Rapport de Soutenance finale

SD\n – ALIAS

<*A Lexeme Is A Sound*>

Julie THÉODOSE-SOREL Mathieu BASCLE
Nicolas TANDÉ Yohan BAINIER

20 juin 2005

Table des matières

1 Introduction **2**
1.1 Présentation de l'équipe 2
1.1.1 Pourquoi SD\n et ALIAS ? 2
1.1.2 Membres de l'équipe 3
1.1.3 Moyens techniques 4
1.2 Présentation du projet 5
1.2.1 Définition du TTS 5
1.2.2 Objectifs 5
1.2.3 Plateformes 6
1.2.4 Environnement 6

2 Autotools **7**
2.1 Présentation 7
2.2 Dépendances 8
2.2.1 configure.in 8
2.2.2 Makefile.am 9
2.3 Code source portable 10
2.4 Bootstrapping 10
2.4.1 aclocal 10
2.4.2 libtoolize 10
2.4.3 autoheader 11
2.4.4 automake 11
2.4.5 autoconf 11
2.5 Utilisation 11

3 Dictionnaire **12**
3.1 Présentation 12
3.2 Représentation 12
3.2.1 A l'extérieur 12
3.2.2 A l'intérieur 13
3.3 Tri du dictionnaire 13

3.3.1 Tri par insertion 13
3.3.2 Tri rapide 15
3.4 Utilisation 17

4 Traitement du Langage Naturel 18
4.1 Phonétique 18
4.2 Analyseur syntaxique 21
4.2.1 Vérification 21
4.2.2 Détection 21
4.2.3 Conversion 22
4.3 Des chiffres et des lettres 27
4.3.1 Traduction d'un paquet 28
4.3.2 Diviser pour régner 29

5 Partie audio 31
5.1 Choix du format audio 31
5.1.1 PCM 31
5.1.2 MP3 31
5.1.3 Ogg Vorbis 32
5.1.4 Speex 32
5.1.5 Notre Choix 32
5.2 Principe 32
5.2.1 Différents Threads 32
5.2.2 Contrôle de la Thread audio et passage de sons . . . 32
5.3 Implémentation 33
5.3.1 Ogg Vorbis 33
5.3.2 xBSD et Linux 2.4 : OSS 33
5.3.3 Linux 2.6 : ALSA 33
5.3.4 Mac OS X 33
5.4 Vitesse et tonalité 34
5.4.1 Le format PCM 34
5.4.2 Méthode de base : vitesse et tonalité 34
5.4.3 Seconde méthode : vitesse 34
5.4.4 Synthèse 35
5.5 Correspondance phonème et nom de fichier 35
5.6 Inclusion du son dans l'interface 35
5.7 Utilisation des Autotools 36
5.8 Exportation vers les différents formats 36
5.8.1 Le format *PCM* brut : le *RAW* 36
5.8.2 Le format Microsoft *WAVE* (ou *WAV*) 36
5.8.3 Le format *Ogg/Vorbis* 37

6 Interface **38**
6.1 Qu'est ce que *Qt* ? 38
6.2 Pourquoi *Qt* ? 38
6.3 Démarrer avec *Qt* 38
6.4 Présentation de L'interface 39
6.4.1 *Qt* Designer 39
6.4.2 Définition des objets utilisés 39
6.4.3 Première soutenance 40
6.4.4 Seconde soutenance 40
6.4.5 Troisième soutenance 41
6.4.6 Soutenance finale 42

7 Annexes **43**
7.1 Installation 43
7.1.1 Configuration 43
7.1.2 Compilation 43
7.1.3 L'installation 44
7.2 Site web 45
7.3 Bibliographie 46

8 Conclusion **47**

Chapitre 1

Introduction

1.1 Présentation de l'équipe

1.1.1 Pourquoi SD\n et ALIAS ?

Tout d'abord, notre équipe se compose de quatre étudiants actuellement en Info SPÉ au sein de la prestigieuse EPITA. Nous avons choisi le nom de groupe SD\n pour une raison très simple : nous faisons partie d'un département secret à l'intérieur même du groupe A2 appelé les "Back slash n" fondé exclusivement pour veiller au bon déroulement de tout les événements qui pourraient survenir lors de l'année scolaire, ayant rapport de près ou de loin avec notre groupe de SPÉ, et ainsi contrôler tous les débordements possibles pour assurer la suprématie totale de notre vision du code par rapport à celle des autres élèves.

Quant à ALIAS, le choix est tout aussi symbolique. Cet acronyme ne signifie pas uniquement "A Lexeme Is A Sound", il est aussi en rapport avec nos activités clandestines. Plus sérieusement, le nom de notre utilitaire définit parfaitement ce qu'il est, à savoir un Text To Speech basé sur la reconnaissance de phonèmes.

Mais finalement, qui sont les SD\n ? Des étudiants syphoné ou bien alors les plus grands génies de tout les temps ? Peut être même les deux à la fois qui sait...

1.1.2 Membres de l'équipe

Mathmath

Je m'appelle Mathieu BASCLE, 21 ans, recruté par le SD\n cette année. Notre chef m'a approché malgré mes faibles connaissances en langage C car je sais travailler en équipe, que je suis quelqu'un d'organisé et que je ne trahirai jamais la cause. Travailler sur un Text To Speech va me permettre d'approfondir ma connaissance du langage C et me permettre de manipuler des algorithmes que je n'aurais peut être pas eu l'occasion de manipuler sinon. A part passer mes journées devant un écran d'ordinateur ou de jouer les taupes parmi les étudiants normaux, je lis des mangas et mêne une vie relativement normale.

Nittch

Ayant à plusieures reprises testé des utilitaires de synthèse vocale, j'étais particulièrement intéressé par le projet. De plus le manque d'applications francophones dans ce domaine m'a motivé pour en faire mon projet.

Juliju

Je suis Julie THÉODOSE SOREL, dit Juliju. Ma passion dans la vie, c'est de savourer des cookies, avec du reblochon, ou tout simplement, savourer un grand bol de thé avec quelques flocons de maïs, surnageant à sa surface. Toutefois, l'informatique tient une grande place dans ma vie. Et là, j'ai rencontré Mathieu et Yohan, qui ont transformés ma vie. Je veux dire qu'ils m'ont, en effet, dans un premier temps, proposé de participer à leur grand projet : un traducteur algo/C. Mais, cependant, ce projet n'était pas à la hauteur des attentes de Krisboul, et nous avons choisi d'un commun accord, de faire un "Text To Speech". Mon recrutement pour intégrer ce projet, a été bien dur : course d'obstacles à travers les couloirs, défis nombreux et variés (les légendaires "t'es pas cap'..." et la mythique série de photos des assistantes d'anglais[1]) à accomplir, de façon à montrer à notre chef de projet, ma motivation. Une fois les tests passés, j'ai pu enfin intégrer le département SD\n, pour mon plus grand bonheur. Ce projet de grande envergure, me permettra donc de progresser et d'apprendre de nouvelles choses, sous la houlette de Yohan-sama.

[1] A l'insu de leur plein grés (http ://sdalias.sourceforge.net/misc/photos/assistantes/)

Alceri

Je m'appelle Yohan BAINIER, Steeve FRANK calls me "Donut"...J'ai pas mal de loisirs mais mes deux favoris sont l'analyse mytiligraphique et la recherche du prénom de Jennifer aux p'tites oreilles. En effet, après avoir essuyé un monstrueux rejet du traducteur Algo/C par Krisboul, je me suis consolé en essayant de trouver le prénom de Jenna (c'est un Alias que nous lui avons attribué). C'est ainsi que j'ai eu l'idée du nom Alias, ensuite en regardant Gloire & Fortune (honte sur moi), j'ai pu voir un épitéen (Mathieu BENARD) qui avait réaliser un TTS pour son projet de Spé : ALIAS le TTS était né, du moins le projet. Biensur, j'ai dû évaluer les compétences technico-psycho-pathologique de la team en faisant subir une série de "t'es pas cap"' aux membres...

1.1.3 Moyens techniques

Le matériel

	Mathieu	Yohan	Julie	Nicolas
Type	Desktop	Laptop	Desktop	Laptop
Processeur	Athlon XP 2000+	G4 1.5	Athlon XP 1800	Centrino 1.5
RAM	768Mo	1Go	512Mo	512Mo
OS	Fedora Core	Mac OS X	FreeBSD	Gentoo

CVS

CVS[2] est un outil Open Source qui permet à plusieurs personners de travailler simultanément sur un même ensemble de fichiers, sans craindre de perdre des données et en conservant un historique de toutes les modifications effectuées depuis la création des fichiers.

[2]Concurrent Version System

1.2 Présentation du projet

1.2.1 Définition du TTS

Un *Text To Speech* (ou *TTS*) est un type d'utilitaire basé sur la synthèse vocale permettant de lire un texte, provenant d'une source textuelle ou d'un flux continu de texte, le plus souvent avec la carte son d'un ordinateur, mais aussi pourquoi pas à l'aide d'un buzzer. Ces systèmes sont très utiles pour les personnes disposant d'handicaps, ou plus généralement pour communiquer avec les usagers d'un système informatisé d'une façon plus agréable.

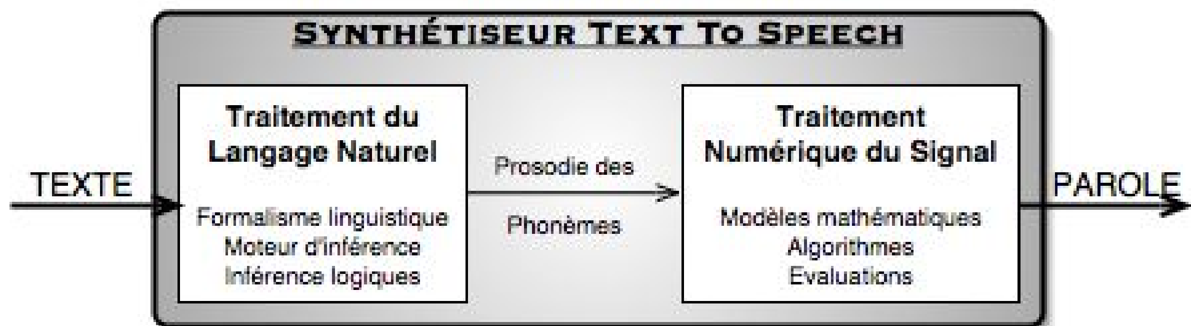


FIG. 1.1 – Diagramme de fonctionnement d'un TTS

1.2.2 Objectifs

Nous souhaitons réaliser un *Text To Speech* permettant de lire le texte entré à l'aide d'une interface graphique avec la carte son sur les différentes architectures sur lequel tournera notre projet. Il pourra aussi être utilisé en console directement. Une sauvegarde au format *wav* ou *ogg* de la sortie sonore sera aussi possible, pour une relecture ultérieure. Notre projet ne gérera par contre "que" la langue française.

1.2.3 Plateformes

Dans un souci de diffuser massivement notre logiciel, et de cette façon accomplir notre destinée (c'est à dire conquérir le monde), nous voulons faire un logiciel qui sera utilisable sous différentes plateformes. Ainsi, non seulement notre projet fonctionnera sous Unix, mais il devra aussi être utilisable sous Mac OS. Le fait que notre chef de projet ait en sa possession un Mac, n'est nullement entré en compte dans notre choix. Il va sans dire que cette décision nous a été spontanée, et que c'est de tout coeur que nous l'accomplirons.

1.2.4 Environnement

Pour mener à bien la réalisation de notre projet, nous aurons besoin d'utiliser différents environnements. Tout d'abord, pour l'interface, il a été décidé d'utiliser QT. De fait, nous allons nous servir de C++. D'autre part, pour mener à bien la création de notre parseur, nous nous sommes interrogés sur le meilleur choix de langage : Caml, C ou C++ ? En vue de notre future scolarité en cycle ingénierie, où la grande majorité des projets sont réalisés, d'une part en C, d'autre part en C++, nous avons donc décidé de favoriser le choix de ces deux langages algorithmiques. A noter que le C++, "comprend" le C.

Chapitre 2

Autotools[1]

2.1 Présentation

Autoconf, *Automake*, et *Libtool* sont des programmes qui augmentent la portabilité des logiciels et qui permettent de simplifier leur création. De nos jours la portabilité est un aspect très important des logiciels : en effet, si ces derniers ne fonctionnent que sur une seule plateforme, ils seront moins populaires et surtout, beaucoup moins utiles aux clients.

Autoconf est un outil qui rend les projets plus portables en exécutant des tests afin de trouver les caractéristiques du système avant que le projet soit compilé. Ainsi, le code source peut être adapté aux différents systèmes.

Automake est un outil pour la création des *Makefile*'s. *Automake* simplifie considérablement la phase de description de l'organisation du projet et fourni des fonctionalités supplémentaires aux *Makefile*'s comme la gestion de dépendances entre les fichiers sources.

Libtool est une interface en ligne de commande pour le compilateur et l'éditeur de liens qui simplifie la création de bibliothèques statiques et partagées —de manière portable—, indépendamment de la plateforme sur laquelle il fonctionne.

[1] *GNU Autoconf, Automake, and Libtool* éditions New Riders

2.2 Dépendances

Pour garantir le fonctionnement des *autotools*, nous devons leurs fournir deux fichiers obligatoires.

2.2.1 configure.in

Ce fichier contient tous les tests (sous la forme de macros) que le script *configure* exécutera. Quelques macros essentielles sont **AC_INIT** qui initialise *configure*, **AM_CONFIG_HEADER** qui génère le fichier *config.h* permettant ainsi d'effectuer des tests de portabilité dans le code source, et **AM_INIT_AUTOMAKE** utilisée pour initialiser *automake* demande deux arguments qui sont le nom du projet et sa version ; ceux-ci seront passés à *config.h* et seront disponibles sous la forme de deux constantes : *PACKAGE* et *VERSION*.
D'autres macros non moins importantes sont **AC_PROG_CC** (réciproquement **AC_PROG_CXX**) qui substituera la macro constante *CC* (réciproquement *CXX*) par le compilateur C (réciproquement C++) du système ; et **AC_OUTPUT** qui doit être appelée à la fin de *configure.in* pour créer tous les *Makefile*'s listés en argument.

Il est possible de passer un chemin préfixe où chercher *ALSA*, *Ogg* et *Vorbis*. La macro `AC_ARG_WITH` permet à *configure* de prendre en compte ce genre d'option :

```
$ ./configure --with-ogg-prefix=/sw
```

Dans ce cas, la variable `$withval`, créée par `AC_ARG_WITH` contiendra "/sw". Il est alors possible d'effectuer des tests de détections de la bibliothèque *Ogg* à partir de ce préfixe... Lorsqu'une bibliothèque est trouvée, une constante est définie dans le fichier *config.h* mais aussi de manière globale afin que sa portée atteigne aussi les *Makefile*'s. La macro `AC_DEFINE` permet de définir une constante dans *config.h*, et `AM_CONDITIONAL` permet de définir une constante (booléenne) pour les *Makefile.am*'s. Par exemple, si *ALSA* est détectée, on définit `HAVE_ALSA` via `AC_DEFINE` mais aussi par `AM_CONDITIONAL`.

2.2.2 Makefile.am

A la racine du projet, ce fichier doit contenir les sous-répertoires dans lesquels le programme *make* devra entrer. Dans les répertoires du projet contenant des fichiers sources, *Makefile.am* devra contenir les règles de construction de chaque source. Voici en exemple le fichier qui compile la *file* :

```
CFLAGS              = -Wall -W -ansi @ALIAS_INCLUDES@
LIBS                =

noinst_HEADERS      = queue.h

noinst_PROGRAMS     = test_queue
test_queue_SOURCES  = \
  queue.c             \
  test_queue.c
```

La constante **CFLAGS** contient les flags à passer au compilateur C.
@ALIAS_INCLUDES@ contient tous les répertoires du projet où le compilateur peut trouver des fichiers d'en-tête C (.h) ; elle a été définie dans *configure.in* pour être accessible à tous les *Makefile.am*.
_HEADERS précise qu'il s'agit d'un fichier source qu'il faudra inclure dans la distribution du projet (générée via *make dist*) cependant, le préfixe **noinst** précise que ce fichier doit être ignoré lors de l'installation avec *make install*.
_PROGRAMS est une autre règle *automake* définissant le nom de tous les exécutables, nom qui sert ensuite de label préfixe lors de l'appel de **_SOURCES** qui, comme son nom l'indique, définie les sources à utiliser pour créer la cible spécifiée en préfixe.

Voici un autre exemple illustrant la compilation conditionnelle de sources :

```
if HAVE_ALSA
libaudio_la_SOURCES         += alsa_support.c
LIBS                        += @ALSA_LIBS@
CFLAGS                      += @ALSA_CFLAGS@
endif
```

Cet exemple est un extrait du *Makefile.am* qui sert à compiler le module audio de notre projet. On remarque que si *ALSA* est présente, un fichier source sera rajouté à la bibliothèque partagée (générée par *libtool*), et les

macros `LIBS` et `CFLAGS` du compilateur seront modifiées en conséquences.

2.3 Code source portable

Comme nous venons de le voir, certaines constantes ont été définies ou non lors du *configure*. De ce fait, nous sommes capables de supprimer ou d'ajouter du code selon l'architecture ou les bibliothèques présentes.
`__APPLE__` nous permet de savoir si nous sommes sur un **MacOS**. Voici un autre exmple extrait de *decode.c* :

```
#ifdef HAVE_ALSA
# include <alsa/asoundlib.h>
# include "alsa_support.h"
#endif
```

Ceci permet d'inclure les fichiers d'en-têtes nécessaires au support *ALSA* seulement si la bibliothèque a été détectée avant la compilation.

2.4 Bootstrapping

Une fois tous les fichiers de dépendances créés, il faut exécuter les *autotools* dans un ordre précis. Cette phase peut-être automatisée en créant un fichier script *bootstrap*[2] appelant les *autotools* un à un.

2.4.1 aclocal

Le programme *aclocal* crée le fichier *aclocal.m4* en combinant les macros installées du système et celles de l'utilisateur pour définir toutes les macros demandées par *configure.in* dans un seul fichier.

2.4.2 libtoolize

Si `AC_PROG_LIBTOOL` est présente dans *configure.in*, *automake* appelle *libtoolize* pour générer les fichier *ltmain.sh* et *ltconfig* nécessaire à l'éditeur de liens.

[2] Littéralement : programme d'amorce

2.4.3 autoheader

autoheader exécute *m4*[3] sur *configure.in* pour générer des macros constantes (dans *config.h.in*) de la même manière qu'en langage C (ou C++).

2.4.4 automake

automake génère plusieurs fichiers nécessaires à *configure* comme les *Makefile.in*, et d'autres scripts pour l'installation du projet et l'automatisation de la reconnaissance du système (*config.guess*).

2.4.5 autoconf

autoconf développe les macros *m4* de *configure.in* en utilisant leurs définitions contenues dans *aclocal.m4*, afin de générer le script *configure*.

2.5 Utilisation

Après toutes ces phases et en supposant qu'elles se soient bien déroulées, il ne reste plus qu'à lancer le script *configure* qui ne fait que tester ce que nous lui demandons via *configure.in*. Enfin, si les tests ont été passés avec succès, le fichier d'en-tête *config.h* sera disponible et à l'image de *config.h.in*. Le programme *make* se chargera alors d'entrer dans tous les répertoires listés dans *Makefile.am* pour compiler le projet.

Les *Autotools* constituent un outils formidable et très pratique pour le portage de code mais ils restent néanmoins relativement difficiles à utiliser.

[3] GNU M4 est un outil à multiusage pour le traitement de texte

Chapitre 3

Dictionnaire

3.1 Présentation

La langue française contient de très nombreuses exceptions au niveau phonétique, si bien qu'il est impossible de créer une règle pour chacune d'entre elles. C'est pour cela que nous avons eu l'idée de créer un dictionnaire contenant chacune de ces exceptions. Biensur, ce dictionnaire ne contiendra jamais tous les mots de la langue française car nous avons pour but de créer un Text to Speech capable de reconnaître un maximum de mots avec des règles bien définies et pas un simple programme qui cherche un mot dans une base de données et en restitue le son. De plus, le dictionnaire nous permettra de traiter efficacement les acronymes ainsi que les signes mathématiques comme le +, le - ou encore le %. Au départ, il est certain que le dictionnaire ne contiendra que quelques entrées que nous lui aurons fournies, mais le but est qu'à chaque fois que l'utilisateur rencontre un mot mal prononcé par le Text to Speech, il puisse l'intégrer au dictionnaire : ceci garantit une évolution constante du programme au fil de son utilisation.

3.2 Représentation

3.2.1 A l'extérieur

Nous avons choisi une représentation physique très simple pour représenter le dictionnaire, je veux bien évidemment parler du fichier texte. En effet, c'est le type de fichier le plus simple à utiliser et le plus efficace pour stocker un nombre peu important de données au format texte. En effet, nous pensons qu'à terme le dictionnaire ne comportera pas plus de

500 entrées, ce qui est relativement faible et ne nécessite pas la mise en place de vraies bases de données qui sont généralement utilisées pour stocker plusieurs dizaines de milliers de données. La structure du fichier se décompose comme suit : tout d'abord le mot correctement orthographié (ALIAS ne gérera pas les fautes d'orthographe) suivit d'un espace puis de sa transcription phonétique et d'un retour chariot. Cette structure assez simple permet une édition à la main très aisée, pour peu que l'on connaisse les caractères de transcription phonétique utilisés par le programme pour restituer les sons.

3.2.2 A l'intérieur

Comme le dictionnaire n'est pas prévu pour contenir plus de 500 éléments, nous avons opté pour une représentation statique plutôt qu'une représentation dynamique. En effet, la représentation sous forme de tableau nous permet d'utiliser des méthodes de recherche et de tri (voir paragraphes suivants) très efficaces et simples à implémenter alors qu'une représentation sous forme d'arbre ou de liste chaînée nous aurait posé plus de problèmes. De plus, le nombre de mots contenus dans le dictionnaire restant relativement statique au cours de l'exécution du programme (sauf si l'utilisateur en rajoute pendant ce temps grâce à la fonction ajout), l'utilisation de structure dynamique n'aurait vraiment servi à rien. La structure du dictionnaire se décompose comme suit : un entier représentant le nombre de mots stockés dans le dictionnaire et un tableau de lexèmes dont les éléments comportent deux champs : le mot à proprement parlé et ses phonèmes associés.

3.3 Tri du dictionnaire

Comme son nom l'indique, un dictionnaire doit être trié pour pouvoir accélérer le temps de recherche pour un mot. Pour cela, plusieurs méthodes s'offraient à nous et nous en avons retenues deux pour leur rapidité d'exécution : la méthode du tri par insertion et celle du tri rapide, la méthode du tri à bulle étant complètement obsolète (étant de complexité n^2).

3.3.1 Tri par insertion

Le tri par insertion est l'un des algorithmes de tri les plus simples. Le tri par insertion prend les éléments un par un dans un ensemble non trié

et les insère dans un ensemble trié en parcourant celui ci afin de trouver où placer chaque nouvel élément. Bien qu'il puisse sembler que le tri par insertion ait besoin de deux emplacements distincts pour les ensembles non triés et triés, il trie en fait sur place, ce qui constitue un avantage certain en terme de place. Le tri par insertion est simple, mais il est inefficace pour les grands ensembles de données car la recherche de l'emplacement où placer chaque élément dans l'ensemble trié nécessite potentiellement de le comparer à tous les autres éléments déjà présents dans ce dernier. Un de ses avantages est que l'insertion d'un élément unique dans un ensemble déjà trié ne nécessite qu'un seul parcours de ses éléments, et non pas une exécution complète de l'algorithme, ce qui rend le tri par insertion particulièrement adapté au tri incrémental (décalage de données).

Le tri par insertion fonctionne essentiellement en insérant, les uns après les autres, les éléments d'un ensemble non trié dans un ensemble trié. Dans notre cas, les deux ensembles sont dans le dictionnaire, un unique bloc de mémoire contiguë. Au court de l'exécution, le dictionnaire est graduellement consommé par l'ensemble trié et, lorsque la fonction se termine, l'emplacement est complètement trié. Le tri par insertion consiste en une boucle imbriquée. La boucle extérieure, itérant sur j, contrôle l'élément de l'ensemble non trié qui sera inséré dans les éléments triés. Comme l'élément situé immédiatement à droite de l'ensemble trié est toujours le suivant à insérer, on peut également considérer que j est la position indiquant la frontière entre les éléments triés et non triés dans le dictionnaire. Pour chaque élément en position j, une boucle interne, itérant sur i, sert à parcourir à rebours l'ensemble des éléments triés jusqu'à trouver l'emplacement adéquat pour cet élément. Au fur et à mesure que l'on recule dans l'ensemble, chaque élément en position i est copié dans la position immédiatement à sa droite afin de laisser de la place pour l'insertion. Lorsque j a atteint la fin de l'ensemble non trié, le dictionnaire est trié !

La complexité d'exécution du tri par insertion ne dépend que de ses boucles imbriquées. Sachant cela, la boucle externe s'exécute en $T(n) = n-1$, fois un certain temps, où n est le nombre d'éléments à trier. En considérant la boucle interne dans le pire des cas, on suppose qu'il faudra aller complètement à gauche pour insérer l'élément dans l'ensemble trié : elle pourrait donc itérer une fois pour le premier élément, deux fois pour le deuxième, etc… jusqu'à la fin de la boucle externe. Le temps d'exécution de la boucle imbriquée est donc représenté comme une sommation de 1 à $n-1$, ce qui donne un temps $T(n) = (n(n+1)/2) - n$, fois un certain temps ce qui donne une complexité en n^2, pour tomber à n losque l'on utilise ce

tri dans un tri incrémental. Le tri par insertion trie sur place et ses besoins en espace mémoire sont donc ceux des données à trier.

Malgré tous ses avantages, le tri par insertion a, dans le pire des cas, un temps d'exécution supérieur à celui du tri rapide. C'est pourquoi nous avons choisi d'utiliser ce dernier.

3.3.2 Tri rapide

Le tri rapide est un algorithme de tri de type diviser pour régner. Il est largement considéré comme le meilleur dans le cas général. Comme le tri par insertion, il s'agit d'un tri par comparaison et sur place, mais son efficacité en fait le meilleur choix pour les ensembles moyens à grands. Si l'on prend l'exemple du tri manuel d'une pile de chèques, on commence avec une pile non triée que l'on divise en deux. Sur une pile, on place tous les chèques de numéro inférieurs ou égaux à ce que l'on pense être la valeur médiane, sur l'autre on place les chèque de numéro supérieur à cette valeur. Lorsque ces deux piles sont constituées, on divise chacune d'elles de la même manière et on répète ce traitement jusqu'à obtenir des piles ne contenant qu'un seul chèque. En réempilant ces chèques, on obtient alors une pile de chèques triés.

Etant donné sa popularité, il est surprenant que le pire des cas pour le tri rapide ne soit pas meilleur que le pire des cas pour le tri par insertion. Cependant, on peut rendre cette situation suffisamment rare et ainsi considérer que l'algorithme s'exécutera dans un cas moyen, qui est considérablement meilleur. La clé de tout cela consiste à choisir correctement la valeur de partitionnement dans l'étape de division.

Le tri rapide a de mauvaises performances lorsque l'on choisit des valeurs de partitionnement qui forcent toujours la majorité des éléments à aller dans une seule partition. On doit plutôt répartir les éléments de façon aussi équilibrée que possible. Une approche fonctionnant bien pour le choix de partition consiste à les choisir au hasard. Statistiquement, cela permet d'éviter à un ensemble de données de mal se comporter, même si l'on voulait enliser l'algorithme de façon délibérée. Le partitionnement peut encore être amélioré en choisissant trois éléments au hasard et en choisissant leur médiane comme valeur de partitionnement : c'est la méthode par la médiane des trois qui garantit quasiment un cas d'exécution moyen. Comme cette approche du partitionnement s'appuie sur les

propriétés statistiques des nombres aléatoires pour améliorer les performances globales du tri rapide, ce dernier est un bon exemple d'algorithme probabiliste.

Le tri rapide fonctionne essentiellement en partitionnant récursivement un ensemble non trié d'éléments jusqu'à ce que toutes les partitions ne contiennent qu'un seul élément. Dans l'implémentation choisie ici, le dictionnaire contient au départ l'ensemble non trié de nb_elts éléments, stocké dans un unique bloc de mémoire contiguë. Le tri rapide trie sur place, tout le partitionnent est donc également réalisé dans le dictionnaire. Lorsque la fonction se termine, le dictionnaire est entièrement trié !

Comme nous l'avons vu, une partie importante du tri rapide consiste à partitionner des données : cette tache est réalisée par une autre fonction. Cette dernière répartie les éléments entre les positions i et k dans le dictionnaire, où i est inférieur à k.

On commence par choisir une valeur de partitionnement à l'aide de la méthode par la médiane des trois. Lorsque cette valeur a été choisie, on va de k vers la gauche dans le dictionnaire jusqu'à trouver un élément qui lui est inférieur ou égal : celui ci appartient à la partition gauche. Puis, on va de i vers la droite jusqu'à trouver un élément supérieur ou égal à la valeur de partitionnement : il appartient à la partition droite. Lorsque l'on a trouvé deux éléments dans la mauvaise partition, on les échange. On continue jusqu'à ce que i et k se croisent. Lorsque ce croisement a eu lieu, tous les éléments à gauche de la valeur de partitionnement lui sont inférieurs ou égaux, et tous ceux à droite lui sont supérieurs. Lors du premier appel, i est initialisé à 0 et k à $nb_elts - 1$.

L'analyse du tri rapide est centrée sur son comportement dans le cas moyen, communément considéré comme son unité de mesure. Bien que le pire des cas ne soit pas meilleur que celui du tri par insertion (n^2), le tri rapide s'exécute en un temps plus proche de celui de son cas moyen ($n \times log(n)$) où n est le nombre d'éléments à trier. Le calcul de la complexité d'exécution du tri rapide dans le cas moyen dépend du fait qu'il y aura une distribution équitable de partitions équilibrées et déséquilibrées. Cette supposition est raisonnable si l'on utilise la méthode de partitionnement par la médiane des trois. Cet algorithme trie sur place, ses besoins en mémoire sont ceux des données à trier.

3.4 Utilisation

La réalisation du dictionnaire nous a donc permis de réaliser le lexer, c'est à dire un programme qui détecte si un lexème appartient au dictionnaire (c'est donc une exception) et retourne sa liste de phonèmes associés, sinon, qui traite le lexème avec les règles de phonétique standard (voir implémentation du lexer). Le dictionnaire étant trié et statique, nous pouvons donc utiliser la méthode de la dichotomie pour chercher si un lexème est présent ou non dans la liste avec un temps de recherche très court.

La recherche dichotomique commence par un ensemble de données trié. Pour débuter la recherche, on compare l'élément central de cet ensemble à l'élément recherché : si le premier est plus grand que le second, on prend la moitié inférieure de l'ensemble comme nouvel ensemble de recherche. Sinon, on prend la moitié supérieure de l'ensemble. Ce traitement se répète sur chaque ensemble, de plus en plus réduit, jusqu'à trouver l'élément recherché ou ne plus pouvoir diviser l'ensemble. Cette recherche fonctionne pour tous les types de données pour lesquels on peut établir un ordre entre les éléments.

La recherche dichotomique fonctionne essentiellement par division successive d'un ensemble de données trié et par comparaison de l'élément central de chaque division. Nous avons utilisé la même méthode d'implémentation que celle vue en cours.

La complexité d'exécution de la recherche dichotomique dépend du nombre maximum de divisions possibles pendant le traitement : pour un ensemble de n éléments, on peut réaliser jusqu'à $log(n)$ divisions. Dans le cas de la recherche dichotomique, cette valeur représente le nombre de comparaisons qu'il faudra réaliser dans le pire des cas : lorsque la cible est absente par exemple. Par conséquent, la complexité d'exécution de la recherche binaire est en $log(n)$.

Chapitre 4

Traitement du Langage Naturel

Un *Text To Speech* comporte une phase essentielle appelée le **Traitement du Langage Naturel**[1] qui permet de générer la suite de *phonèmes* correspondant au texte passé au module TLN. Celui-ci est composé d'un *analyseur syntaxique* et d'un *générateur de phonèmes*. Mais avant de continuer, il faut connaître la phonétique[2].

4.1 Phonétique

En français, il a **26 lettres**, **37 *phonèmes*** et plus de **130 graphèmes** différents. Un *phonème* correspond à une unité de son et un graphème est une occurrence graphique de ce son.
Les tableaux suivants présentent les différents *phonèmes*.

[1]TLN
[2]http ://phonetique.free.fr/

	Mot-clé	Autres graphèmes
[i]	lit	stylo, île, maïs, meeting
[é]	télé	Parler, nez, pied, messieurs, poignée, volontiers
[è]	règle	chienne, merci, jouet, mais, maître, payer, treize, être, Noël, volley
[a]	sac	à, femme
[u]	lune	sûr, eu (avoir au passé composé)
[e]	je	
[E]	feu	nœud, jeûne
[F]	fleur	cœur, club

		Mot-clé	Autres graphèmes
Labio-dentales	[f]	flûte	phare, affaire
	[v]	valise	wagon
Dentales	[s]	soleil	poisson, citron, garçon, démocratie, science, asthme, six
	[z]	maison	zoo, deuxième, blizzard
Palatales	[H]	chat	short, schéma, fasciste
	[j]	jupe	girafe
[l]		lampe	elle
[R]		roue	beurre, rhume

		Mot-clé	Autres graphèmes
	[U]	poule	où, goûter, football, août
	[o]	vélo	landau, bateau, drôle,
	[O]	pomme	album, alcool, capharnaüm
	[A]	pâte	
	[D]	un	parfum
	[C]	lapin	chien, pain, peinture, daim, imparfait, syndicat, sympa
	[B]	gant	dent, jambe, empereur, paon, Caen
	[~]	ballon	ombre, punch
	[J]	pied	crayon, soleil, paille, hyène, païen
	[V]	huit	sueur, suave, ennuyeux
	[W]	doigt	ouate, wallon, équateur, moelle, poêle, croît, asseoir (+ nasale : loin)
Bilabiales	[p]	pile	appartement
	[b]	bol	abbaye
	[m]	mur	flamme
Dentales	[t]	table	datte
	[d]	dé	addition
	[n]	noeud	anniversaire
	[G]	ligne	manière
Vélaires	[k]	cadeau	qualité, képi, accord, orchestre, ticket, coq
	[g]	gâteau	bague, aggraver, second, ghetto
	[N]	parking	

4.2 Analyseur syntaxique

L'analyse se fait en trois temps : la vérification de la chaîne, la détection de lexème(s), et enfin la conversion en phonèmes.

4.2.1 Vérification

La vérification est effectuée de la manière suivante :

- mettre la chaîne de caractères en minuscules.
- déterminer si les symboles spéciaux de la langue (ponctuation, guillemets) ne se répètent pas de manière consécutive.
- vérifier qu'il n'y a pas de symboles "baroques" (© Olivier RODOT) comme '=', etc... Nous ne gérons pas non plus les parenthèses.

Prenons par exemple la chaîne "Bonjour,, tout, le Monde !". Une fois passée à la fonction `lowerify` nous obtenons la chaîne "bonjour,, tout, le monde !". Puis la fonction `is_valid` appliquée à cette dernière chaîne déclenchera une erreur indiquant que le symbole ',' n'est pas valide puisqu'il n'est pas autorisé à être présent deux fois consécutives.

4.2.2 Détection

La langue française possède de trop nombreuses exceptions qui rendent le parser beaucoup trop complexe à réaliser, c'est pourquoi nous avons eu l'idée d'implémenter un dictionnaire. Nous appelons *exception* le fait de tester la lettre suivante lorsque la lettre courante n'a pas de phonème associé directement. Par exemple, le 'v' ne déclenche aucune *exception*, ce qui n'est pas le cas du 'o', qui lui en déclenche au moins une rien que pour savoir s'il est suivi d'un 'n'...

On ne peut évidemment pas gérer tous les cas du français, l'algorithme serait immédiatement alourdi pour quelques cas particuliers (moins de mille). Avec notre algorithme d'analyse dépourvu du dictionnaire, le mot "cation" est prononcé [k a s i ~]. Pour forcer la bonne génération de phonème (qui est [k a t i ~]) il aurait fallu gérer ce cas à partir de la lettre 'c' ce qui aurait déclenché cinq *exceptions* : ce n'était pas envisageable !

Notre méthode consiste à considérer comme cas particulier tout mot qui déclenchera **plus de trois *exceptions***. Ainsi, il nous suffit d'ajouter au

dictionnaire tous les mots qui nous posent ce genre de problème.

Il s'agit alors pour chaque mot de savoir s'il est un *lexème* ou non. Un *lexème* est un mot appartenant au *dictionnaire* ; ainsi, à chaque *lexème* est associé une suite de *phonèmes*. Mais tout ceci a déjà été expliqué dans la section concernant le *dictionnaire*.

4.2.3 Conversion

Cette partie du TLN a été codée en deux temps : de manière intuitive puis logiquement. Mais quelque soit la manière de coder le générateur, nous avons dû implémenter une *file* pour stocker les *phonèmes* dans leur ordre de création.

Étude intuitive (ou naïve)

Cette étude se résume à un simple parcours de chaîne pendant lequel un *phonème* est associé (puis enfilé) à une séquence de deux symboles. Voici un exemple de génération de *phonèmes* avec la chaîne "Bonjour" :

- la chaîne est "lowerifiée" et devient "bonjour".
- 'b' n'est pas suivi d'un autre 'b' donc on avance de 1 et on enfile [b].
- 'o' est suivi de 'n' donc on avance de 2 (car 'n' vient de servir) et on enfile [~].
- on avance de 1 et on enfile [j].
- 'o' est suivi de 'u' donc on avance de 2 (car 'u' vient de servir) et on enfile [U].
- enfin, pour 'r' qui est seul dans son coin, on avance de 1 et on enfile [R].

Finalement, la séquence obtenue est [b ~ j U R]. C'est donc, à priori, une méthode efficace. Mais si on considère cette nouvelle chaîne, qui, une fois "lowerifiée", devient "bonne année", on se rend compte que la génération des *phonèmes* n'est pas correcte :

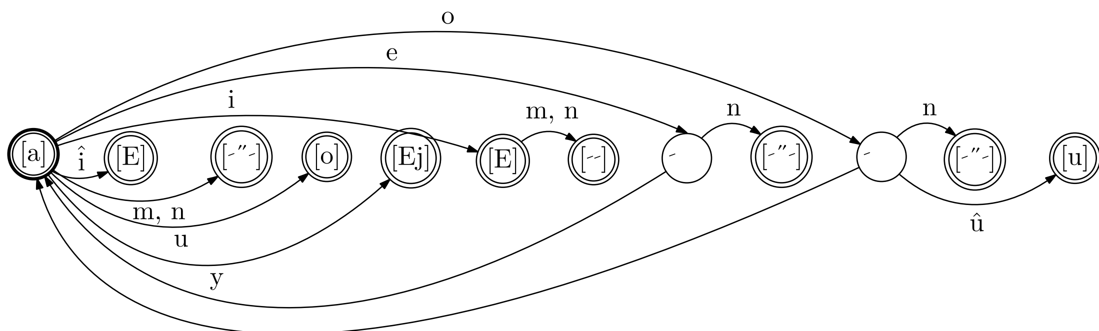
- 'b' n'est pas suivi d'un autre 'b' donc on avance de 1 et on enfile [b].
- 'o' est suivi de 'n' donc on avance de 2 (car 'n' vient de servir) et on enfile [~].
- 'n' est suivi de 'e', on avance de 2 est on enfile [n e].
- 'a' est suivi de 'n', on avance de 2, on enfile [B].

- 'n' est suivi d'un symbole non présent dans l'alphabet, on avance de 1, on enfile [n].
- 'é' est un symbole spécial, on avance de 1, on enfile [é].
- enfin, on avance de 1, on enfile [e].

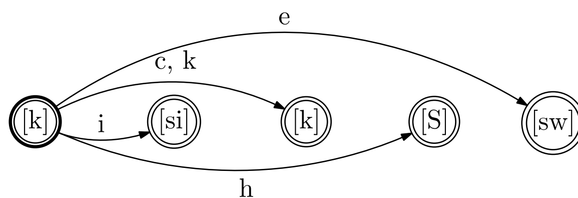
La séquence ainsi générée est [b ~ n e B n é e] ce qui serait prononcé "bon ne an né e". C'est très loin d'être [b O n a n é].

Étude logique

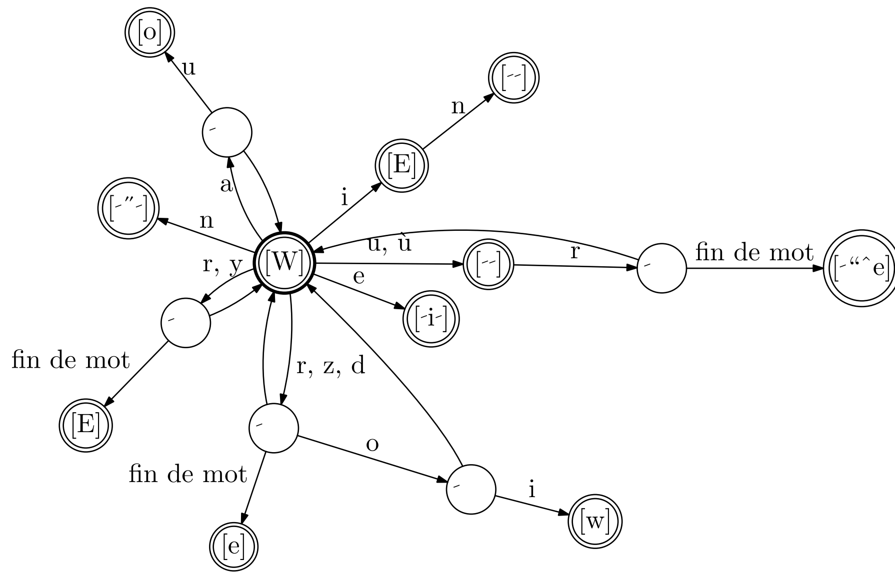
Nous venons de voir une méthode très limitée, celle-ci utilise un outil informatique : les automates. Nous avons dû construire un automate (pas nécessairement déterministe) pour chaque lettre. Voici les principaux automates :



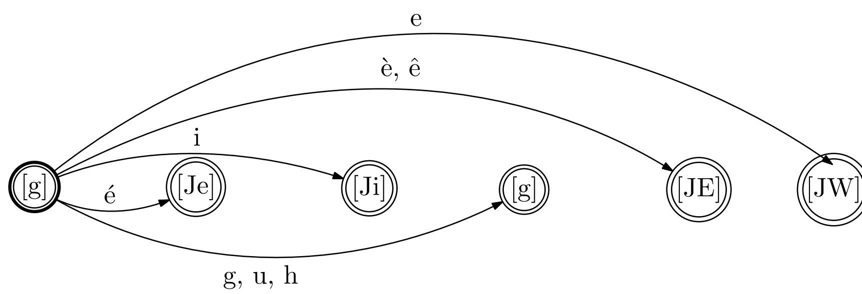
Automate du 'a'



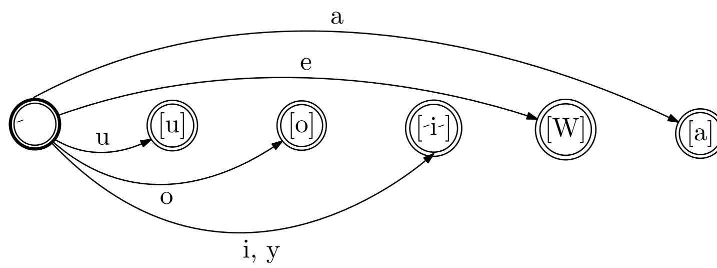
Automate du 'c'



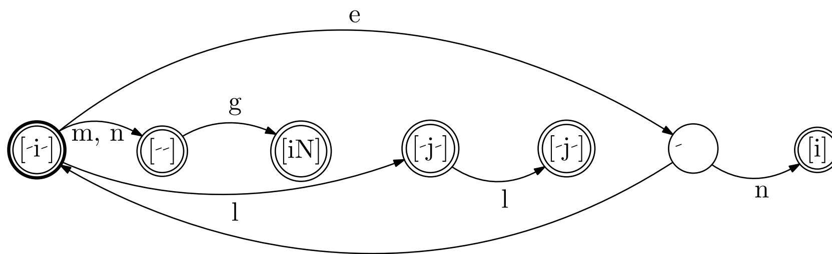
Automate du ’e’



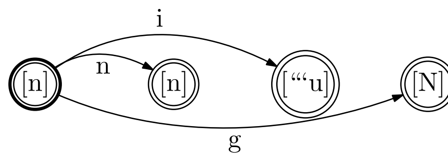
Automate du ’g’



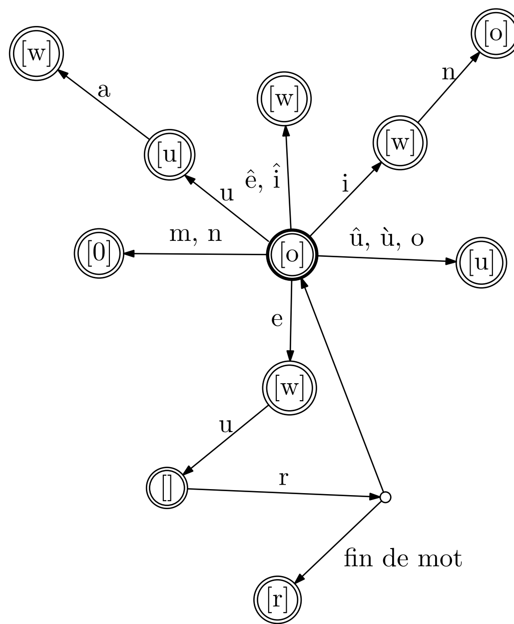
Automate du ’h’



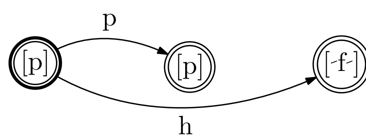
Automate du 'i'



Automate du 'n'



Automate du 'o'



Automate du 'p'

Reprenons l'exemple précédent ("bonne année") pour illustrer la capacité d'un automate à gérer toutes les situations qui ne dépendent pas de la nature des mots :

- 'b' n'est pas suivi d'un autre 'b' donc on avance de 1 et on enfile [b].
- 'o' est suivi de 'n' **mais** 'n' est suivi d'un autre 'n' donc on avance de 1 et on enfile [O] ;
- 'n' est suivi de 'n', on avance de 2 est on enfile [n].
- 'e' est suivi du caractère spécial espace, il n'est donc pas prononçable, on avance de 1.
- 'a' est suivi de 'n' **mais** 'n' est suivi d'un autre 'n' : on avance de 1, on enfile [a].
- 'n' est suivi de 'n', on avance de 2 est on enfile [n].
- 'é' est un symbole spécial, on avance de 1, on enfile [é].
- enfin, 'e' est suivi du caractère spécial '\0', on avance de 1.

On obtient au final la séquence de phonèmes [b O n   a n é].

Voici maintenant ce qui se passe en apportant l'amélioration des *liaisons* :

- 'o' est suivi de 'n', et 'n' est suivi d'un caractère spécial (espace), on avance de 2 et on enfile [~]. **De plus**, 'n' est suivi d'un autre mot qui commence par une voyelle donc on enfile [n].
- 'a' est suivi d'un espace, on avance de 1 est on enfile [a].
- 'e' est suivi d'un 'n' qui est lui-même suivi d'une consonne, on avance de 2 et on enfile [B].
- 'v' a son propre phonème, on avance de 1, on enfile [v].
- 'i' est suivi d'un 'e', aucune exception trouvée, on avance de 1 et on enfile [i].
- 'e' est suivi d'un espace, il n'est donc pas prononçable, on avance de 1.
- 'd' a son propre phonème, on avance de 1, on enfile [d].
- 'o' est suivi d'un 'r', aucune exception trouvée, on avance de 1, on enfile [o].
- 'r' est suivi d'un 'm', aucune exception trouvée, on avance de 1, on enfile [R].
- 'm' est suivi d'un 'i', aucune exception trouvée, on avance de 1, on enfile [m].
- 'i' est suivi d'un 'r', aucune exception trouvée, on avance de 1, on enfile [i].

- 'r' est suivi de la fin de la chaîne, on avance de 1, on enfile [o].

On obtient au final la séquence de phonèmes
[~n a Bvi de doRmiR].

L'analyseur syntaxique est aussi capable de convertir les nombres...

4.3 Des chiffres et des lettres

Nous venons de présenter un *dictionnaire* capable de gérer pour nous les mots déclenchant de trop nombreuses exceptions. Ainsi, nous avons pu montrer en exemple la décomposition en *phonèmes* du mot *cation* et du chiffre 2. Mais, les combinaisons des **10 chiffres de base** étant infinies et le *dictionnaire* étant par définition un ensemble fini, nous avons implémenté un nouvel algorithme dédié au traitement des nombres.

Cet algorithme nécessite l'usage de quelques fonctions préliminaires dont la détermination de la longueur du nombre —1234 a une longueur de 4— qui permet en même temps de savoir si le pointeur (de la chaîne de caractères) pointe sur un nombre, car dans le cas contraire, notre fonction retourne 0, ce qui a pour effet de prévenir le *parser* principal que c'est à lui d'agir.

La fonction principale `parse_number` utilise la fonction `is_number` pour déterminer si le prochain caractère est le début d'une suite de chiffres. `is_number` parcourt la chaîne tant que le caractère courant est un chiffre (donc un code ASCII[3] compris entre celui de '0' et celui de '9') puis retourne la longueur du nombre en terme de chiffres. Cette longueur est très importante pour choisir quel coefficient doit être attribué au paquet que nous allons traiter. Par exemple, le nombre 626921023 se découpe par paquets de trois nombres : soient 626, 921, et 023, auxquels on affecte les coefficients respectifs `MILLION`, `MILLE`, et `NO`. Ce sont des macro-constantes représentant les *phonèmes* des mots million et mille ; `NO` est juste une redéfinition de `NULL`...

Finalement, la phase de traduction s'effectue via deux fonctions et plus de cent macro-constantes représentant la suite de phonèmes associée, entre

[3] American Standard Code for Information Interchange

autres, aux nombres de 0 à 100.

4.3.1 Traduction d'un paquet

Cette tâche est une phase essentielle à la décomposition d'un nombre en sa suite de *phonèmes* associés. La fonction `process_number` va effectuer une série d'opérations mathématiques sur la *longueur* du nombre afin de connaître l'aspect du paquet en cours de traitement. Voici un petit exemple avec le nombre (paquet) 627 provenant de 627921023 :

- `len`[4] est divisible par 3 (c'est-à-dire, que le reste de la division entière de `len` par 3 est égal à 0), donc 6 doit être considéré comme l'unité de centaine, ainsi, nous devrons enfiler les symboles de `SIX` suivis de ceux de `CENT` à ceci près que 6 génère une exception face à un coefficient car nous ne disons pas [s i s s B] mais [s i   s B]. Le premier chiffre venant d'être traité, `len` est décrémentée.

- la gestion des dizaines n'induit pas de difficultés supplémentaires puisque celles-ci sont déjà présentes, nous le rappelons, sous la forme de macro-constantes. De plus, un tableau de chaînes global contient toutes ces constantes dans leur ordre croissant ; de ce fait, il est facile d'accéder aux symboles du nombre des dizaines :

  ```
  gl_numbers[10*(numb[k]-'0')+numb[k+1]-'0']
  ```

  où `gl_numbers` est le tableau de chaînes, `numb`, la chaîne pointant sur une représentation du nombre et `k`, la position dans cette chaîne. `len` est décrémentée de 2. Il est utile pour la suite de préciser que `len` est passée par référence à la fonction `process_number`.

- il ne reste plus qu'à enfiler les *phonèmes* du coefficient (s'il y en a un). La manière de déterminer s'il s'agit de `MILLIARD`, `MILLION`, `MILLE`, ou `NO` relève de la fonction principale d'analyse du nombre (`parse_number`).

[4] len = la longueur du nombre à parser

4.3.2 Diviser pour régner

Il faut établir une règle de base (un cas d'arrêt) à notre algorithme de traitement des nombres. Cette règle se charge de traduire les nombres de 0 à 999999999999. Grâce à cette règle nous pouvons ensuite gérer l'infini[5] (évidemment limité par la représentation des entiers `long` —et même `long long`— en mémoire). Dans les deux cas, l'algorithme boucle tant que `len` est strictement positive.

Domaine fini : moins de 12 chiffres

Plusieurs cas se présentent à nous :

- `len` est strictement supérieure à 9 et inférieure à 12 : il s'agit d'un paquet appartenant aux milliards.
- `len` est strictement supérieure à 6 et inférieure à 9 : il s'agit d'un paquet appartenant aux millions.
- `len` est strictement supérieure à 3 et inférieure à 6 : il s'agit d'un paquet appartenant aux milliers.
- `len` est inférieure à 3 : il s'agit d'un paquet non coefficienté (d'où l'usage de `NO`).

Domaine infini : plus de 12 chiffres

Le plus dur est de déterminer dans quel domaine (`MILLIARD`,`MILLION`, etc.) le prochain paquet va se trouver. Encore une fois, une série (un peu plus importante) de tests est nécessaire :

- `len` est congrue à 3 modulo 9 : il s'agit d'un paquet appartenant aux milliards.
- `len` est divisible par 9 : il s'agit d'un paquet appartenant aux millions.
- `len` est congrue à 6 modulo 9 : il s'agit d'un paquet appartenant aux milliers.

[5]De nos jours l'infini a une limite assez lointaine ©Krisboul

Par ailleurs, on effectue deux autres tests similaires à chaque expression conditionnelle, avec les changements de variables `len1` = `len` + 1 et `len2` = `len` + 2.

Lorsque le nombre a été converti dans sa suite de *phonèmes*, on déplace (incrémente) l'index (de la chaîne principale) `i` de la longueur `len` afin de permettre au *parser* de continuer sa tâche…

Chapitre 5

Partie audio

5.1 Choix du format audio

5.1.1 PCM

Les cartes sons utilisent le format PCM (*Pulse Code Modulation*) en interne, mais plusieurs variantes existent. Les données peuvent être alignées en *Little Endian*, ou en *Big Endian*, et avec une fréquence d'échantillonage et un nombre de canaux variables. Ce format a l'avantage d'être celui de la carte, mais il est très gourmant en espace disque. Nous avons donc dû choisir un format de compression.
Nos échantillons seront très courts, contenant juste des *phonèmes*, les différences de tailles ne seront donc pas significatives d'un format à l'autre.

5.1.2 MP3

Le premier format de compression qui nous est venu à l'esprit est le très répandu MP3 (*MPEG-1/2 Audio Layer 3*). C'est un format de compression avec pertes réduisant la taille d'un son en moyenne de 12 fois. Il supprime les fréquences inaudibles pour l'oreille humaine, ainsi que les sons très faibles, tout en maintenant une qualité d'écoute très bonne. Ce format a été étudié pour un débit binaire (*bitrate*) fixe, cependant, il peut-être utilisé avec un débit binaire variable. Toutefois, ce format est soumis aux brevets logiciels dans certains pays, et pour un projet qui pourrait être commercialisé, cela peut-être génant.

5.1.3 Ogg Vorbis

Tout comme le MP3, le Ogg Vorbis est un format de compression avec perte. Cependant, il a été conçu pour un débit binaire variable, permettant une qualité sonore constante, et non un débit constant. De plus, ce format utilise des API libres (licence BSD), et très bien documentées. Il semble donc corriger les légers défauts du MP3.

5.1.4 Speex

Ce format est diffusé par Xiph.org, la même fondation qui fournit l'Ogg Vorbis. Il est conçu pour compresser de la voix, cependant il s'agit d'un algorithme de compression avec perte, et n'a sa place que dans des contextes de diffusion par réseau (*Streaming*). Il ne semble donc pas nous convenir.

5.1.5 Notre Choix

Nous nous sommes donc naturellement tournés vers le format Ogg Vorbis.

5.2 Principe

5.2.1 Différents Threads

Nous utilisons une Thread différente pour la lecture audio. Ceci permet de ne pas bloquer le reste du programme pendant la lecture, qui peut être très longue (la durée totale de tous les *phonèmes* mis bout-à-bout).

5.2.2 Contrôle de la Thread audio et passage de sons

Une fois la Thread audio lancée avec en paramètre le type de sortie désiré, il faut pouvoir lui communiquer les sons à lire, ainsi que pouvoir l'arrêter, ou même changer de matériel audio. Ceci a été réalisé à l'aide des *Mutex* et des variables conditionnnelles.
La Thread attend un signal venant de la variable conditionnnelle. Puis vérifie des booléens pour savoir si le matériel doit changer, ou s'il faut quitter. Ensuite elle vérifie l'état d'une file qui doit contenir les sons à lire, et les lire si elle n'est pas vide.
Sur la Thread principale du programme, il suffit de changer la valeur des

booléens, puis d'envoyer un signal pour contrôler la Thread audio, ou alors d'enfiler des sons, et d'envoyer le signal pour qu'ils soient lus.

5.3 Implémentation

5.3.1 Ogg Vorbis

Nous utilisons bien évidement l'API Ogg Vorbis venant de Xiph.org pour la partie décompression. Il s'agit d'ouvrir un fichier Ogg, et de le parcourir pour obtenir les données PCM correspondantes.

5.3.2 xBSD et Linux 2.4 : OSS

Les systèmes BSD et Linux (jusqu'au 2.4 officielement) utilisent le système OSS (*Open Sound System*) pour lire des sons. En fait, il s'agit d'ouvrir le périphérique associé (le plus souvent */dev/snd*) et d'y écrire les données PCM, après avoir pris le soin de l'initialiser et de le paramétrer avec *ioctl*.

5.3.3 Linux 2.6 : ALSA

Depuis la version 2.5 de linux, et donc la version stable 2.6, le système utilise ALSA (*Advanced Linux Sound Architecture*) pour lire un son. Malgré que la compatibilité OSS soit toujours proposée, il n'y a aucune obligation qu'elle soit présente sur une machine. Pour garantir une installation aisée, il nous est paru plus utile d'intégrer son support dans notre projet.
Pour utiliser ALSA, nous devons comme pour OSS initialiser le matériel, qui n'est plus identifié par un fichier, mais par une référence dans le système, comme par exemple *hw :0,0*. Cette fois ci, l'envoi de données se fait d'une manière asynchrone : nous intérrogeons ALSA pour savoir la taille disponible dans son buffer, et nous demandons à l'API Ogg Vorbis de nous décompresser cette taille, pour y introduire les données PCM. Cette méthode permet de décompresser et de lire à la volée, ce qui doit être lu, et pas plus.

5.3.4 Mac OS X

Le portage sous **MacOS X** n'a finalement pas été fait en utilisant *CoreAudio* à cause d'un manque de documentation. Nous avons choisi la bibliothèque *SDL* car elle nous paraîssait la plus portable. De plus, elle est sous license GPL.

5.4 Vitesse et tonalité

La lecture de phonèmes ne correspond pas seulement à enchaîner les fichiers sonores. Il nous est indispensable à terme de pouvoir changer la vitesse de lecture ainsi que la tonalité afin de rendre la lecture plus réaliste. Nous allons donc ici vous présenter les méthodes sélectionnées déjà implémentés dans la partie audio pour des évolutions futures.

5.4.1 Le format PCM

Le format PCM ou *Pulse Code Modulation* contient les données brutes à envoyer à la carte son. comme nous travaillons sur de la voix, nous allons uniquement nous intéresser aux enregistrements *mono*. Les enregistrements PCM contiennent les variations de pressions enregistrées par le microphone lors de la capture, ce qui est suffisant pour que l'oreille humaine n'entende pas de différences majeures avec le son original. Les données sont capturées à une fréquence donnée, sur un nombre de *bits* donnés. La carte son lors de la relecture tire à profit le caractère continu du signal pour tenter de le restituer.

5.4.2 Méthode de base : vitesse et tonalité

La première méthode qui vient à l'esprit pour changer la vitesse de lecture est de modifier la taille du *buffer* en allongeant ou en réduisant les signaux. Par exemple, si on recopie chaque "case" sur deux "cases", nous obtenons un enregistrement deux fois plus long, la vitesse est donc divisée par deux. De la même manière, si nous effaçons une "case" sur deux, nous obtenons un enregistrement deux fois plus court.
Cette méthode marche bien pour modifier la vitesse de lecture, mais comme nous ne modifions pas le signal, les fréquences sont aussi modifiées avec le changement de durée. Ainsi, si nous multiplions par deux la longeur, nous divisons par deux la tonalité. Cette méthode n'est donc pas très adaptée pour améliorer le naturel du rendu.

5.4.3 Seconde méthode : vitesse

Nous avons vu précédement que si nous modifions la longueur de l'enregistrement, nous modifions également les fréquences et donc la tonalité. Par contre, si nous recopions ou sautons des portions entières de l'enregistrement, nous allons modifier la durée de l'enregistrement, mais nous

n'allons pas modifier les fréquences. La taille de nos *buffers* étant négligeable comparativement à la taille des enregistrements, si nous en enlevons quelque-uns, le son reste toujours audible et compréhensible. Nous obtenons donc des enregistrements dont la vitesse est modifiée, sans changer le contenu du son, le ton de la voix restant le même.

5.4.4 Synthèse

Pour modifier la tonalité, nous allons donc utiliser la première méthode, et ensuite ramener la durée de l'enregistrement à celle désirée à l'aide de la seconde méthode. Par exemple si nous multiplions la tonalité par deux, il faudra ensuite multiplier la durée par deux également. Nous modifions donc notre *buffer* à l'aide d'une unique fonction qui va modifier à la fois la vitesse et la tonalité.

5.5 Correspondance phonème et nom de fichier

Le *parser* utilise une file de *phonèmes* en sortie, et nous utilisons pour la sortie audio, une file de noms de fichiers. Il a donc été nécessaire d'utiliser une fonction de conversion qui associe une liste de *phonèmes* à une liste de fichiers. Le programme lors de son chargement détecte les fichiers sons grâce à un fichier de configuration optionnel, ainsi qu'avec les variables fournies par les autotools lors de la précompilation.

5.6 Inclusion du son dans l'interface

Le bouton "lecture" et les possibilités de sauvegardes ont étés reliés à la partie audio, et, la sortie audio est initialisée avec le lancement du programme. Comme cette liaison a été prévue depuis le début du développement, elle s'est effectuée sans difficulté majeure, et nous pouvons maintenant entendre les sorties du *parser*, pourvu qu'elles n'utilisent que des *phonèmes* implémentés (enregistrés).

5.7 Utilisation des Autotools

Le Code a été intégralement revu, dans le but de faciliter le portage. Ainsi, nous utilisons les autotools pour utiliser ou non, les sorties *ALSA* et *OSS*. Ce travail a été difficile, car les deux bibliothèques n'utilisent pas les mêmes unités pour leur sorties. Ainsi *ALSA* compte en *frames*, tandis que *OSS* compte en *octets*. Nous avons aussi rajouté une sortie nulle (ou sortie *virtuelle*), permettant d'utiliser le projet sur des configurations sans sorties sonores, ou de permettre de lancer le projet sur des architectures non supportées. Il permet aussi d'accélerer le traitement en vue de conversions en masse, car nous ne devons pas attendre la carte son dans ce cas là.

5.8 Exportation vers les différents formats

Notre *Text-to-Speech* a de nombreuses utilités, mais une fonctionnalité souvent recherchée par les utilisateurs est la possibilité de sauvegarder la sortie, dans le but de la réutiliser. C'est pourquoi nous avons implémenté, les sorties en format *PCM*, *WAVE*, et *Ogg/Vorbis*.

5.8.1 Le format *PCM* brut : le *RAW*

Le format *PCM* (*Pulse Code Modulation*), est le format que nous utilisons depuis le début en interne. Il peut donc nous être utile de l'exporter à des fins de débogage. De plus, comme tous les programmes travaillant le son utilisent ce format en interne, ou presque, il peut être intéressant pour l'utilisateur final de disposer de fichiers en *RAW* pour les inclure dans des programmes simplistes. Nos fichiers *RAW* utilisent le format interne à notre programme, à savoir des données 16 bits signées, en stéréo entrelacée, et à une fréquene de 44.1KHz, ce qui est de loin le format le plus courant de nos jours.

5.8.2 Le format Microsoft *WAVE* (ou *WAV*)

Cependant, le format PCM n'est pas directement lisible, car il faut que l'utilisateur connaisse le format des données (par exemple, sur combien de bits sont stockés les informations), pour pouvoir les lire. Ainsi, nous avons

choisi d'exporter le son dans un format très standard depuis son utilisation dans *Microsoft Windows*, à savoir le format *WAVE*. Le format *WAVE* est supporté par la majorité des systèmes d'exploitations et systèmes traitant le son que cela soit pour le lire ou bien pour le modifier. Les fichiers *WAVE* sont conformes aux spécifications *RIFF* (*Resource Interchange File Format*). Mais que sont les spécifications *RIFF* ? Il s'agit d'ajouter à des fichiers multimédia des en-têtes, ou *CHUNK*, permettant de délimiter les différentes parties d'un fichier, comme par exemple, plusieurs bandes sons pour un film, ou des informations sur les droits d'auteurs pour de la musique, ou même des index de pistes (*tracks*), permettant de changer de chanson dans un album. Dans notre cas, il s'agit d'inclure les informations sur le format des données *PCM* dans le fichier lui même, et ainsi, permettre sa lecture sur n'importe quel lecteur.
À la création du fichier *WAVE*, nous connaissons quasiment tous les paramètres du fichier, à savoir, qu'il n'y a qu'une seule piste au format *PCM*, qu'il s'agit de données stéréo, avec une fréquence de 44.1KHz, et que la profondeur est de 16 bits. Il ne nous reste plus qu'à calculer le nombre d'octets par seconde, et le nombre d'octets par échantillonage utilisés, et, mise à part la durée totale du fichier, nous pouvons donc écrire dans le fichier tous les paramètres. Ensuite, nous écrivons les données *PCM* qui sont heureusement dans le même format que l'impose la norme *WAVE*, c'est-à-dire des données signées et entrelacées. Enfin, quand le fichier est terminé, nous n'avons plus qu'à remplir la taille dans les en-têtes.

5.8.3 Le format *Ogg/Vorbis*

Le format *WAVE* est un format standard, cependant, il est non compressé, il peut donc être intéressant d'utiliser un format qui l'est. Comme nous imposons déjà à l'utilisateur de posséder les bibliothèques *Ogg/Vorbis*, il y a de fortes chances qu'il possède le programme de compression *oggenc* qui fait partie du même projet. Nous utilisons ce programme externe à notre projet pour compresser les sorties en *WAVE*. Nous permettons à l'utilisateur de choisir son programme de compression, selon la configuration de sa machine, et les éventuelles licences qu'il pourrait acquérir. Comme implémenter des algorithmes de compression n'est pas le but premier de notre projet, ce n'est pas bien gênant. Si le programme est absent de la machine de l'utilisateur, cette fonctionnalitée est tout simplement ignorée.

Chapitre 6

Interface

6.1 Qu'est ce que *Qt* ?

Qt est un utilitaire complet, permettant le développement d'applications graphiques en *C++*. Cet utilitaire a été développé par TrollTech[1].

6.2 Pourquoi *Qt* ?

Bien que nous ayons choisi de programmer en langage C, nous avons choisi un utilitaire qui utilise le *C++*. Et ceci pour plusieurs raisons.
Tout d'abord, *Qt* est un outil d'interface multiplateformes. En effet, il est inutile de rappeler que nous avons prévu d'accomplir notre destiné et donc , de développer une application sous Unix et sous Mac. Il était donc préférable de choisir *Qt* à son concurrent *GTK+*. De plus, *Qt* est orienté objet, ce qui n'est pas négligeable.

6.3 Démarrer avec *Qt*

L'installation de *Qt* est simple. Nous ne nous attarderons donc pas sur la chose. Nous allons plutôt parler de l'utilisation fonctionnelle de *Qt*. En effet, il est possible de taper directement son code *Qt* dans un éditeur, puis de compiler ce code. Cependant, nous avons d'abord préféré faire l'usage

[1] http ://www.trolltech.com/

d'un éditeur de code, qui est lui même livré avec *Qt*. Il s'agit de *Qt* designer.

6.4 Présentation de L'interface

Pour la première soutenance, nous avons fait une interface plutôt simple, a l'aide de *Qt Designer*.

6.4.1 *Qt* Designer

Qt Designer est un éditeur de code, facile à appréhender Il suffit de sélectionner puis déposer le *Widget* voulu. Sur la gauche, nous avons une présentation des différents *Widgets* et les éléments à la droite permettent une meilleure navigation au sein de notre projet.

Pour débuter, il suffit d'un clic sur *File*, *new*, *new project*. Ensuite, on ajoute une *Qdialog* à notre projet. La *Qdialog* sera la fenêtre principale de notre projet, celle sur laquelle sera déposée tous les *Widgets*.

Une fois notre interface réalisée, il ne reste plus qu'à compiler notre projet. L'utilitaire *qmake* nous permet d'éditer un *Makefile*, et une fois celui-ci créé, un simple *make* dans le répertoire et nous sommes en présence de notre projet.

6.4.2 Définition des objets utilisés

- *Widget*
  L'interface graphique est composée de *Widgets*, qui ne sont qu'une façon de nommer les fenêtres, boutons, champs de saisie, et autres combobox.

- *Signal* et *Slot*
  Les *signaux* et *slots* permettent l'interaction entre objets. L'utilisation d'un *Widget*, tel que le clic sur un bouton, produit un événement qui émet un *signal*. Ce *signal* appelle alors un slot, c'est à dire une fonction qui est la réponse correspondante au *signal*. Dans notre cas, le *signal* pourra être un clic sur un bouton, et le slot, la fonction qui sera alors effectuée, par exemple l'effacement, la fermeture.

6.4.3 Première soutenance

Notre première interface était composée de trois *TextEdit*, qui permettent l'écriture du texte, et de quatre boutons. Dans le plus grand des *TextEdit*, nous écrivions la phrase que nous souhaitions. Dans le deuxième *TextEdit*, apparaissaît, le résultat du parsage réalisé. Quant au troisième *TextEdit*, il affichait différents renseignements relatifs au son.

Il y avait aussi quatre boutons (*Pushbutton*). Le premier permettait le parsage de la phrase contenue dans le premier *TextEdit*, selon le principe des "*signals / slots*" expliqué précédemment.
On appelait la fonction de parsage sur ce texte, puis le résultat était affiché dans le second *TextEdit*. Par ailleurs, le bouton lecture permettait selon un même procédé de traiter le texte. Le bouton "effacer" efface les entrées contenues dans le *TextEdit*, et le bouton "quitter" permet de quitter l'application.

6.4.4 Seconde soutenance

A partir de la seconde soutenance, le projet a été réalisé sans *Qt Designer* (et ceci afin de permettre un meilleur portage).

Création de ma propre *Widget*

Pour débuter, un *Widget* a été implémenté par nos soins.
En effet, nous déclarons une nouvelle classe, MyForm, héritée de la classe *QDialog*. Le *Widget* contiendra d'autres *Widgets*, de différentes classes tels que des *QPushButton*, *QTextEdit*, et des *QCheckBox*. Ces derniers sont déclarés comme des variables de type *Private*.
Ensuite, nous déclarons les différents *slots* qui seront appellés, lorsque l'on aura récupéré les *signals* émis par l'application, comme par exemple, le clic d'un bouton.

Nos *slots*

Certains *signals* sont déjà définis, comme celui permettant de quitter l'application. Les autres sont définis par nous même, et déclarés comme *private slots*, au sein de notre classe MyForm.

Il est nécessaire d'insérer le mot *Q_OBJECT* qui est une macro propre à Qt) , pour s'assurer le bon fonctionnement de nos *signals* et *slots*. Il était

prévu de réaliser l'intégration de toutes les parties du projet. C'est la raison pour laquelle, trois fonctions, trois *slots* ont été définis. Le premier *slot* est appelé lorsque le *signal* de clic sur le bouton *Parser* est détecté. À ce moment, la fonction gérant le parsage est appelée, et le résultat présenté, sur le plus grand des *QTextEdit*, l'autre servant à écrire le message que l'on souhaite écrire.

6.4.5 Troisième soutenance

De nouvelles modifications ont été apportées. En effet, notre application dispose dorénavant d'une barre de menus (*Widget* reconnu sous le nom de *QMenuBar*). Ce *Widget* est un descendant de la *QDialog*, qui rappelons le, forme la fenêtre principale de notre projet.

Toutes ces nouveautés font de notre interface une application un peu plus aboutie. De fait, les utilisateurs ont une plus grande préférence pour des applications soignées, surtout faciles à manipuler. Par ailleurs, il se trouve qu'une barre de menus confère un aspect plus professionnel au projet. Et pour finir elle nous permet d'utiliser des boîtes de dialogue.

Les *QFileDialog* sont des *Widgets* descendants de la *Qdialog*. Ce *Widget* possède différentes propriétés, *slots*, *signals* tels que *GetOpenFilename*, qui nous permettent d'ouvrir une boîte de dialogue afin d'ouvrir un fichier, ou bien de l'enregistrer. Des filtres peuvent être aussi prédéfinis, pour que l'utilisateur final puisse employer à loisir un fichier sous tel ou tel format. Ainsi, pour l'enregistrement des sons, nous avons définis un filtre autorisant uniquement l'enregistrement de fichiers de type *.wav*, *.raw* ou *.ogg*.

Les *QFileDialog* sont appelées à partir de *Popupmenu*. Un *Popupmenu* est le *Widget* rend possible le fait de regrouper des exécutables, des fichiers ou des dossiers dans un menu disponible à partir d'un raccourci. Les *Popupmenu* font partis de la classe des *QPopupmenu* qui héritent des *Qdialog*.

Comme on peut le voir deux nouveaux *Widget* sont apparus. Les premiers sont des *Qlabel* qui ont pour mission de nous permettre d'afficher du texte directement dans notre application. Ces *Widget* sont à l'opposé des *QTextedit*, qui, rappelons le, nous permettent de saisir du texte une fois notre application lancée.

Les *QTextedit* sont là pour permettre à l'utilisateur d'insérer de nouveaux mots dans le dictionnaire depuis l'interface. Dans le premier *QTextedit*, l'utilisateur rentre le mot (ou *lexème*) et dans l'autre sa version phonétique. Un clic sur le bouton *Ajouter* et voilà notre nouveau mot ajouté directement dans le dictionnaire !

6.4.6 Soutenance finale

Très peu de choses ont changées : une harmonisation entre les versions **MacOS X** et **Linux**, ainsi qu'un nettoyage du code.

Chapitre 7

Annexes

7.1 Installation

L'installation du projet est très simple puisque celle-ci respecte les normes GNU. Il s'agit donc des instructions génériques d'installation GNU.
Tout d'abord, désarchivez la tarball du projet dans un répertoire quelconque. Entrez dans le répertoire ALIAS créé.

7.1.1 Configuration

Pour une liste complète des options, tapez `./configure --help`. Il est tout de même bon de savoir que le projet exige la présence de *OGG/Vorbis*. Donc, si ces bibliothèques ne sont pas dans un répertoire traditionnel comme `/usr/lib`, il vous faudra préciser leur chemin à l'aide de ces deux options :

```
--with-ogg-prefix=<chemin vers ogg>
--with-vorbis-prefix=<chemin vers vorbis>
```

L'exemple le plus classique reste sur **MacOS X** car on a coutume d'installer ces bibliothèques à l'aide de *fink*[1] :

```
./configure --with-ogg-prefix=/sw --with-vorbis-prefix=/sw
```

7.1.2 Compilation

Si `./configure` (la configuration) s'est bien déroulée, la compilation ne devrait pas poser de problème. Tapez make et allez vous faire un café,

[1] http ://fink.sourceforge.net/

il y a en général 2 minutes de compilation.

7.1.3 L'installation

Si `make` (la compilation) s'est bien passée, vous pouvez installer *ALIAS* à l'aide de la commande `make install`. MAIS, si vous avez laissé le répertoire d'installation par défaut (en général `/usr/local`), vous devrez avoir les privilèges **root**.
Pour les Utilisateurs **MacOS X**, déplacez l'application créée (à la racine du projet), vers le répertoire de votre choix (habituellement Applications).

7.2 Site web

Voici la page de téléchargement où l'on peut obtenir la version d'ALIAS prévue pour la soutenance finale.

FIG. 7.1 – La page "DOWNLOADS" du site

7.3 Bibliographie

- O'Reilly – *Maîtrise des algorithmes en C*
- O'Reilly – *LaTeX par la pratique*
- O'Reilly – *Learning GNU Emacs*
- O'Reilly – *Learning the bash shell*
- O'Reilly – *Managing Projects with GNU Make*
- New Riders – *GNU Autoconf, Automake, and Libtool*
- CVS – *http ://www.gnu.org/software/cvs/*
- PHP – *http ://www.php.net/*

Chapitre 8

Conclusion

C'est avec fierté et une certaine joie non dissimulée que nous pouvons annonçer que la mission Alias a été accomplie avec succès. Le projet respecte parfaitement le cahier des charges et est complètement fonctionnel. Il est important de noter qu'aucun des membres du groupe n'a péri au cours de la mission, ou n'est sur le point de disparaître. Le projet de cette année nous a beaucoup apporté, tant sur le plan de l'acquisiton des connaissances que sur le plan personnel, et nous attendons avec impatience les nouvelles aventures que nous réservent les années d'ingénieries.

Mission accomplie, en attente de nouvelles instructions...

Rapport final terminé.

ALIAS