letters to an equally long string of phones, multi-phones and epsilons. This is done by executing the following commands:

```
$FESTVOXDIR/src/lts/build_lts  align
```

Entries which look like the following will be produced in the file lex.align.

```
( ( "ក" "ក" "ា" "ត" ) nil k-A: k a: t )
```

The above script also builds the feature vectors necessary for the last step.

II.3.5. Split the data into training set and test set

The following command is used to divide the data generated so far into training set and test set.

```
$FESTVOXDIR/src/lts/build_lts  traintest
```

This command puts every tenth entry into the test set and everything else in the training set.

II.3.6. Building CART models for predicting phone from letters (and context)

The final step in the LTS training process is the building of CART trees which will be used to predict the pronunciation of any words not found through lexicon lookup. The program wagon which is provided by the EST, can be used to automatically build suitable trees for such a task. Classification trees will be used in this case, where the features considered will be names of items in the Segment relation (four phone names to the left and right of current segment will be considered). In order to build the trees, features must be fed in a format compatible with wagon. Files known as feature vectors are already built in step 4.

The model can be build by running the following command:

```
$FESTVOXDIR/src/lts/build_lts  build
```

Once the models are created they must be collected together into a single list structure.
The trees generated by wagon contain fully probability distributions at each leaf, at
this time this information can be removed as only the most probable will actually be
predicted. This substantially reduces the size of the tress.
This task can be carried out using the following command:

```
$FESTVOXDIR/src/lts/build_lts  merge
```

This will produce a file containing a set!
for the given variable name to an assoc list of letter to trained tree
which can be used in Festival to predict pronunciation of words not found in lexicon.

II.4. Result

Training from	Train Set		All Data	
Testing from	**Test Set**	**All Data**	**Test Set**	**All Data**
Phone accuracy	92.01%	94.78%	95.18%	95.20%
Word accuracy	63.11%	74.75%	75.69%	76.40%

All Data:	17816 words
Train Set:	16035 words
Test Set:	1781 words

Figure 6. Accuracy of LTS Model

From the above figure, we see that the model trained from all data give better result because the test set includes in the training data. We decide to use LTS model trained from all data in our Khmer TTS.

II.5. Conclusion

We have described the process of building letter to sound model in Festival. After preprocessing, the lexicon used to build the model contains 17816 words. The accuracy of the resulted model trained from all data and tested on all data is 76.40%.

III. Information in the lexicon

Our pronunciation lexicon is based on official Khmer dictionary (Chuon Nat dictionary). The entries have already been collected by PAN Cambodia team in phase I. So we just take those entries (18632 words) and add the following information: *sound, sound2, sound_type, ipa* as detailed below.

- *sound*: word pronunciation using script not IPA that is obvious to pronounce.
- *sound2*: alternative pronunciation. In case of there 2 possible pronunciations, *sound* store the most used one.
- *soud_type*: type of the pronunciation. It can be *regular*, *irregular* and *change*.
  - *regular*: regular or normal pronunciation. The word follows simple pronunciation rules.
  - *irregular*: irregular or abnormal pronunciation. The word does not follow simple pronunciation rules.
  - *change*: group change in pronunciation. The word follows simple pronunciation rules, but there is group change (see report on letter to sound conversion section 3.2).
- *ipa*: word pronunciation using IPA.

sound, which contains word pronunciation using script (not IPA), is used to facilitate the data entry process. The pronunciation in form of IPA can be derived automatically from it. Inputing pronunciation in script form is much easier than doing the same thing using IPA, especially for people who are not familiar with phonetic or IPA.
sound_type information can be used to create statistical model to predict group change. The group change model can then be integrated with rule-based method for letter to sound conversion.
ipa, which can be derived automatically from *sound*, is used to store pronunciation of word in form of IPA. It is the one that will be actually used in our Khmer Text To Speech.
Note that all the pronunciations includes syllable boundary.

III.1. Building process

To facilitate data entry, we develop a simple web-based application using PHP / MySQL. This application allows multiple users to input data at the same time without conflict (1 entry can not be entered by more than one user).

We have 3 steps to follow in order to build our pronunciation lexicon. The first step is to input *sound* (the word pronunciation using script), *sound2* (alternative pronunciation) and *sound_type* (regular, irregular or change). This step can be done even by people who do not have knowledge in phonetic. For word with complex pronunciation, we use electronic version of Chuon Nat dictionary developed by Buddhist Institute to check the pronunciation.
After finishing the first step, we can do the second step which consists of generating pronunciation in IPA from the pronunciation in script. We wrote a program in Python to do this task.
The last step is for quality assurance. For this step, we have a linguist to verify the data entered in step 1 and the pronunciation in IPA generated automatically by the program in step 2. As we do not have much resource, only 1 linguist does the verification.

III.2. Conclusion

We have described the building process of our pronunciation lexicon. The lexicon is based on Chuon Nat dictionary which contains about 20K words. The lexicon includes information concerning group change in word which can be used to build probabilistic model to predict group change. This model can be added to rule-based method for letter to sound conversion to improve its accuracy.

III.3. Future work

Our lexicon is based on official Khmer dictionary which are not well updated. Many new words do not appear in the dictionary. Our future work is to collect the words which are not in the dictionary from text corpus (that PAN Cambodia team has already collected) and add those words in our lexicon.

IV. Text normalization

IV.1. Introduction

This report introduces the algorithms of how the digits' sequence of Khmer script is read. It is a part of text normalization to extend the nonsense word (NSW) in to reading word.

IV.2. Methodology

khmer_digits=[សូន្យ, មួយ, ពីរ, បី, បួន, ប្រាំ, ប្រាំមួយ, ប្រាំពីរ, ប្រាំបី, ប្រាំបួន]
khmer_two_digits=[ដប់, ម្ភៃ, សាមសិប, សែសិប, ហាសិប, ហុកសិប, ចិតសិប, ប៉ែតសិប, កៅសិប]
khmer_level_number=[‘’,’’,រយ, ពាន់, ម៉ឺន, សែន, លាន, កោដ]

Note: the table of the index is from 0

Figure 1: variable of khmer number

***eight_digits_expands* algorithms**

Input:

- ***digits***: String of Khmer digits and its length is smaller then 9

Output:

- ***toReturn***: String of Khmer digits in reading

Local variable:

- ***index_i***: integer, index are used in loop

```
Initialize toReturn = “”
Initialize index_i = 0

For  index_i = 1 To length(digits) Do
        If (digits[index_i] <> ‘0’)
        Begin
        If ( index_i == 2) Then
                        toReturn = khmer_two_digits[toInteger(digits[index_i])] + toReturn
        Else
                        toReturn = khmer_digits[toInteger(digits[index_i])] +
                        khmer_level_number[length(digits) - index_i] + toReturn
        End;
End For;
```

Resutl: Every digit is read according to its digit’s name and its level. The digit ‘0’ is not read. This function can only read the Khmer digits less then 9 digits.

Example:

០ ▪””

១២៣ ▪ មួយរយម្ភៃបី *(one hundred and twenty three)*

***khmer_number expands* algorithms**

Input:

- ***digits***: String of Khmer digits

Output:

- ***toReturn***: String of Khmer digits in reading

Local variable:

```
Initialize: tmp = digits
If ((length(tmp) == 1) and (tmp[0]= ‘0’)) Then
        toReturn = khmer_digits[0]
Else
Begin
        While (length(tmp) >= 9 ) Do
        Begin
                toReturn = “កោដ”+eight_digits_expands(subString(tmp, length(tmp) – 7
                        length(tmp))) + toReturn;
                tmp = subString(tmp, 0, length(tmp) – 7);
        End While;
        If (length(tmp) != 0) Then
                toReturn =  eight_digits_expands(tmp) + toReturn;
End If;
```

Result: If the number of the digits is greater then 8, it will cut in to N parts of 7 digits and it add the word កោដ at the end of the first N-1 parts.

Example:

១០០០០០០០០ ▪ ដប់កោដ ***(100000000)***

- **Implementation**

The algorithms have been implemented in **Java** Language and the accuracy is good. Now we are implementing in scheme language.

V. Syllabification

V.1. Introduction

In the previous report, we presented the algorithm and the test done on Java for the syllabification. However, the data used for the previous experimentation is not good enough for the evaluation. Thus, in this report, we will present about our experimentation and the result of the syllabification.

V.2. Syllabification test system

Our system used to test the syllabification algorithm to see its accuracy is written in Java which connects to the database system MySQL. The system can be used to syllabify one phonetic word or can connect to the database system to retrieve the words and syllabify them. Moreover, we store some necessary information needed for the syllabification, this time, in a normal text file.

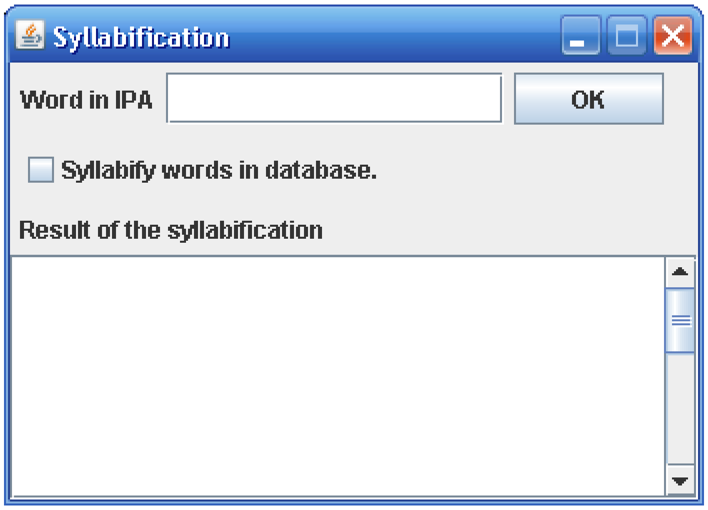


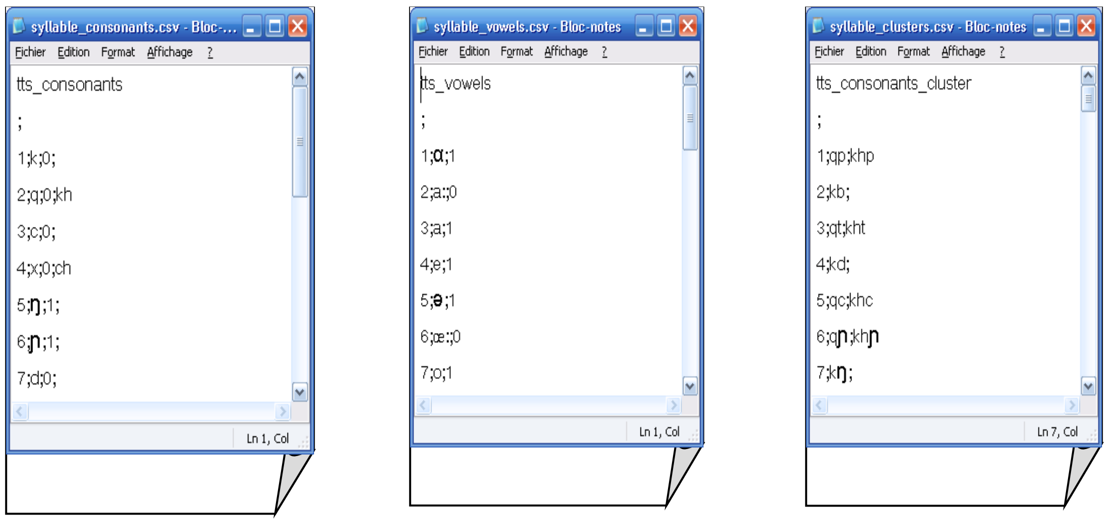
Figure 1: Main interface of syllabification test system

Description

Control name	**Description**
Textbox	Input the text for the syllabification.
Button	To start syllabifying.
Checkbox	**Check**: the syllabification will be done on data read from MySQL database. **Uncheck**: syllabify the word input in text box.
TextArea	Display the syllabification result.

Data files

As mentioned earlier, we store all necessary information for the syllabification such as, consonants, vowels, and the clustering consonants, in text file. Here, below, we present them.


syllable_consonants.csv - Bloc-...
Fichier Edition Format Affichage ?
Ln 1, Col
syllable_vowels.csv - Bloc-notes
Fichier Edition Format Affichage ?
Ln 1, Col
syllable_clusters.csv - Bloc-notes
Fichier Edition Format Affichage ?
Ln 7, Col

```
tts_consonants
;
1;k;0;
2;q;0;kh
3;c;0;
4;x;0;ch
5;ŋ;1;
6;ɲ;1;
7;d;0;
```

```
tts_vowels
;
1;ɑ;1
2;a:;0
3;a;1
4;e;1
5;ə;1
6;æ:;0
7;o;1
```

```
tts_consonants_cluster
;
1;qp;khp
2;kb;
3;qt;kht
4;kd;
5;qc;khc
6;qɲ;khɲ
7;kŋ;
```

Figure 2: The content of consonant file, vowel file, and consonant cluster.

V.2. Result & conclusion

In this experimentation, we use our lexicon which contains **18632 words** storing in MySQL server. To evaluate the result of the syllabification, we compare the syllabified word given by our system with the manually syllabified words stored in our lexicon. Our system produces **17 997 words** correct which equals **to 96.59%**. However, we consider that our algorithm provides high-accuracy and acceptable result because the problem causing incorrectness of the syllabification comes from: - *misspell of word in IPA*, and *some incoherence uses of the IPA*.

VI. Word segmentation technique

VI.1. Introduction

Almost all techniques to statistical language processing, including speech recognition, machine translation and information retrieval, are based on words. Although word-based approaches work very well for western languages, where words are well defined, it is difficult to apply to Khmer. Khmer sentences are written as characters strings with no spaces between words. Space is inserted in the sentence under some circumstance such as breathing break in reading, number break etc… To human perception, it is very easy to judge the break point of the word in the sentence, but it is very difficult for the computer to identify the fact. So far, no research has been conducted on the topic for Khmer language.

In this report, we present the technique of word segmentation for Khmer and the improvement of its defined algorithms as our work is based on the previous research result.

Problems

As mentioned in the report done by PAN research group, the two major problems in Khmer word segmentation. They are:

- *Ambiguity issue*: Khmer language has no word boundary and has no exact definition to identify a word. Moreover, a word can be established using many other words. Therefore, the problem of ambiguity is very principle. For example:
    - លើក = លើក or លើ|ក
    - ប្រជាជាតិខ្មែរ = ប្រជា|ជាតិ|ខ្មែរ or ប្រជាជាតិ|ខ្មែរ or ប្រជា|ជាតិខ្មែរ

- *Unknown word identification*: Unknown words are defined as words that are not in the lexicon. Unknown words can be categorized as many types: *error words, abbreviation (acronym), proper names, derived words, compounds, and numeric type compound.*

Propositions

1. **Methods solving the ambiguity issue:** the research results two proposed algorithms for solving this type of problem, *the maximum matching algorithm,* and *orthographic syllable Bi-gram model.* The main idea of the maximum matching algorithm is to choose the segmentation with the smallest number of words. However, there is an uncertainty when two segmentations have the same number of words.

2. **Error word detection method:** the scope of the research is the problem of sound similarity problem which means that different writing form results the same pronunciation. To solve this problem, one has proposed the used of **Khmer Common Expression** which will represent the word with the same sound to one common writing expression.

VI.2. Goal of the research

Even the solutions proposed for the previous research are made but it doesn't response to our case. We need to have an algorithm of word segmentation which is high accuracy and is high speed. Looking back to algorithm proposed, the *maximum matching algorithm* is fast but the accuracy is not acceptable. As for *word and orthographic syllable bi-gram model* the accuracy of the segmentation is acceptable but not its speed.

The main goal of this research is to find the solution to speed up the segmentation process using *word bigram model.*

VI.2.1. Proposition

Word bigram algorithm used bigram data which stored in files in disk. Each time the system needs that kind of information, it reads from files which slow down the segmentation process. To speed it up, we proposed to load all word bigram data into memory by defining a proper data structure as present in **Figure 1**, which will use less memory storage. However, we use the same technique as the previous one by loading all data in **index file** into memory storing in a hash table. Each element of hash table is the instance of the class **WordInfo** which contains the index of word, and other related information. To adapt to our new proposition, we change on each element of hash table, from the instance of **WordInfo** into its subclass **TTSWordInfo** which possesses the root of binary tree research linking to a tree of all indexes of the related second words.

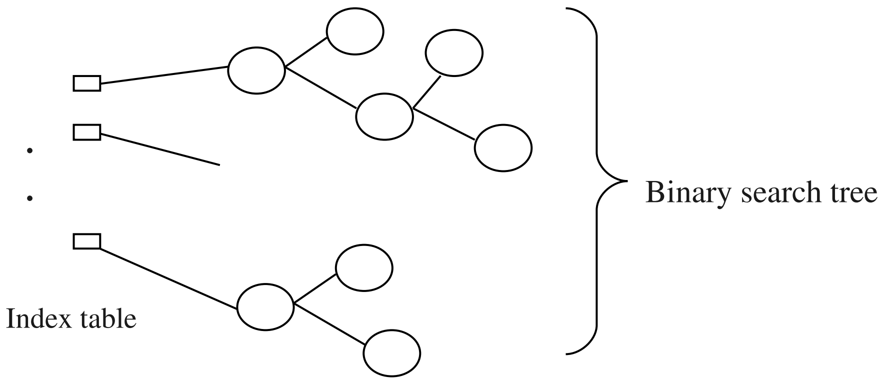


Figure 1: Data structure used to store bigram data in memory

Each time we need the bigram information of two words, we just find the index of the first word and the second word, and then search the index of second word on the tree in the first word. Seeing the second word, we can have the bigram information of those two words.

Our proposition needs the modification of the storage structure of the bigram data because we need to speed up our loading when the application starts. The previous storage structure enables storing data in folders whose name can be calculated using the index of the first word. In another hand, each folder contains many files which stored basically the index of the second word and the related bigram information (cf. [1]). Our change made provides a simple storage structure, a simple algorithm, and a faster loading process. In the following figure, we present the storage structure we proposed.

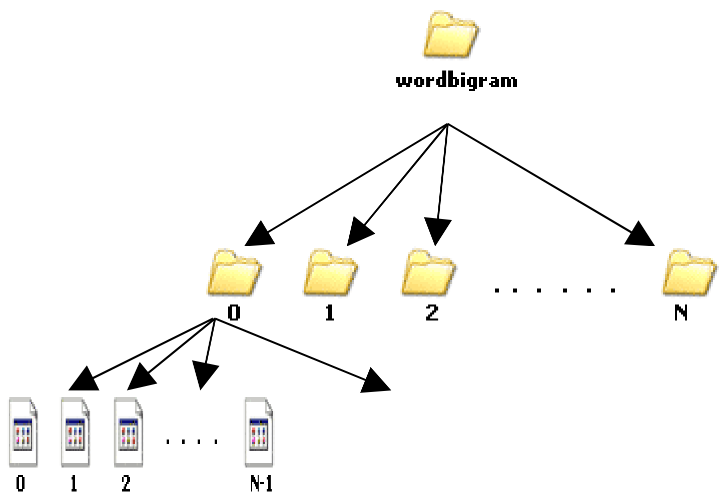


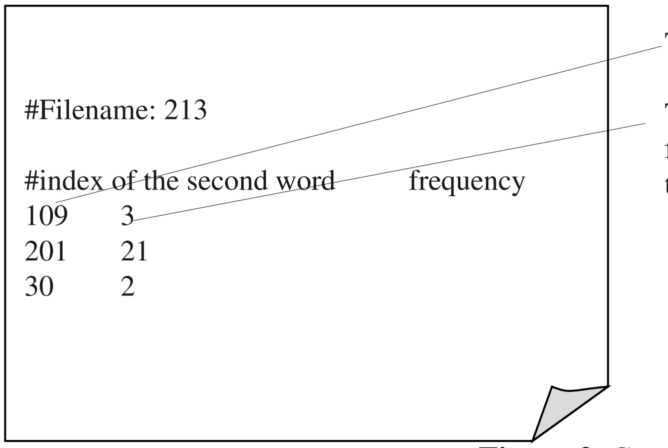
Figure 2: Storage structure of the bigram data.

The root folder, named **wordbigram**, contains many subfolders noted **M** and each subfolder contains **N** files. The number of files stored in each subfolder varies based on the number of files recorded in **index file** (We can say **N = Number of file in each folder**). The subfolder named numerically starting from **0**. So do the files they contain, which are named using **index of the word**. For example, the file named **231** corresponds to the data of word having index **231**.

Technically, to find the name of folder **F** of a word whose index **i** and each folder contains **N** files, we use the following formula:

$$F = (i - (i \text{ modulo } N)) / N$$

Below, we present the content of the bigram data in each subfolder.



```
#Filename: 213

#index of the second word          frequency
109      3
201      21
30       2
```

The index of the second word.

The frequency of the bigram which is taken from the corpus of the word bigram data in the previous research. Cf. [1]

Figure 3: Content of the bigram data file.

Diagram class

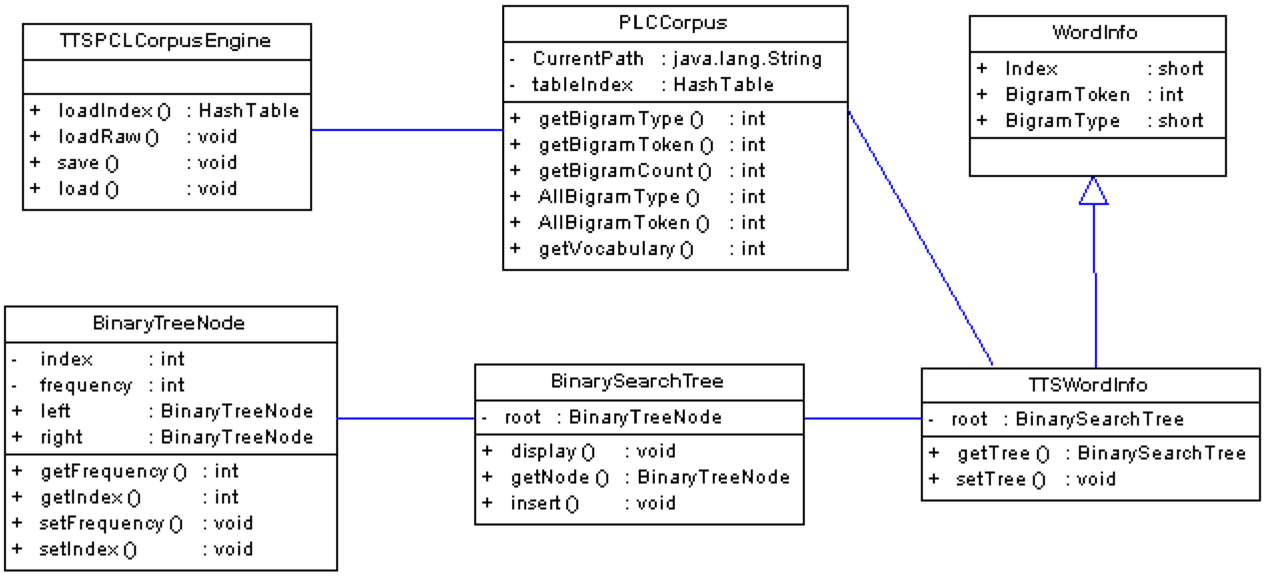


Figure 4: Class diagram of new classes

Class diagram description

The tables below describe the elements of each class presented in the class diagram above.

- PLCCorpus

cf. [1]

- TTSPLCCorpusEngine

Class name	**TTSPLCCorpusEngine**	
Description	This class provides four static methods which used to load and save bigram data.	
Attributes	N/A	
Operations	**loadIndex ( )**	Load the content of index file into hash table.
	loadRaw ( )	Load the bigram data using previous proposed storage structure (cf. [1]) into our new data structure binary search tree.
	load ( )	Load the bigram data using our current storage structure into our new data structure binary search tree.
	save ( )	Save the content in our binary search tree into files of our new storage structure.

- WordInfo

Class name	**WordInfo**	
Description	This class represents each line of the data in the index file.	
Attributes	**Index**	Store index number
	BigramToken	Store number of the bigram token
	BigramType	Store number represents the bigram type.
Operations	N/A	

- **TTSWordInfo**

Class name	**TTSWordInfo**	
Description	This class inherits from the class **WordInfo.** It contains an additional attribute, named **root**, which is the root of the binary search tree. In the tree, we store the index of the second word pair and the frequency the two words have.	
Attributes	**root**	Root of the binary search tree.
Operations	**getTree ( )**	Return the root of the tree.
	setTree ( )	Set the root of the tree.

- BinaryTreeNode

Class name	**BinaryTreeNode**
Description	This class represents the node of the binary search tree.

Attributes	**index**	Store index number, normally, of the second word.
	frequency	Store the frequency.
	left	Left hand of the node used to point to another node on the left.
	right	Right pointer of the node.
Operations	**getIndex ( )**	Return the index of the word.
	getFrequency ( )	Return the frequency of the two words pair.
	setIndex ( )	Set the index of the word.
	setFrequency ( )	Set the frequency.

- BinarySearchTree

Class name	**BinarySearchTree**	
Description	This class represents the binary search tree. It provides some functionality to manipulate on the tree.	
Attributes	**root**	It represents the root of the binary search tree.
Operations	**insert ( )**	Insert an element to the tree.
	getNode ( )	Return a node which contains the index passed as parameter.
	display ( )	Display all contains of the tree.

VI.3.Conclusion

We proposed a new data structure which will be used to store the bigram data on memory and the change of storage structure of data on disk. However, we need to do the experimentation to prove that this proposition response to speed up the segmentation process. We hope to delivery our testing result in our next report.

VII. Diphone database module

VII.1. Introduction

According to the previous report the phonetic analysis of khmer sound can be divided into 21 consonants, 21 vowels (12 long vowels and 9 short vowels) and 10 diphthongs. In order to adapt with the new words in the future we decide to create a combination of all diphones possible.

Suppose that:

C	= consonant
LV	= long vowel
SV	= short vowel
D	= diphthong
P	= silent

Diphone form:

1. C - LV	: 21 * 12 = 252
2. LV - C	: 12 * 21 = 252
3. C – SV	: 21 * 9 = 189
4. SV – C	: 9 * 21 = 189
5. C – D	: 21 * 10 = 210
6. D – C	: 10 * 21 = 210
7. C – C	: 21 * 21 = 441
8. LV – SV	: 12 * 9 = 108
9. LV – D	: 12 * 10 = 120
10. LV – LV	: 12 * 12 = 144
11. SV – SV	: 9 * 9 = 81
12. SV- D	: 9 * 10 = 90
13. SV – LV	: 9 * 12 = 108
14. D – SV	: 10 * 9 = 90
15. D – D	: 10 * 10 = 100
16. D – LV	: 10 * 12 = 120
17. P – C	: 21
18. C – P	: 21
19. P – LV	: 12
20. LV – P	: 12
21. P – SV	: 9
22. SV – P	: 9
23. P – D	: 10
24. D – P	: 10
25. P – P	: 1

So the total diphone is 2809. However some word we borrow from English doesn’t exist in our phonetic here, for example the phone “er” in the word “teacher” in English. So to make our speech processing tools handle this thing, supplemental diphones are needed.

VII.2. Phonetic symbol

Some specials character in the real phonetic are difficult to do the treatment in computer so we propose some symbol for all phonetics to solve this problem.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
ប៉ព	បប៊	ផភ	តទ	ដឌ	ថធឋឍ	ចជ	ឆឈ	កគ	ខឃ
P	b	ph	t	d	th	c	ch	k	kh
P	b	ph	t	d	th	c	ch	k	kh

C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21
អអ៊	មម៉	ណន	ញញ៉	ងង៉	សស៊	ហហ៊	រ៉	លឡ	វ៉	យយ៉
ʔ	m	n	ɲ	ŋ	s	h	r	l	w	j
?	m	n	n.	N	s	h	r	l	w	j

Table-01 Consonants symbol

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
◌ិ	◌ី	◌ឹ	◌ើ	◌ុ	◌ូ	◌ិ	េ◌	◌ី	◌ុ
អ៊	អ៊̲	អ	អ៊̲	អ៊	អ៊̲	អ	អ̲	អ̲	អ
I	i:	ə	ə:	u	u:	E	e:	Œ:	o
I	i:	@	@:	u	u:	E	e:	W:	o

V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21
េ◌ា	អ៊់	អ៊	◌ា◌់	◌ា	អ់	អ	◌ឹ	◌ឺ	េ◌	ែ◌
អ៊̲	អ៊	អ៊̲	អ	អ̲	អ	អ̲	អ៊	អ៊̲	អ៊̲	អ៊̲
o:	ɔ	ɔ:	a	a:	ɑ	ɑ:	ɨ	ɨ:	ẹ:	ɛ:
o:	O	O:	a	a:	A	A:	Ix	ix:	e.:	E:

Table-02 Vowels symbol

D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
◌ែ	◌ើ	◌ូ	◌ួ	◌ោ	◌ៀ	◌ឿ	◌ា	◌ា◌់	◌ា◌់
អ	អ	អ	អ៊	អ	អ៊	អ៊	អ៊	អ៊	អ៊
a:ɛ	a:ə	ɔ: o	u:ə	a:o	i:ɜ	ɨ:ə	i:ə	oa	ɛa
aE	a@	Oo	u@	ao	iVx	ix @	i@	oa	Ea

Table-03 Diphthongs symbol

VII.3. Non sense word

The technique to record all khmer diphones is to create non sense word which include each diphone inside. To get the best diphone signal we add unnecessary word before and after the diphone which is needed in our project.

C-LV and LV-C

The format C – LV and LV – C can be combining by using a non sense word C - LV - C

Example:

តាប៉េប៉េ (ta pe: pe:)

<ta> is an unnecessary word
<pe:> is diphone we need in the format of C – LV
<e:p> is another diphone we need in the format of LV- C
<e:> the last e: is also an unnecessary sound

C-SV and SV-C

The format C – SV and SV – C could not be combining by using a non sense word C – SV- C, because between SV and the last C there existe an embedded consonent sound C11 (ʔ). So we need one non sense word for each format C – SV and SV – C.

តាបិបិ(ta pE pE) we get <pE> diphone

តាប៊ិពតា (ta pEp ta) and we get <Ep> diphone

To get a complete non sense words, reference to the non sense words document.

VII.4. Conclusion

After recording all the non sense word we can uses the praat software to notice starting point, middle point and ending point of a diphone in one non sense word signal. The problems we face in this phase are:

- Incorrect phonetic in the lexicon library file
- Define the starting point and the ending point of a diphone from the non sense word signal

- The duration of the vowel (short or long)
- Some mistake during recording period

To solve the problems above, we check the incorrect diphone and do the record again, incorrect phonetic can be modified in the lexicon library file, short and long duration of vowel will be applied by using the technique of how to make an intonation of the word.

VIII. General conclusion

Natural languge processor is an important component in Text to Speech application. we have spent a significant time on this part to make sure that the application created is acceptable and reliable. the researches included, the text to sound conversion, text normalization, sylibification and diphone identification. The research is completely done. The outcome of our research is acceptable. However some works such as text to sound conversion, the accuracy can not be reached 100% cause by the complexity of word form of khmer language.

References:

[1] http://www.seasite.niu.edu/khmer/writingsystem/writing_introduction/intro_set.htm,
[2] Developing a New Voice for Hiberno-English in The Festival Speech Synthesis System, Nigel Rochford
[3] http://en.wikipedia.org/wiki/Speech_synthesis, 09/10/2008
[4] Khmer Word Segmentation project, PAN Localization Team
[5] Binary search tree, http://en.wikipedia.org/wiki/Binary_search_tree