Rapport de deuxième Soutenance

SD\n – ALIAS

*<**A** **L**exeme **I**s **A** **S**ound>*

Julie THÉODOSE-SOREL Mathieu BASCLE
Nicolas TANDÉ Yohan BAINIER

23 avril 2005

Table des matières

1 Introduction

Équipe Section Disparue retour à la ligne, au rapport pour cette seconde soutenance.
Ce rapport présente l'état d'avancée de notre projet, et plus particulièrement, celui de la mission S-II.

La mission à accomplir pour cette soutenance, était la suivante : présentation d'une *Interface* terminée par la recrue THÉODOSE SOREL, un *Parser* plus efficace, couplé avec une liste d'exceptions, fournis par l'Officier BAINIER et la recrue BASCLE, et une lecture des sons par *Speaker* plus fluide, tache réalisée par la recrue TANDÉ

La mission a été effectuée sans pertes humaines, ni matérielles (Un exploit !). La tâche était ardue, nous étions constamment assaillis par des ennemis féroces (plus communément connus sous le nom de *Vacances D'hiver*), mais c'est avec fierté que nous vous présentons le résultat de cette mission.

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.

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*.

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

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.

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.

2.3 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.

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.

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++).

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*).

autoconf

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

2.4 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.

[2]Littéralement : programme d'amorce
[3]GNU M4 est un outil à multiusage pour le traitement de texte

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 en-

semble 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 $log(n)$.

4 Module de Traitement du Langage Naturel

4.1 Liaisons

Lors de la première soutenance, le parser[4] ne gérait pas les liaisons entre les mots si bien que la phrase “On a envie de dormir” produisait la série de phonèmes (presque juste) [~ a B v i d e d o R m i R]. Voici maintenant ce qui se passe avec l’amélioration que nous lui avons apporté :

- ’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 B v i d e d o R m i R].

[4] analyseur syntaxique

4.2 Dictionnaire

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 nouvelle 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.

5 Audio

Lors de la dernière soutenance, nous avons implémenté la lecture de fichiers *Ogg Vorbis* à l'aide des systèmes *OSS* et *ALSA*. Cependant 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.

5.1 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.

5.2 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.

5.3 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 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.
Cependant, le rendu actuel n'est pas totalement fluide, ce qui provoque des grésillements sonores, mais cela devrait être corrigé pour la prochaine soutenance, dans la mesure où nous copierons l'ensemble des enregistrements sur le disque, afin de pouvoir le sauvegarder sous différents formats.

6 Interface

Précédemment, nous avons présenté une interface réalisée avec *Qt* qui comme nous le rappelons, est un utilitaire permettant le développement d'applications graphiques en C++. Ce projet avait été fait avec *Qt Designer*. Pour cette soutenance, il a été défini que l'interface serait finie, et que celle-ci serait réalisée sans *Qt Designer* (et ceci afin de permettre un meilleur portage).

6.1 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 appelés, lorsque l'on aura récupéré les *signals* émis par l'application, comme par exemple, le clic d'un bouton.

6.2 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.
La partie son est intégrée, puisque désormais le clic sur le bouton *clic* permet de lancer les sons de Nicolas.

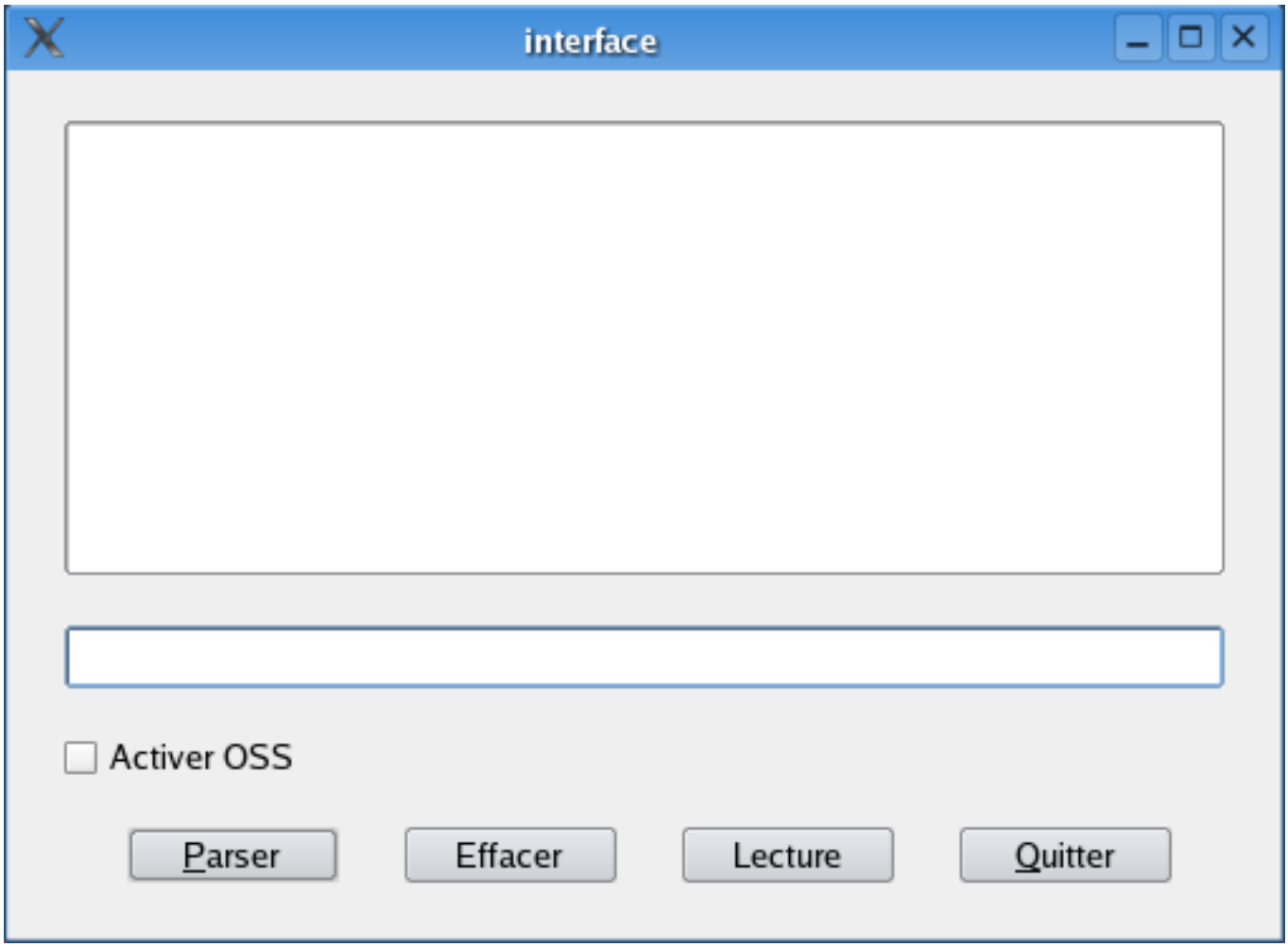

FIG. 1 – La nouvelle interface

Nous pouvons donc dire que cette partie finie. Désormais, nous pourrons entièrement nous consacrer à d'autres parties de notre projet.

7 Conclusion

Seconde partie du projet terminée. Mission S-II terminée. En effet, toutes les tâches ont été correctement effectuées par l'ensemble de l'équipe.

Prochaine mission, nom de code : S-III. Les tâches à réaliser, seront les suivantes : implémentation de la gestion des caractères spéciaux, par le *Parser*, puis export du son vers différents formats, portage du son sous Mac OS X.

Cette mission ne sera pas facile à mener à bien. Divers ennemis jalonneront notre route, notamment, *Contrôle de Mathématiques*, *Contrôle d'Algorithmique* et plus tard, l'ennemi ultime , plus connu sous le nom de *Vacances de Pâques*. Mais la victoire sera notre.

Second rapport terminé.