HLIN601 - TER

Développement d'un moteur de preuves pour la logique classique propositionnelle

Auteurs :
M. Quentin ANDRE
M. Eliott DUVERGER
Mme Marie LEGRAND
M. Baptiste DARNALA

Encadrants :
Pr. David DELAHAYE

Version 0.1 du rapport de Ter
15 novembre 2019

Remerciements

Nous tenons à remercier M David Delahaye, notre professeur encadrant, pour nous avoir guidé et conseillé pendant toute la durée de ce projet. Ces précieux conseils nous ont permis de faire face et de surmonter les nombreux problème que nous avons rencontré durant ce projet.
Nous tenons également a remercier le personnel du LIRMM pour nous avoir accueilli dans leur locaux lors des réunions hebdomadaire avec notre encadrant.

Table des matières

Introduction **1**

1 Domaines de l'informatique dans lequel se situe le projet **3**

1.1 Notions de logique classique 3

1.2 Analyse Lexicale et Syntaxique 4

1.3 Objet, Héritage et Arbres de preuve 4

1.4 Interface utilisateur 4

2 Présentation du problème précis sur lequel nous avons travaillé **7**

2.1 Analyse Lexicale et Syntaxique 7

2.2 Objet, Héritage et Arbres de preuve 7

2.3 Création d'une interface graphique 8

3 Travail réalisé **9**

3.1 Introduction 9

3.2 Analyse syntaxique 9

3.2.1 Flex : l'analyse lexicale 9

3.2.2 Bison : l'analyse grammaticale 10

3.3 Objet, Héritage et Arbres de preuve 11

3.3.1 Objet en logique classique 11

3.3.2 Séquent et Arbres de Preuves 12

3.4 Interface graphique sur le terminal 13

Conclusion **15**

Bibliographie **17**

Introduction

Dans le cadre de notre troisième année de licence et de l'UE TER [1] L3, il nous a été donné le sujet suivant : "Développement d'une interface graphique servant à réaliser des preuves en logique classique" proposé par le Pr. David Delahaye. Le projet a commencé le 22 janvier 2018 et a prit fin le 27 avril de la même année.

L'objectif de ce projet est de permettre à un utilisateur de résoudre une preuve logique étape par étape sous la forme d'un arbre. Pour cela, l'utilisateur, en choisissant la prochaine à développer, ajoute une feuille à l'arbre correspondant à la transformation d'un connecteur logique.

Lors de notre première rencontre avec notre encadrant, celui-ci nous a introduit à la résolution de preuves de logique classique propositionnelles. Forts de notre formation de Licence Informatique à l'Université de Montpellier ainsi que l'accompagnement de notre encadrant, nous avons acquis les connaissances nécessaires à la création d'une application à destination des étudiants de logique.

D'un commun accord, nous avons décidé d'utiliser le C++ (vu en cours durant les années L2 et L3) dans l'objectif d'utiliser l'environnement de développement Qt. Ce dernier semblait une alternative abordable, efficace et esthétique pour réaliser des interfaces graphiques. Sur les conseils du Pr. Delahaye, nous avons utilisé les librairies Lex & Yacc nous permettant d'analyser les formules à traiter.

Dans un souci d'efficacité, nous avons appliqué la méthode du "Pair Programming". Cette méthode consiste à travailler en binôme sur une partie spécifique du projet ; par exemple la modélisation, l'analyse du texte ou le développement d'un interface graphique.

Quels sont les enjeux de la création d'une interface graphique dynamique on verra plus tard ?

Dans un premier temps, il sera question de la modélisation informatique de la construction de preuves de logique classique. Nous développeront ensuite l'analyse lexicale et syntaxique d'une formule du premier ordre. Pour finir, nous parlerons de la réalisation d'un interface graphique avant de conclure.

1. Travaux Étudiant Recherche

Chapitre 1

Domaines de l'informatique dans lequel se situe le projet

1.1 Notions de logique classique

Avant de commencer, définissons la logique classique propositionnelle. Elle permet l'évaluation de formules (dîtes formules bien formées) construites sur la base d'un alphabet, d'une grammaire et d'une syntaxe. Le mode d'évaluation est booléen, c'est à dire que chaque formule bien formée renvoie une valeur de vérité : VRAI ou FAUX. Elle est composée de variables représentée par des lettres, des connecteurs logiques (NON / ET / OU / IMPLIQUE / ÉQUIVALENT) qui peuvent être unaires (comme c'est le cas du NON) ou binaires (ce qui correspond au reste des connecteurs). Un connecteur unaire ne peut être utilisé que devant une seule formule bien formée. Par opposition, les connecteurs binaires se placent entre deux formules.

$$NON : \neg A$$
$$ET : A \wedge B$$
$$OU : A \vee B$$
$$IMPLIQUE : A \rightarrow B$$
$$EQUIVALENT : A \longleftrightarrow B$$

La notion de preuve de logique classique propositionnelle correspond à l'application de règles d'inférence sur des séquents. Un séquent est une ensemble d'hypothèses et un ensemble de conclusions, les deux étant séparées par le symbole $\vdash$.

$$\boxed{A \wedge B,\ C \vee B \quad \vdash \quad B \rightarrow C,\ \neg A \wedge B}$$

Le but de la preuve est de développer une hypothèse ou une conclusion pour avoir d'autres séquents. Un arbre de preuve est alors formé, et la preuve n'est terminée que lorsque chaque feuille de l'arbre est fermée. Pour fermer une feuille il faut trouver un axiome. Un axiome est une proposition indémontrable qui sert de fondement à une théorie mathématique. En logique, il définit une FBF que l'on ne peut plus dériver par les règles d'inférence. L'application successive des règles permet l'obtention d'un arbre dont les feuilles sont des axiomes. L'obtention d'un axiome ferme la feuille du-dit arbre, on ne peut donc plus continuer à la développer. La preuve se termine lorsque toutes les feuilles de l'arbre sont des axiomes.

$(P \vee P) \Rightarrow P \wedge C$

1.2 Analyse Lexicale et Syntaxique

L'analyse syntaxique est utilisée dans de très nombreux domaines de l'informatique. En effet, aujourd'hui il est très fréquent qu'un utilisateur entre une chaîne de caractère qui sera analysée, transformée, et utilisée par le programme. Cette technologie est principalement utilisée dans le développement d'intelligence artificielle, dans le génie logiciel et dans la recherche en général.

1.3 Objet, Héritage et Arbres de preuve

La programmation objet est également utilisée dans de très nombreux domaines de l'informatique. En effet, la modélisation de phénomène et d'objet réel grâce a la programmation objet ainsi que le principe d'héritage permet d'identifier et de capter toutes les propriétés des objets en question.
De même, la structure d'arbre que nous utiliserons pour modéliser les preuves et les étapes de calcul des séquents, est récurrente dans de nombreux domaines de l'informatique. Dans notre cas, le choix d'une structure en arbre pour les preuves fut assez évidentes, étant donné la forme que prenne les preuves que nous souhaitons modéliser.

1.4 Interface utilisateur

Il est aujourd'hui difficile d'imaginer un logiciel tout public sans interface graphique destiné a faciliter son utilisation. Presque la totalité des softwares d'aujourd'hui disposent

d'une interface graphique qui est bien plus ergonomique qu'une succession de lignes de code.

Chapitre 2

Présentation du problème précis sur lequel nous avons travaillé

2.1 Analyse Lexicale et Syntaxique

Le problème précis était donc que l'utilisateur entre sa formule logique, et qu'elle soit analysée, et interprétée par le programme de façon à ce qu'on puisse la traiter. Il nous est donc apparu deux choix : laisser l'utilisateur entrer sa formule et l'analyser avec un parser [1] au clavier ou utiliser un clavier virtuel.

La solution du clavier virtuel était assez simple pour la création de l'objet : Formule en C++. Cependant, imaginer l'utilisateur cliquer un par un sur chaque symbole nous repoussait car cela n'était pas ergonomique. C'est pourquoi nous avons opté pour la solution de l'analyseur syntaxique.

Le projet étant déjà commencé en C++ nous nous sommes tournés vers les technologies Lex & Yacc puis par la suite Flex & Bison (Qui sont similaires).

2.2 Objet, Héritage et Arbres de preuve

La modélisation et la manipulation des formules, des séquents, et des arbres de preuves a poser très peu de problèmes. L'essentiel du problème résidant dans les choix a faire pour rendre le moins complexe possible la manipulation des séquents et la génération des objets. Le problème précis de cet partie est donc le suivant :

Pour pouvoir réaliser les preuves présentés dans la section précédentes, il nous fau-

1. Analysateur syntaxique en anglais

dra développer une structure d'arbre de preuve permettant de développer les séquents en respectant les règles de calcul des séquents. Il faudra également modéliser les séquents eux-mêmes, comprenant les formules représentant les hypothèses et les conclusions. Enfin il faudra modéliser les formules et leur spécificité :
On veut pouvoir identifier
Les variables,
Les différents opérateurs binaires et unaire
$\wedge, \vee, \rightarrow, \longleftrightarrow, \neg$

La création des objets énumères ici c'est donc faites en C++, ce langage nous permettant de les manipuler facilement.

2.3 Création d'une interface graphique

Une partie du projet consistait à réaliser une interface graphique de notre application. Cette interface a pour but de faciliter le développement des différentes branches de l'arbre graduellement pour arriver à obtenir tous les axiomes de l'arbre. Idéalement on commencerait par taper sa formule dans une entrée texte, puis on développerait certaines hypothèses (ou conclusions) en cliquant dessus. Après quelques développements une arborescence devrait être visible. Et une dernière problématique serait l'échelle : une fois la preuve conséquemment développée soit l'arbre sort de l'écran. Il faudrait donc un système de zoom et de déplacement pour pouvoir parcourir le plus librement possible l'arbre de preuve. Toutes ces fonctionnalités nécessitent d'utiliser la bibliothèque de *QtCreator* pour fonctionner. Un des principaux enjeux de la réalisation de cette interface a donc été l'union du code C++ lui-même utilisant les outils Lex & Yacc.

Chapitre 3

Travail réalisé

3.1 Introduction

Pour la partie d'analyse du texte nous avons donc commencé à utiliser les technologies Lex & Yacc puis vers Flex et Bison. Ces outils génèrent des grands fichiers en C, Flex génère un scanneur lexical et Bison produit un parser.

3.2 Analyse syntaxique

Pour travailler sur la partie analyse syntaxique, nous avons tout d'abord décidé d'utiliser Lex & Yacc comme nous avait suggéré notre professeur encadrant. Mais après de nombreux essais infructueux en utilisant ces outils, surtout dus a des échecs de compilation, nous avons décidé d'utiliser des versions plus récentes possédant une communauté plus active et une meilleure documentation : Flex & Bison.

3.2.1 Flex : l'analyse lexicale

Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux. Un terminal représente tout ce qui peut être dans la formule, par exemple les littéraux ou les connecteurs logique comme ET ou IMPLIQUE. L'analyseur syntaxique va analyser toute la formule et ensuite envoyer tous les terminaux à l'analyseur grammaticale pour qu'il puisse travailler dessus.

L'outil que nous avons utilisé pour l'analyse lexical s'appelle Flex et c'est dans un fichier .lex que nous avons écrit tout les différentes règles qui vont produire les terminaux.

Un fichier Flex est divisé en plusieurs partie, une partie déclaration et une partie pro-

duction

La partie déclaration peut contenir à la fois du code écrit dans le langage cible qui va être délimiter par %{ et %} et un partie expression régulière. Pour notre fichier nous avons du liée tous les fichiers externes en utilisant les "include" du C++. Pour la partie expression régulière, nous avions besoin d'une règle pour les "blanc" (espace ou tabulation) et une règle pour les lettres. Les expressions régulières sont sous la forme :

notion expression_reguliere

Les notions définies seront utilisées dans la deuxième partie du fichier, la partie production.

Dans cette partie, on indique a Flex ce qu'il devra faire lorsqu'il rencontrera telle ou telle notion. Les production sont de la forme :

expression_reguliere action

Si l'action comporte plus d'une ligne, elle devra être écrite entre deux accolades dans le langage cible. Dans notre fichier nous avons du écrire plusieurs règles, il y a une règle "blanc" qui fait que l'analyseur ne doit rien faire en cas de "blanc" et passé au caractère suivant. Il y a une règle "var" qui utilise la variable interne de Flex et Bison "yylval" qui est une variable global dans laquelle on stocke la valeur sémantique du terminal. Dans cette variable on stocke un nouvelle objet de type "var" construit en utilisant la variable interne de Flex et Bison qui représente le ou les caractère détecté par Flex. La structure "Var" a besoin d'un paramètre de type string mais "new var(yytext)" ne marche pas car c'est un pointeur alors on doit utiliser la fonction strdup() qui renvoie un pointeur sur une nouvelle chaîne de caractères qui est dupliqué depuis "yytext". Toutes les autres règles sont pour les connecteurs logiques, les parenthèses et le retour à la ligne qui représente la fin de la formule. Ce sont des règles simples qui vont juste renvoyer un terminal qui sera utilisé pas Bison.

3.2.2 Bison : l'analyse grammaticale

Le fichier Bison est lui aussi composé de plusieurs parties distinctes : l'entête, les symboles terminaux et les règles de productions.

Dans l'entête, on lie les autres fichiers du projet et on définit certaine fonctions globales dont yyparse() ; (fonction principale du parser).

La partie suivante définit les symboles terminaux : Les tokens et les opérateurs. Les tokens sont les chaînes de caractères reconnues par Flex qui seront considérées comme une seule entité. On a donc tout se dont on a besoin :

- VARIABLE
- NON

— ET / OU / IMPLIQUE / ÉQUIVALENT
— PARENTHÈSE GAUCHE / PARENTHÈSE DROITE
— FIN

Ensuite, pour chaque opérateur, on précise son association opérative %left ou %right. Par exemple, le NON s'associe à droite alors que les connecteurs binaires s'associent à gauche.

Bison va alors les gérer comme une pile et les traiter. Pour ce faire, il faut donner des règles de production. Ces règles vont appeler les constructeurs des objets C++. Pour donner un exemple, s'il détecte une expression telle que : **expression** IMPLIQUE **expression** La règle appellera le constructeur du connecteur binaire en question (avec en paramètre les deux expressions). Les règles vont alors s'appliquer tant jusqu'à ce la totalité de la formule logique soit créée.

Pour récupérer cette formule nous appelons donc la fonction yyparse() ; depuis le fichier main du projet. Pour l'assigner à une variable, on passe en paramètre une référence sur un objet FORMULE qui, indirectement, sera modifiées.

3.3 Objet, Héritage et Arbres de preuve

3.3.1 Objet en logique classique

Nous avons utilisé la POO [1] pour représenter les différentes notions de logique classique. On distingue donc :

— Formule est de type éponyme. Elle représente toutes les fbf.
— Les variables sont de type var et est représenté par une chaîne de caractères. Une variable est un fbf elle hérite donc de formule.
— L'opérateur NOT, l'objet de type Op_not qui représente l'opérateur unaire de négation d'une fbf. Cet opérateur est également une formule (par héritage).
— L'ensemble des opérateurs binaires sont de types Op_binary. Un opérateur binaire prend en compte deux fbf (de chaque côté de son symbole) pour retourner une valeur de vérité. Le type Op_binary hérite lui aussi de formule.

Il y aussi les 4 opérateurs binaires qui hérite tous de op_binary :

— L'opérateur AND représenté par l'objet op_and
— L'opérateur OR représenté par l'objet op_or
— L'opérateur IMPLIQUE représenté par l'objet op_imp
— L'opérateur EQUIVALENCE représenté par l'objet op_equ

Chaque opérateur binaire possède un constructeur prenant en paramètres 2 formules afin de pouvoir initialiser les attribut "gauche" et "droite" de nos opérateur binaire et créer

1. Programmation Orientée Objets

dynamiquement nos formules.
De la même manière, l'opérateur NOT lui possède un constructeur prenant en paramètre une formule pour initialiser l'attribut "suiv" de NOT qui désigne la FBF venant après la négation.
L'objet var lui possède un constructeur qui prend un chaînes de caractères en paramètres pour initialiser l'attribut identifiant la variable : l'attribut "form" (qui est déjà présent dans formule). Afin de pouvoir afficher les formules que représentons, il a fallu ajouter un attribut "form" à la classe qui est une chaîne de caractères, de cet manière tout nos opérateurs en hérite. De cet manière, lorsque l'on crée un objet op_imp on va initialiser form a -> .
Il nous a fallu également créer un attribut "TYPE_FORMULE" dans la classe formule et "TYPE_OPERATEUR" dans la classe Op_binary c'est de attribut sont essentiel pour pouvoir identifier le type de formule que l'on souhaite traiter, et de quel opérateur il s'agit quand on a faire à un opérateur binaire. Cela est essentiel notamment pour pouvoir décider de la méthode de calcul à appliquer dans un séquent.
Avec tout cela nous avons de quoi créer et afficher nos FBF, nous pouvons maintenant aborder le véritable support de preuve.

3.3.2 Séquent et Arbres de Preuves

Pour représenter nos séquent, nous avons créer une classe séquent doté de 2 attribut : 2 listes de formules représentant d'une part les hypothèses et d'autres part les conclusions. A cela s'ajoute une méthode qui permet de déterminer si le séquent est un axiome c'est-à-dire qui vérifie simplement si il existe une formule hypothèse égale à une formule conclusion.
Il y a également une fonction d'affichage qui affiche une par une les formules hypothèses puis les formules conclusion. La particularité de cette fonction d'affichage est qu'elle affichera un index après chaque formule. Cet index permettra a l'utilisateur sélectionner la formule qu'il souhaite développer.
Enfin les autre méthode de la classe séquent sont les méthodes de calcul de séquent, elles prennent toutes un index (entier) en paramètre qui correspond à l'index de la formule que l'on veut développer.

La classe ArbrePreuve à pour attribut "fils" un tableau d'arbre de preuve qui sera de longueur 0 si l'arbre est un feuille, 1 si l'arbre n'a qu'un arbre fils, et 2 si l'arbre a deux arbre fils. Ces arbres fils sont crées lorsque l'on développe le séquent attribut de l'arbre "s". Aussi il y un attribut "clôture", un booléen qui permet de savoir si l'arbre est clos ou pas. On dit d'un arbre qu'il est clos si il a été développer (qu'il a un ou deux arbre fils) ou que le séquent de l'arbre est un axiome.
On ne pourra développer qu'une seule fois un arbre, et on ne pourra développer que les feuilles qui ne sont pas des axiomes. L'arbre sera entièrement clos lorsque toutes les feuilles seront des axiomes, ce qui est synonyme de fin de la preuve.

La classe ArbrePreuve comprend aussi un constructeur qui créer un arbre a partir d'un séquent. Il y a aussi une fonction d'affichage qui affichera l'arbre est ces fils. Et un fonction développer(index), c'est cette fonction qui permettra à l'utilisateur de développer la formule qu'il souhaite. Étant donné qu'il existe deux règle de calcul pour chaque opérateur, une applicable sur les hypothèse et une sur les conclusions, il suffit de récupérer le type de formule à l'index donné pour savoir quelle méthode de calcul appliquer. Il y a une seule exception, c'est la méthode de calcul de l'équivalence dans les hypothèse qui possède deux méthode différentes, on laissera alors le choix a l'utilisateur de choisir l'une des deux.
Avec tout cela nous avons toutes les classes nécessaires pour réaliser les preuves par calcul des séquents.
Ci-dessous, un diagramme de classe synthétisant les classes d'objets que nous avons créer.

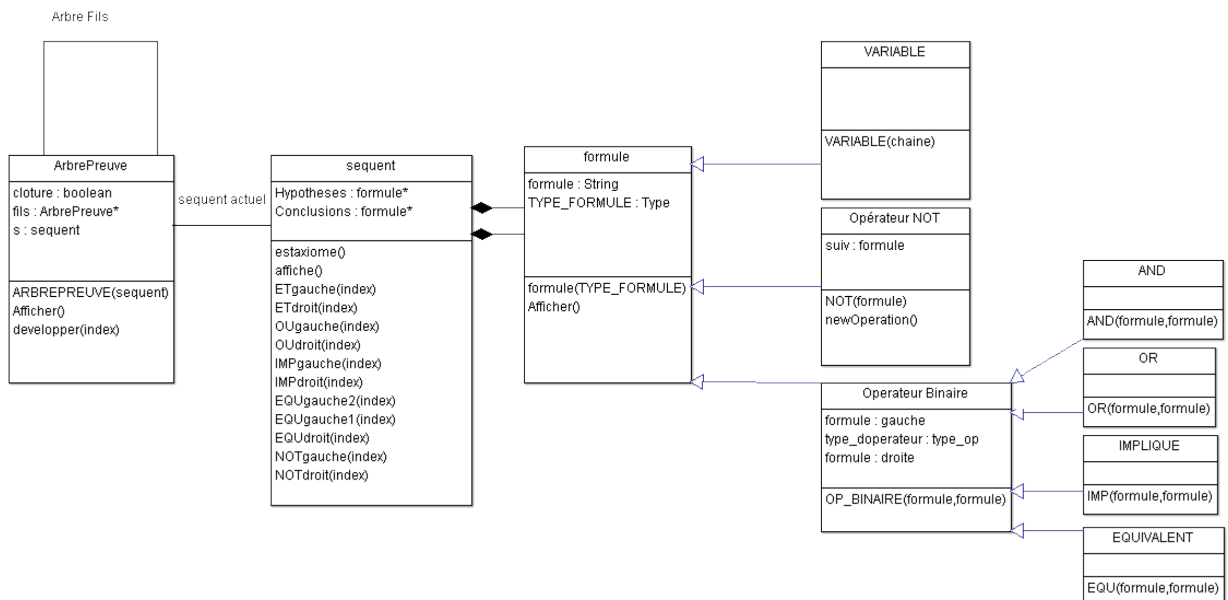

FIGURE 3.1 – Diagramme de classe

3.4 Interface graphique sur le terminal

A cause de problèmes pendant le développement du projet nous avons perdu du temps, c'est pourquoi nous avons fait le choix de faire une croix sur une vraie interface graphique.

Ce choix a principalement été motivé par les difficultés rencontrées au moment de faire cohabiter *QtCreator* et Lex & Yacc. Nous avons donc opté pour un affichage terminal.

Nous avons donc intégré dans le code du projet un moyen d'afficher l'arbre de preuve contenant les séquents obtenus et leurs hypothèses et conclusions relatifs. Les hypothèses et les conclusion sont numérotés pour que l'utilisateur puisse indiquer à quel endroit il veut développer la preuve. Il y a bien entendu une vérification sur le fait que l'utilisateur rentre un chiffre valide.

L'affichage d'un séquent est de la forme :

$>(i$ FROM $j)$
$>(Indice)$ Hypothèse(s)
$>$ ——————————
$>(Indice)$ Conclusion(s)

Avec i et j deux entiers respectivement l'indice de séquent et l'indice du séquent dit parent. La racine aura l'indice (0 FROM 0). Les indices d'hypothèses ou de conclusion sont des caractères [a-zz].

L'utilisateur pourra alors saisir le séquent qu'il souhaite développer puis l'hypothèse ou la conclusion à utiliser.

A l'obtention d'un axiome, l'utilisateur est informé et la branche ayant donné cet axiome est alors fermée, il est alors demandé à nouveau à l'utilisateur de saisir un indice de séquent.

La fin du programme est possible à tout moment si l'utilisateur entre la commande "QUIT" ou si tous les axiomes ont été trouvés, ce qui affiche le message : "Toutes les branches de l'arbre sont fermées car tous les axiomes ont été trouvés. Il n'y a plus d'action possible !" et l'arbre est affiché en entier.

Pour plus de confort, les branches fermées ne sont pas affichées. Si l'utilisateur souhaite voir l'arbre complet (au stade de développement courant), il suffira de taper la commande "PRINT ALL" au moment d'entrer le numéro de séquent. L'affichage revient à son état réduit lors de la prochaine commande.

Le programme d'affichage récupère tout simplement les données déterminées par le programme C++ et les affiche par profondeur.

Tout d'abord affichage de la racine. Ensuite affichage de tous les séquents ayant pour indice parent 0, puis tous les séquents ayant pour indice parent 1, etc...

Conclusion et perspectives

Ce projet a été pour nous l'occasion de découvrir des sujets que nous avions peu ou jamais abordé en cours tel que l'analyse lexicale avec Flex Bison, ou encore la création d'une interface utilisateur. En parallèle, cela nous a permis de consolider et d'approfondir nos connaissances dans la programmation Objet Avancé ainsi que la gestion des structure particulières comme la structure d'arbre pour les preuves. Cependant, tout ne s'est passé comme nous le souhaitons : Même si le moteur de preuve est tout à fait fonctionnel, nous avons été contraints de nous contenter d'une interface utilisateur via le terminal plutôt qu'une véritable interface graphique comme il était initialement prévu. En effet nous avons perdu une quantité de temps non négligeable sur la partie d'analyse syntaxique (à cause de problème de compatibilité entre technologie). Cela rentre dans les perspectives d'avenir et de mise à jour de notre logiciel. Avec cela, nous pouvons envisager d'étendre ce moteur de preuve à la logique du premier ordre que nous avons vu cet année, mais également d'imaginer un moyen de récupérer l'arbre de preuve généré par exemple au format PDF.

Bibliographie

https ://www.gnu.org/software/bison/manual/bison.html

http ://www.linux-france.org/article/devl/lexyacc/

https ://fr.wikipedia.org/wiki/Calcul_des_s%C3%A9quents

http ://aquamentus.com/flex_bison.html

http ://bayledes.free.fr/systeme/bisonflex.phtml

https ://www.enib.fr/ harrouet/Data/Courses/Flex$_{B}$*ison.pdf*

http ://tldp.org/HOWTO/Lex-YACC-HOWTO-5.html