# Résolveur d'expressions logiques

Zachary McSween Manickchand
MCSZ82330405
mcsz01@uqo.ca

2025-06-30

**Résumé**

L'objectif du projet est de résoudre une expression logique. Pour ce faire, l'expression est transformée en un arbre binaire. Cela permet au programme d'utiliser un algorithme d'exploration gourmande pour optimiser l'arbre. Après avoir été optimisé, l'arbre est ensuite transformée en string, de sorte que l'utilisateur puisse voir l'expression logique optimisée.

**Mots clés :** algorithmes, rapport étudiant, algorithme gourmande, arbre binaire.

## 1 Introduction, problématique et objectifs du projet

Au cours du premier trimestre, les étudiants apprennent les expressions logiques en mathématiques discrètes (MAT1153). L'objectif du projet est de simplifier les expressions logiques en les réduisant à leur forme minimale. Alors qu'il est plus simple pour les gens de déterminer les expressions logiques à l'aide de tables de vérité, ce projet utilisera les lois de la logique pour déterminer la forme minimale.

En raison des exigences du projet, cela signifie que le projet doit être un simple interprète. Mais vu autrement, ce projet donne l'occasion d'explorer le fonctionnement interne d'un agent rationnel par le biais d'une base de connaissances. La logique du premier ordre (FOL) étant le moteur d'un agent rationnel, et comme s'agit d'un super-ensemble de la logique propositionnelle, le projet de résolution d'expressions de logique propositionnelle est donc l'un des éléments constitutifs d'un agent.

### 1.1 Définition de la tâche

Le projet simplifie une expression de logique propositionnelle, ce qui signifie que le programme prend une entrée utilisateur (ici une chaîne de caractères) et fournit à l'utilisateur une expression simplifiée correspondante qui est équivalente. Cela signifie implicitement que le système qui doit être mis en œuvre est un interprète. Cet interprète transforme l'entrée de l'utilisateur en un arbre qui peut ensuite être utilisé dans l'étape de simplification. Cette étape utilise un algorithme gourmand qui essaie séquentiellement 3 techniques de simplification. Après l'étape de simplification, il retransforme l'arbre en une sortie utile pour l'utilisateur (sortie chaîne de caractères).

## 2 Littérature existante

### 2.1 FOL

Dans le FOL d'un agent, les avantages seraient **1)** la résolution d'expressions guidée dans l'arbre par le système de FOL et **2)** la simplification à l'avance des expressions. Dans le premier cas, cela permettrait au système de résoudre les problèmes logiques liés à cette « unité » (le projet en cours) et serait donc utile lorsque l'expression est en cours de modification. Par exemple, le système pourrait simplifier l'expression si l'un des arguments est une tautologie. Dans ce dernier cas, le système pourrait

simplifier l'expression si l'un des arguments est une tautologie, ce qui permettrait au résolveur FOL d'effectuer moins de calculs au moment de la requête de l'utilisateur.

Ainsi, en supposant que le développement de ce projet soit plus avancé, le résolveur FOL serait simplement le directeur de ce résolveur *propositionnel*.

### 2.2 Projet existant

Des projets existants similaires ont abordé la logique propositionnelle, il s'agit de Z3 et de Prolog.

Z3 est un producteur de théorèmes de Microsoft Research [1] qui résout les problèmes de raisonnement automatisé. Le producteur de théorèmes de Z3 est un solveur de satisfiabilité modulo des théories (SMT), ce qui signifie qu'il traite de nombreux problèmes pratiques [2], la logique propositionnelle n'étant qu'un sous-ensemble de ses capacités. L'approche de ce projet est différente simplement en raison de son champ d'application plus restreint.

D'autre part, Prolog (***Pro****grammation en* ***log****ique*) est un langage de programmation qui est ancré dans la FOL [3], [4]. Puisque le langage de programmation est un langage qui résout les problèmes de la FOL, son but premier n'est pas de simplifier la logique propositionnelle, même si cela améliore ses performances. L'approche de ce projet est différente, car le champ d'application est plus restreint (pas un langage de programmation) et l'accent est mis sur l'optimisation de l'expression, et non sur l'exécution.

## 3 Mise en œuvre

La méthodologie utilisée pour ce projet est la suivante **1)** analyser le texte et créer un arbre binaire **2)** cet arbre est ensuite parcouru à l'aide d'un algorithme d'exploration gourmande.

Cet algorithme est un algorithme gourmand qui choisit entre une simple réduction de valeur où le sous-arbre est évalué. Si les valeurs à gauche et à droite de l'opérateur sont identiques ou si l'une des valeurs est un booléen, les valeurs (tautologie et contradiction) sont simplifiées à l'aide des lois de la logique. En outre, si les cas précédents ne sont pas vrais, l'expression est alors évaluée à l'aide de la loi d'absorption.

Voir la loi idempotence, la loi de domination, et la loi d'absorption. Le code peut être trouvé ici : Résolveur d'expressions logiques.

### 3.1 Infrastructure matérielle

L'ensemble du code de ce projet utilise le langage C avec les bibliothèques *<stdbool.h>*, *<stdio.h>*, *<stddef.h>* et *<string.h>* de la bibliothèque C standard. En outre, à des fins de débogage, le fichier *<assert.h>*, qui fait également partie de la bibliothèque standard C, est également utilisé.

Le projet est compilé à l'aide de GCC et Makefile afin d'automatiser le processus de construction et de test.

#### 3.1.1 Architecture de code

L'architecture du code est libre au sens où aucune structure ne lui a été attribuée lors de sa construction, à l'exception de la séparation des préoccupations. Cela signifie qu'une architecture modulaire, bien que non utilisée, pourrait facilement être ajoutée à l'avenir au système, puisque les fichiers sont découplés d'eux-mêmes et ont une bonne cohésion interne.

Fichier Code:

- main.c (101 lignes)
- node.c (438 lignes)
- parser.c (206 lignes)
- solver.c (489 lignes)

#### 3.1.1.1 Main.c

C'est l'« Orchestrateur » de *node.c*, *parser.c*, et *solver.c*, permettant ainsi une architecture plus découplée et un code plus facile à maintenir. Il a deux responsabilités : **1)** il prend en argument une chaîne d'utilisateurs à partir de la ligne de commande et la passe à *parser.c* pour créer un arbre, puis à *solver.c* pour résoudre l'expression ou **2)** il exécute certains tests prédéfinis et, avec les mêmes procédures qu'au point 1, essaie de résoudre chaque test.

#### 3.1.1.2 Node.c

Le fichier *node.c* est une bibliothèque utilitaire de première partie, ce qui signifie que tout le code gérant les subtilités de la gestion de l'arbre se trouve dans ce fichier. En outre, en raison d'un engagement précoce en faveur de la séparation des préoccupations, aucun code spécifique à un problème ou à un domaine ne se trouve dans ce fichier, ce qui rend les fonctions agnostiques par rapport au problème en question.

Par exemple, le fichier contient une logique pour la création d'un arbre et la suppression récursive d'un sous-arbre.

Pour trouver le code s'attaquant aux problèmes spécifiques du projet, le code peut être trouvé dans les fichiers *parser.c* et *solver.c*.

#### 3.1.1.3 Parser.c

Avec l'aide de la bibliothèque utilitaire *node.c*, *parser.c* gère la construction d'un arbre binaire basé sur la chaîne de caractères entrée par l'utilisateur. La technique utilisée pour l'implémentation de l'analyseur est l'utilisation d'un **automate** défini dans le Chapitre 3.2.2 (voir page 4); à la place d'un état, l'implémentation utilise un **ENUM** pour tous les états possibles et utilise une instance pour garder une trace de l'état actuel.

#### 3.1.1.4 Solver.c

Après avoir été analysé en un arbre, l'orchestrateur transmet l'arbre à *solver.c* qui, à son tour, parcourt l'arbre de manière récursive. En parcourant l'arbre, il recherche des modèles de simplification précalculés et, lorsqu'il en trouve, exécute une opération de simplification en utilisant les lois de la **logique propositionnelle** et l'aide de la bibliothèque utilitaire *node.c* pour manipuler l'arbre.

## 3.2 Analyseur syntaxique

### 3.2.1 Lexème

Le lexème est la première étape du pipeline du compilateur. Le lexème convertit simplement une chaîne de caractères en entrée (tableau de caractères) en un tableau ENUM. Cela rend le compilateur plus efficace en réduisant les comparaisons de chaînes et en simplifiant le flux de travail du développement.

Le lexateur transforme les valeurs de la chaîne en options de l'énumération suivante :

- **VAR** : Il s'agit de l'opérateur de variable. Les variables ne peuvent avoir que la taille d'un seul caractère.
- **NOT** : Il s'agit de l'opérateur **négation** et il est représenté syntaxiquement par ¬ dans les expressions logiques, mais pour le compilateur il est représenté par **!**.
- **AND** : Il s'agit de l'opérateur **et** : la représentation logique est ∧ et la représentation utilisée par le compilateur est **&**.
- **OR** : C'est l'opérateur **ou** : la représentation logique est ∨ et la représentation utilisée dans le compilateur est **|**.
- **THEN** : C'est l'opérateur **donc** : la représentation logique est ⇒ et la représentation utilisée dans le compilateur est **>**.
- **BTHEN** : Il s'agit de l'opérateur **donc bidirectionnel** : la représentation logique est ⇔ et la représentation utilisée dans le compilateur est **~**.

- **OPEN** : Il s'agit de l'opérateur **parenthèse ouverte** : la représentation logique est ( et la représentation utilisée dans le compilateur est **(**.
- **CLOSE** : Il s'agit de l'opérateur **fermer les parenthèses** : la représentation logique est ) et la représentation utilisée dans le compilateur est **)**.

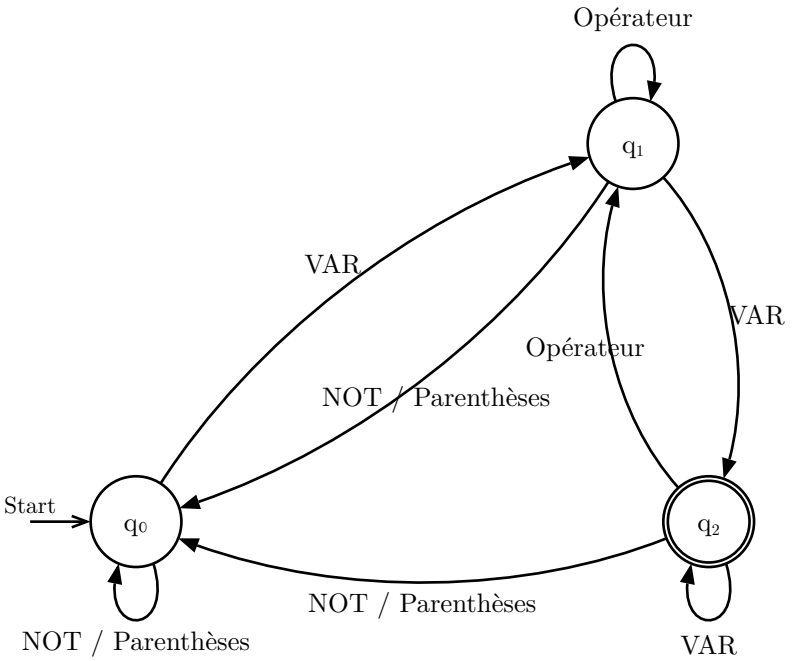
Ainsi, une expression valide pour le **lexème** serait : **`(p & ( q | r ) > s ~ w)`**, qui, en syntaxe logique normale, se traduit par : $p \wedge (q \vee r) \Rightarrow s \Leftrightarrow w$.

### 3.2.2 Bâtisseur d'arbre

L'analyseur syntaxique est la deuxième étape du pipeline du compilateur. L'analyseur syntaxique prend en entrée le tableau d'ENUM créé par l'étape précédente (lexème). La sortie est un arbre binaire, donc un opérateur/variable doit être ajouté à la GAUCHE ou à la DROITE. En outre, lorsque l'on échappe à la portée d'un opérateur ou d'une parenthèse, en raison de la préséance de l'opérateur ou de la parenthèse de fin, l'état de AU-DESSUS doit également être pris en compte.

**Préséance de l'opérateur** : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

L'implémentation de la machine à états finis (trouvée ici) peut être vue dans la figure suivante. Comme les expressions logiques sont écrites de gauche à droite avec un opérateur au milieu, l'implémentation va de la GAUCHE, puis AU-DESSUS, puis à DROITE. Si l'expression ne comporte que 3 parties (2 variables et un opérateur), l'analyseur est terminé, mais s'il y en a plus, l'analyseur, en fonction de la priorité de l'opérateur et des parenthèses, soit **1)** insère AU-DESSUS de l'opérateur actuel, soit **2)** déplace la droite de l'opérateur actuel vers la gauche de ce nouveau nœud et s'insère sur cette droite. Si, par exemple, une parenthèse est introduite pendant l'analyse, l'état se déplace vers la GAUCHE tout en sauvegardant le pointeur de la parenthèse sur une pile et, lorsqu'il « remonte vers la racine » (l'analyseur se déplace vers le parent du nœud actuel), l'analyseur fait référence à la pile, puis reprend là où il s'est arrêté.




L'analyseur utilise les états $q_0$ comme GAUCHE, $q_1$ comme DROITE, et $q_2$ comme AU-DESSUS.

## 3.3 Simplicateur d'arbre logique

### 3.3.1 Le déroulement de la phase d'optimisation

1. L'interprète commence par se déplacer vers le nœud situé sous la racine de l'arbre.

- **NOTA:** Si le nœud est simplement une variable, il n'y a rien à faire et donc renvoie.

2. L'interprète est à l'affût des éléments suivants :
   1. La loi d'**absorption** ; elle est prioritaire, car elle permet d'éliminer une variable/sous-expression sans l'évaluer (ex: le **q** dans l'expression $\boldsymbol{p \vee (p \wedge q)}$).
   2. la loi d'**idempotence**, qui élimine les doublons
   3. Les lois d'**identité** et de **dominance**
3. L'interpréteur n'évalue pas les non-opérateurs comme cible de simplification, donc si la droite/gauche est un opérateur, alors l'interpréteur simplifie récursivement cet opérateur sous forme d'arbre.
4. **NOTA:** L'implémentation actuelle renvoie simplement lorsqu'elle ne trouve pas d'optimisation, elle ne pourrait donc pas réévaluer le nœud en cas de changement en utilisant les lois de distribution ou de DeMorgan.

### 3.3.2 Formalisation de la logique du premier ordre

Comme le projet utilise un arbre binaire pour représenter l'expression, les règles internes pour résoudre les lois suivantes consistent à remplacer l'opérateur par une GAUCHE, une DROITE, une CONTRADICTION ou une TAUTOLOGIE en fonction de l'état de la GAUCHE et de la DROITE. Ce qui suit est une implémentation logique de premier ordre des règles utilisées pour simplifier les expressions.

**Définition logique** Soit l'univers de discours **KB**. Nous définissons les prédicats et les fonctions suivants :

- estNoeud(x) : $x$ est un noeud.
- estVar(x) : $x$ est un noeud variable.
- estNot(x) : $x$ est un noeud NOT.
- estAnd(x) : $x$ est un noeud AND.
- estOr(x) : $x$ est un noeud OR.
- estThen(x) : $x$ est un noeud THEN.
- estBThen(x) : $x$ est un noeud BTHEN.
- Valeur(x) : Une fonction qui renvoie la valeur du noeud $x$ (supposée être définie si $x$ est une variable).
- ValeurEquiv($v_1$, $v_2$) : Une fonction qui renvoie la valeur du nœud $x$ (supposée définie si $x$ est une variable) : La chaîne $v_1$ est équivalente à la chaîne $v_2$.
- InsertionGauche($N_p$, $N_e$) : Insère $N_e$ à la gauche de $N_p$.
- InsertionDroite($N_p$, $N_e$) : Insère $N_e$ à la droite de $N_p$.
- InsertionAuDessus($N_p$, $N_e$) : Insère $N_p$ entre $N_e$ et son ancien parent.
- SupprimerArbre(N) : Insère $N_p$ entre $N_e$ et son ancien parent : Supprime l'arbre de manière récurrente.

$$\begin{aligned}\text{EquivalentNoeud}(N_0, N_1) \Leftrightarrow\ & \text{estNoeud}(N_0) \wedge \text{estNoeud}(N_1) \wedge \\ & (\\ & (\text{estVar}(N_0) \wedge \text{estVar}(N_1) \wedge \text{ValeurEquiv}(\text{Valeur}(N_0), \text{Valeur}(N_1))) \\ & \vee (\text{estAnd}(N_0) \wedge \text{estAnd}(N_1)) \\ & \vee (\text{estOr}(N_0) \wedge \text{estOr}(N_1)) \\ & \vee (\text{estThen}(N_0) \wedge \text{estThen}(N_1)) \\ & \vee (\text{estBThen}(N_0) \wedge \text{estBThen}(N_1))) \\ & )\end{aligned}$$

### 3.3.3 La loi d'absorption

CODE

Définition:

$$p \wedge (p \vee q) \equiv p \vee (p \wedge q) \equiv p$$

$$\begin{aligned}\text{ABSORPTION\_DIRECTIONNELLE}(N, N_0, N_1) \Leftrightarrow (&(\text{EquivalentNoeud}(N_0, N_{1D}) \\ &\vee \text{EquivalentNoeud}(N_0, N_{1G})) \\ &\Rightarrow (\text{InsertionAuDessus}(N_0, N) \\ &\wedge \text{SupprimerArbre}(N)))\end{aligned}$$

$$\begin{aligned}\text{ABSORPTION}(N) \Leftrightarrow (&\text{estNoeud}(N) \wedge (\text{estOr}(N) \vee \text{estAnd}(N)) \\ &\wedge (\text{ABSORPTION\_DIRECTIONNELLE}(N, N_G, N_D) \\ &\vee \text{ABSORPTION\_DIRECTIONNELLE}(N, N_D, N_G)))\end{aligned}$$

La loi d'absorption est une loi pour 2 mêmes expressions, mais l'une étant l'enfant d'un enfant, ce qui permet de très bonnes optimisations puisque l'un des sous-arbres n'a pas besoin d'être évalué.

### 3.3.4 La loi d'idempotence

CODE

Définition:

$$p \wedge p \equiv p \vee p \equiv p$$

$$\begin{aligned}\text{IDEMPOTENCE}(N) \Leftrightarrow (&(\text{estNoeud}(N) \wedge (\text{estAnd}(N) \vee \text{estOr}(N)) \\ &\wedge \text{estNoeud}(N_G) \wedge \text{estNoeud}(N_D) \\ &\wedge \text{EquivalentNoeud}(N_G, N_D)) \\ &\Rightarrow (\text{InsertionAuDessus}(N_G, N) \wedge \text{SupprimerArbre:}(N)))\end{aligned}$$

La loi d'identité peut être décrite comme la loi des interactions entre deux mêmes sous-expressions/ variables pour les opérateurs AND et OR.

### 3.3.5 Les lois d'identité

CODE

Définition:

$$p \wedge \neg p \equiv F$$

$$p \vee \neg p \equiv T$$

$$\begin{aligned}\text{IDENTITE}(N) \Leftrightarrow (&\text{estNoeud}(N) \wedge (\text{estAnd}(N) \vee \text{estOr}(N)) \\ &\wedge ((\text{estNot}(N_G) \wedge \neg\ \text{estNot}(N_D) \wedge \text{EquivalentNoeud}(N_G, N_{\text{DG}})) \\ &\vee (\text{estNot}(N_D) \wedge \neg\ \text{estNot}(N_G) \wedge \text{EquivalentNoeud}(N_D, N_{\text{GG}}))) \\ &\Rightarrow \text{estOr}(N)) \Rightarrow (\text{InsertionAuDessus}(T, N) \wedge \text{SupprimerArbre}(N))\end{aligned}$$

La loi d'identité est une loi qui s'applique aux interactions entre deux sous-expressions/variables identiques dont l'une est négative.

La loi d'identité est une loi qui 1) s'applique lorsque les DEUX arguments sont les mêmes, l'un d'entre eux étant NÉGATIF, et 2) ne s'applique qu'à AND et OR. Bien que cette loi soit limitée aux types d'opérateurs précédents, le programme fonctionne également avec l'autre opérateur (preuve commentée dans les commentaires du code).

### 3.3.6 Les lois de dominance

CODE

Définition:

$$p \wedge T \equiv p$$

$$p \wedge F \equiv F$$

$$p \vee F \equiv p$$

$$p \vee T \equiv T$$

$$\begin{aligned}
\text{DOMINANCE}(N) \Leftrightarrow (\!| & \text{estNoeud}(N) \\
& \wedge ((\text{estOr}(N) \wedge (( \\
& \quad \wedge (\text{EquivalentNoeud}(T, N_G) \vee \text{EquivalentNoeud}(T, N_D)) \\
& \quad \Rightarrow (\text{InsertionAuDessus}(T, N) \wedge \text{SupprimerArbre}(N))) \\
& \quad \vee ( \\
& \quad \wedge (\text{EquivalentNoeud}(F, N_G) \Rightarrow \text{InsertionAuDessus}(N_D, N)) \\
& \quad \vee (\text{EquivalentNoeud}(F, N_D) \Rightarrow \text{InsertionAuDessus}(N_G, N)) \\
& \quad \wedge \text{SupprimerArbre}(N)))) \\
& \vee (\text{estAnd}(N) \wedge (( \\
& \quad \wedge (\text{EquivalentNoeud}(F, N_G) \vee \text{EquivalentNoeud}(F, N_D)) \\
& \quad \Rightarrow (\text{InsertionAuDessus}(F, N) \wedge \text{SupprimerArbre}(N))) \\
& \quad \vee ( \\
& \quad \wedge (\text{EquivalentNoeud } (T, N_G) \Rightarrow \text{InsertionAuDessus}(N_R, N)) \\
& \quad \vee (\text{EquivalentNoeud}(T, N_D) \Rightarrow \text{InsertionAuDessus}(N_G, N)) \\
& \quad \wedge \text{SupprimerArbre}(N)))) \\
|\!) &
\end{aligned}$$

La loi de dominance est la loi des interactions entre les *TAUTOLOGIES* et les *CONTRADICTIONS* avec d'autres sous-expressions ou variables.

Pour réaliser une simplification, le programme **1)** examine l'opérateur **TYPE** du sous-arbre et **2)** détermine si le booléen en question est une *TAUTOLOGIE* ou une *CONTRADICTION*. Dans la logique de premier ordre ci-dessus, seuls *AND* et *OR* sont implémentés (conformément à la définition de la loi), mais dans l'implémentation réelle, *THEN* et *BTHEN* sont également implémentés.

# 4 Résultats

Les résultats du système commencent par l'impression d'une version de l'arbre sans optimisation, puis d'une version de l'arbre avec les optimisations. Ceci est à des fins de débogage et pour montrer une interprétation visuelle de l'expression à l'utilisateur. En outre, lors de l'exécution des tests, le système vérifie si la valeur attendue est égale à la sortie et affiche un message de SUCCÈS ou d'ERREUR. Exemple de résultats de **« ( p & ( q | r ) ) | p »** (équivalent à « $(p \wedge (q \vee p)) \vee p$ ») :

```
---- DÉBUT DU TEST ----

entrée: '( p & ( q | r ) ) | p'

-- DÉBUT DE L'IMPRESSION INITIALE DE L'ARBRE --
6(TRUE)
|>3(OR)
 |>2(AND)
  |>0(p)
  |>3(OR)
   |>0(q)
   |>0(r)
 |>0(p)
-- FIN DE L'IMPRESSION INITIALE DE L'ARBRE --
```

```
--- DÉBUT DES OPTIMISATIONS ---
L'arbre a été réduit avec SUCCÈS.
-- DÉBUT DE L'IMPRESSION DE LA VERSION OPTIMISÉE --
6(TRUE)
|>0(p)
-- FIN DE L'IMPRESSION DE LA VERSION OPTIMISÉE --
--- FIN DES OPTIMISATIONS ---
--- L'ÉVALUATION DES RÉSULTATS ATTENDUE ---

Chaîne de sortie: p
Chaîne de caractères attendue: p
SUCCÈS: `( p & ( q | r ) ) | p` a été optimisé à `p`.
```

La version chaîne de l'arbre affiche d'abord la valeur numérique de l'énumération, puis le nom associé à l'énumération entre parenthèses. Le symbole `|>` indique qu'il s'agit de l'enfant du nœud précédent.

### 4.0.1 Exemple d'échec

La première partie de l'exemple d'échec est la même que celle de l'exemple précédent. Voici donc un exemple du cas d'ERREUR pour « **p > ( q ~ p )** » (équivalent à « $p \Rightarrow (q \Leftrightarrow p)$ »)

```
...
Chaîne de sortie: ( ! p | ( ( ! q | p ) & ( ! p | q ) ) )
Chaîne de caractères attendue: ( p > q )
ERREUR: Attendue `( p > q )`, mais obtenue `( ! p | ( ( ! q | p ) & ( ! p | q ) ) )`.
```

Comme vu plus haut, l'interpréteur avait des difficultés à simplifier les expressions imbriquées. Le problème survient lorsque le solveur revient en arrière, ce qui fait que l'arbre n'est pas réévalué correctement, il s'agit d'un bogue connu qui a été abordé (et a échoué) à de nombreuses reprises au cours du développement. Ce problème n'est pas seulement un bogue, c'est un problème d'architecture. Le solveur n'est pas en mesure de redescendre de manière fiable dans l'arbre après l'avoir parcouru, ce qui limite le projet à de simples expressions.

# 5 Conclusion

En conclusion, ce projet a permis d'implémenter un interpréteur dans le but de simplifier l'expression de la logique propositionnelle. Ce projet pourrait éventuellement servir de préprocesseur et d'optimiseur d'exécution pour un agent. Pour de plus amples améliorations, l'optimiseur d'expression aurait besoin d'un meilleur algorithme pour les simplifications. Pour améliorer la fiabilité du système, des tests unitaires doivent être ajoutés, car seuls les tests de bout en bout figurent dans la base de code. En outre, certains bogues de longue date n'ont pas encore été corrigés. Pour améliorer le projet, il faut donc considérer que ces bogues sont le plus gros problème de l'implémentation actuelle.

# Bibliographie


[1] « Propositional Logic - Z3 Guide ». Consulté le: 24 avril 2025. [En ligne]. Disponible sur: https://microsoft.github.io/z3guide/docs/logic/propositional-logic

[2] A. Fröhlich, A. Biere, C. M. Wintersteiger, et Y. Hamadi, « Stochastic Local Search for Satisfiability Modulo Theories », in *Proceedings of AAAI*, AAAI, janv. 2015. [En ligne]. Disponible sur: https://www.microsoft.com/en-us/research/publication/stochastic-local-search-for-satisfiability-modulo-theories/

[3] A. Colmerauer, H. Kanoui, R. Pasero, et P. Roussel, « Un système de communication homme-machine en francais », *Rapport préliminaire, Groupe de Res. en Intell. Artif*, 1973, [En ligne]. Disponible sur: http://alain.colmerauer.free.fr/alcol/ArchivesPublications/HommeMachineFr/HoMa.pdf

[4] « SWI-Prolog Manual ». Consulté le: 24 avril 2025. [En ligne]. Disponible sur: https://www.swi-prolog.org/pldoc/doc_for?object=manual