Université Pierre et Marie Curie

Projet PSAR
Une interface graphique pour la logique

Auteurs :
Bastien RIGAULT
Sandra LADURANTI

Référents :
Béatrice BERARD
Mathieu JAUME
Bénédicte LEGASTELOIS

26 avril 2016

Résumé

L'objectif de ce projet, réalisé dans le cadre de l'UE PSAR (4I408) est de concevoir un logiciel pédagogique pour l'apprentissage de la logique du premier ordre. Ce logiciel apportera notamment un support visuel sous la forme d'un jardin et de fleurs permettant d'appréhender plus facilement les formules de la logique. Les étudiants auront à leur disposition deux fenêtres : l'une permettant de concevoir un jardin en y positionnant des fleurs et l'autre permettant d'écrire des formules à l'aide d'un clavier visuel. Ils pourront alors soit construire un jardin respectant un ensemble de formule données, ou au contraire établir des formules à partir d'un jardin donné. Les formules pourront être analysées de manière automatique afin de vérifier leur cohérence. Enfin, les étudiants pourront sauvegarder leurs jardins et leurs formules pour les réutiliser ultérieurement.
La première partie de ce rapport présente plus amplement le sujet, les outils mis à disposition pour le réaliser et les premières pistes pour résoudre les problématiques liées à leurs utilisation. La seconde partie explique concrètement quels vont être les fonctionnalités à développer ainsi que l'organisation du travail dans le groupe et la structure général du code. La partie suivante explore plus amplement les choix techniques qui ont été réalisé, leurs avantages et inconvénients ainsi que leurs utilisation et rôle dans la réalisation du projet. Enfin, ce rapport se conclu sur une analyse des résultats obtenu en comparaison des attentes formulées dans le cahier des charges.

Table des matières

Chapitre 1

Présentation du projet

1.1 Sujet

Le sujet ne peut être abordé sans un rapide point sur les différents aspects de la logique du premier ordre telle qu'elle sera utilisée dans le projet :

1.1.1 Introduction à la logique du premier ordre

Une formule de la logique des prédicats, telle qu'elle sera écrite par les étudiants, est constituée de termes, de prédicats, de quantificateurs et de connecteurs logiques. Prenons par exemple la formule suivante :

$$\forall x Rose(x) \Rightarrow estRouge(x)$$

Pour écrire une telle formule, l'étudiant utilise des termes comme briques de base. Ces termes correspondent soit à l'ensemble $F_0 = \{a, b, ..., t\}$ des constantes, soit à l'ensemble $X = \{v, w, x, y, z\}$ des variables.
Ensuite, il dispose également d'un ensemble de prédicat P_i d'arité i, pour $i \in \{1, 2, 3\}$. Chaque prédicat de P_1, P_2, P_3 prend ainsi respectivement 1, 2 ou 3 termes en argument. Dans notre exemple, $Rose(x)$ et $estRouge(x)$ sont deux prédicats de P_1 et prennent la variable x en argument.
À partir de ces formules de base, l'ensemble des formules est défini inductivement : si φ est une formule alors $\neg\varphi$ est également une formule. De même si φ_1 et φ_2 sont des formules alors $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ et $\varphi_1 \Rightarrow \varphi_2$ aussi. Enfin, si $x \in X$ et φ est une formule alors $\forall x\varphi$ et $\exists x\varphi$ aussi.
Ce projet utilise un ensemble de prédicat prédéfinis que l'on regroupera par arité :

Unaire définissant l'état d'une fleur, elle peut être :
- une rose, une paquette ou une tulipe
- petite, moyenne ou grande
- rouge, rose ou blanche
- à l'est, à l'ouest, au sud ou au nord

Binaire définissant une relation entre 2 fleurs, la fleur a peut être :
- à l'est, à l'ouest, au sud ou au nord de la fleur b
- à la même latitude ou longitude que la fleur b
- plus grande, plus petite ou de même taille que la fleur b
- de la même couleur que la fleur b

Ternaire définissant une relation entre 3 fleurs, la fleur c peut être :
- entre la fleur a et la fleur b

La section suivante présente quelques exemples de formules utilisant différents prédicats.

1.1.2 Exemple

On peut par exemple écrire les huit formules ci-dessous.

(f1) d est une rose : Rose(d)
(f2) Toutes les fleurs sont des roses : $\forall x Rose(x)$
(f3) Il existe une rose : $\exists x Rose(x)$
(f4) Toute fleur blanche est plus petite qu'au moins une fleur située à son est :
$\forall x(est\_blanc(x) \Rightarrow \exists y(plus\_petit\_que(x,y) \wedge a\_l\_est\_de(y,x)))$
(f5) Toute fleur est à l'est, ou à l'ouest, ou au sud, ou au nord :
$\forall x(a\_l\_est(x) \vee a\_l\_ouest(x) \vee au\_sud(x) \vee au\_nord(x))$
(f6) Toutes les grandes fleurs sont rouges et il n'existe pas de fleur blanche au sud d'une fleur rouge :
$\forall x(est\_grand(x) \Rightarrow est\_rouge(x)) \wedge \neg\exists x(est\_blanc(x) \wedge \exists y(est\_rouge(y) \wedge au\_sud\_de(x,y)))$
(f7) Il existe une fleur rouge au nord de la fleur g : $\exists x(est\_rouge(x) \wedge au\_nord\_de(x,g))$
(f8) Il existe une unique rose rouge :
$\exists x((Rose(x) \wedge est\_rouge(x)) \wedge (\forall y(Rose(y) \wedge est\_rouge(y)) \Rightarrow x \stackrel{\circ}{=} y))$

Les exemples ci-dessus seront donc appliqués dans un jardin tel que présenté dans la section suivante.

1.1.3 Application de la logique dans le jardin

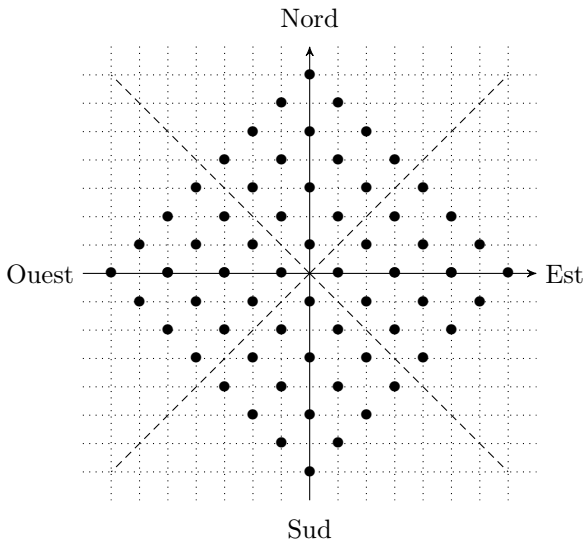

FIGURE 1.1 – Places d'un jardin

Les termes considérés dans le projet désignent des fleurs positionnées sur une grille. La figure 1 représente la grille : les points noirs symbolisent les places où peuvent être disposées les fleurs (il y a au plus une fleur par place). Chaque place est identifiée par ses coordonnées (le point $(0,0)$ est au centre de la grille). L'ensemble des places possibles est donc :

$$\text{Places} = \left\{ \begin{array}{c} (0,7), \\ (-1,6),(1,6), \\ (-2,5),(0,5),(2,5), \\ (-3,4),(-1,4),(1,4),(3,4), \\ (-4,3),(-2,3),(0,3),(2,3),(4,3), \\ (-5,2),(-3,2),(-1,2),(1,2),(3,2),(5,2), \\ (-6,1),(-4,1),(-2,1),(0,1),(2,1),(4,1),(6,1), \\ (-7,0),(-5,0),(-3,0),(-1,0),(1,0),(3,0),(5,0),(7,0), \\ (-6,-1),(-4,-1),(-2,-1),(0,-1),(2,-1),(4,-1),(6,-1), \\ (-5,-2),(-3,-2),(-1,-2),(1,-2),(3,-2),(5,-2), \\ (-4,-3),(-2,-3),(0,-3),(2,-3),(4,-3), \\ (-3,-4),(-1,-4),(1,-4),(3,-4), \\ (-2,-5),(0,-5),(2,-5), \\ (-1,-6),(1,-6), \\ (0,-7) \end{array} \right\}$$

Chaque fleur appartient à une espèce (les roses, les pâquerettes et les tulipes), est d'une certaine taille (grande, moyenne ou petite) et d'une certaine couleur (rouge, rose, blanche).

$$\begin{aligned} Especes &= \{rose, paquerette, tulipe\}, \\ Tailles &= \{grand, moyen, petit\}, \\ Couleurs &= \{rouge, rose, blanc\}. \end{aligned}$$

Certaines fleurs ont un nom : il s'agit d'une constante de F_0 (deux fleurs différentes ne peuvent pas avoir le même nom). Un jardin j est la donnée d'une grille sur laquelle sont disposées des fleurs (chaque jardin contient au moins une fleur) et est représentée par une liste de quintuplets. Chaque quintuplet $((x,y),e,t,c,n) \in j$ est un élément du produit cartésien :

$$Places \times Especes \times Tailles \times Couleurs \times (F_0 \cup \{None\})$$

et exprime qu'une fleur de nom n (n = None si la fleur n'a pas de nom), d'espèce e, de taille t et de couleur c se trouve à la place (x,y) dans le jardin j. L'ensemble des jardins possibles est donc :

$$J = \wp(Places \times Especes \times Tailles \times Couleurs \times (F_0 \cup \{None\}) \setminus \{\emptyset\}$$

1.2 Existant : évaluation des formules

Le projet tourne autour d'un script préalablement fourni en python.

1.2.1 Script python

Le script permet de créer un jardin en disposant les différents éléments avec leurs attributs sur diverses coordonnées. En premier lieu il est nécessaire de bien créer un jardin, il serait inutile de tester une formule sans contexte autour. Le script permet ensuite de créer une formule, celle-ci doit être préalablement vérifiée pour que sa syntaxe ne contienne aucune erreur. Dans un dernier temps le script analyse la formule dans le contexte du jardin donné et retourne un booléen en résultat.

1.2.2 Bilan récapitulatif

En résumé ce script python permet de :
— Construire un jardin en ligne de code
— Construire une formule en ligne de code
— Analyser si une formule est cohérente pour un jardin donnée et inversement.
Les enjeux majeurs de ce projet sont alors de pouvoir :
— Construire un jardin graphiquement
— Construire une formule graphiquement (avec un clavier virtuel)
— Vérifier qu'une formule est syntaxiquement correcte
— Faire communiquer les points ci-dessus avec le script python

1.3 Problématiques soulevées

Deux problématiques importantes sont alors soulevées pour la réalisation de ce projet.
— La récupération et vérification de l'intégrité syntaxique des formules entrées dans le programme avant de les envoyer au script de vérification de leur véracité dans un jardin donné ;
— La communication entre deux langages différents et une interface graphique.

1.4 Hypothèse de solution

1.4.1 Problématique de l'intégrité syntaxique

La solution envisagée est l'utilisation d'un générateur d'analyseur syntaxique, idéalement via l'outil Grammatica version 1.6 (libre de droit sous licence BSD). Après que l'outil ait reçu une grammaire spécifique en entrée, il produit un parseur pour cette grammaire en C#. Ce code n'est généré qu'une seule fois et est capable d'analyser des formules données sous forme de chaînes de caractères. Le résultat final obtenu est soit un message d'erreur détaillé si la formule est incorrecte soit un arbre syntaxique représenté en C# comme l'illustre la figure 1.2.

1.4.2 Problématique de la communication

Le sujet du projet est fourni avec un script de vérification des formules. Il est donc important de pouvoir faire communiquer ce script avec le langage de développement choisi : C#, ainsi qu'avec les briques fournies par l'API de Unity. Ainsi la communication symbolise la brique liante entre la partie syntaxique, le script de vérification, et les entrées utilisateur via l'interface graphique.

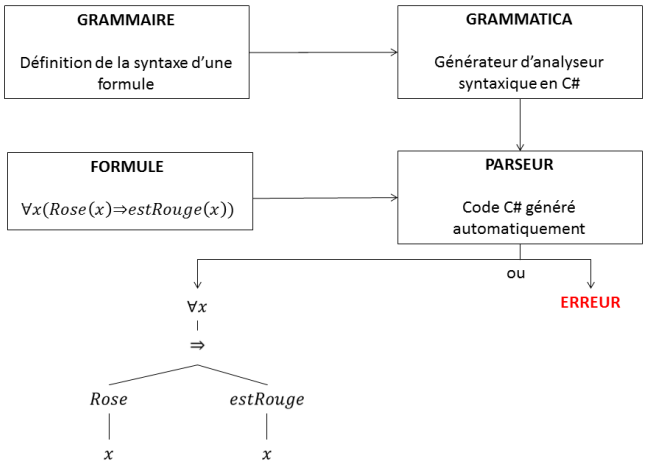

FIGURE 1.2 – Analyse syntaxique

Chapitre 2

Analyse des besoins

Le projet s'axe sur le besoin fonctionnel de base de l'interface graphique autour duquel gravitent des besoins non fonctionnels mais nécessaires au bon fonctionnement de l'application.

2.1 Besoins

Après une analyse des besoins du projet, nous avons défini deux sous catégories. D'un côté, les besoins graphiques, de l'autre, les besoins liés à la syntaxe des formules.

2.1.1 Interface Graphique

L'interface graphique doit être ergonomique et *user-friendly*, le but est d'accompagner les étudiants dans l'apprentissage de la logique du premier ordre de la manière la plus agréable possible. Pour cela un certain nombre d'exigences en terme de GUI ont été mis en place :

- un jardin représenté par une grille de taille fixe ;
- des fleurs pouvant être disposées sur le jardin et ayant un visuel différent selon leurs espèces, tailles et couleurs ;
- un menu permettant de positionner de nouvelles fleurs à la souris et d'en changer les caractéristiques ;
- un menu permettant d'écrire et vérifier des formules de la logique du premier ordre à l'aide d'un clavier virtuel et d'une liste de 5 variables et de 20 constantes ;
- un menu permettant de sélectionner une variable ou une constante ;
- un clavier visuel permettant de sélectionner les connecteurs de la logique (not, et, ou, pour tout, etc.) ;
- un menu permettant de sauvegarder et de charger des jardins et/ou des ensembles de formules.

Aperçu du rendu souhaité :

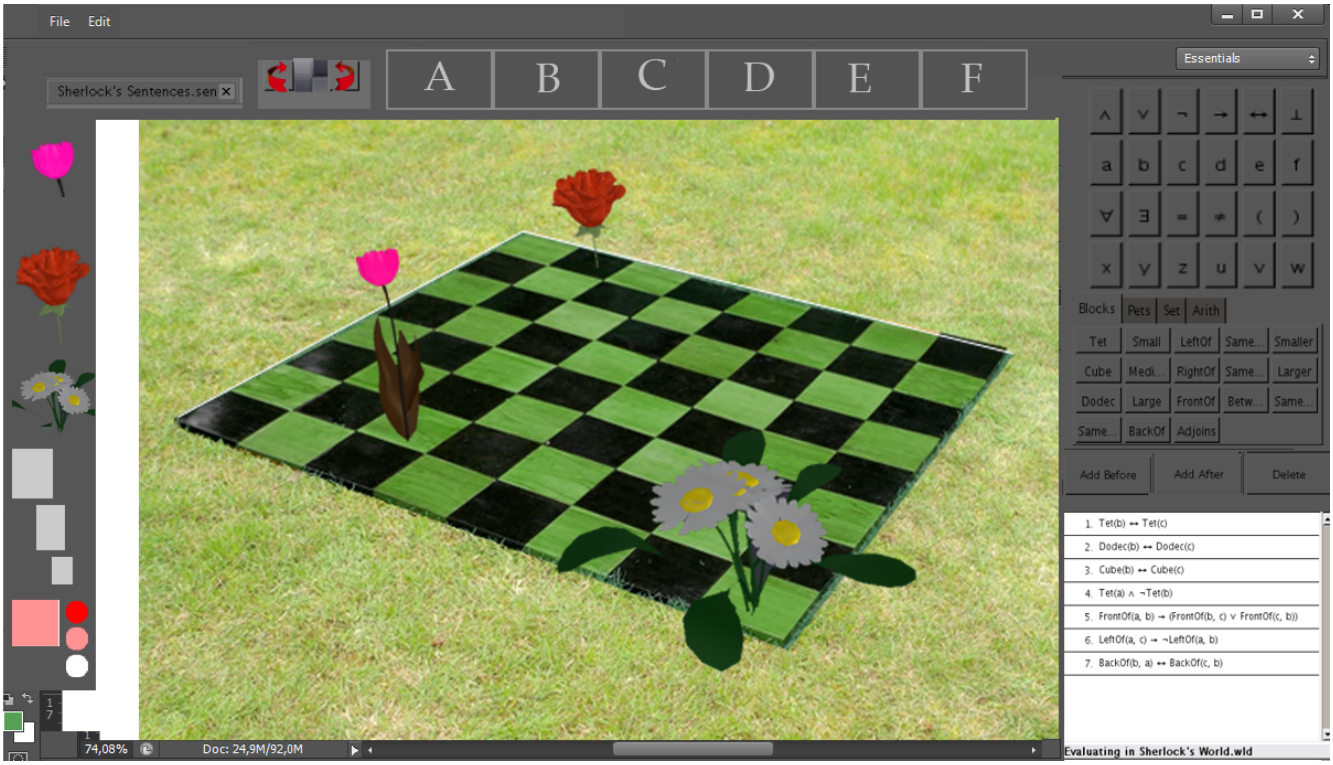

FIGURE 2.1 – Rendu attendu

2.1.2 Analyse Syntaxique

L'analyse syntaxique est un besoin lié directement au script fourni avec le sujet du projet. Si cette partie n'est pas parfaitement fonctionnelle, le script ne pourra donc pas fonctionner correctement et le projet ne pourra aboutir.
Idéalement, dans le cas d'une formule syntaxiquement fausse, l'analyseur pourra indiquer ou se trouve l'erreur dans la formule entrée par l'utilisateur. Cependant ce point n'est pas une nécessité, cette partie doit en priorité stopper le programme et notifier l'utilisateur du non respect de la syntaxe dans l'une de ses formules.

Il n'est utile d'aborder qu'un seul besoin non-fonctionnel : la communication entre deux langages différents, ce point étant technique nous préférons revenir dessus plus tard et seulement l'évoquer ici. Il a cependant été pris en compte les contraintes de développement, détaillées à la fin de cette partie.

2.2 Développement

Le développement est réparti en tâches parallélisées afin d'optimiser le temps de réalisation.

2.2.1 Tâches

Le développement s'axe sur trois grandes tâches, la communication inter langages, l'analyse syntaxique et la création de l'interface graphique. Le but étant de relier les trois blocs ensemble pour obtenir l'application finale.

Priorité	Nom	Raison
1	Communication inter langages	Doit être vérifié en premier car sinon on ne pourra pas utiliser l'Analyseur Syntaxique
2	Analyseur Syntaxique	On doit pouvoir entrer n'importe quel formule et être capable de détecter au plus vite la moindre erreur dans celle-ci
3	Liaison Communication-Analyseur	Comme les principales fonctionnalités permettant de tester sont opérationnelles, nous pouvons passer à cette tâche.
4	Création de l'interface graphique	Dernière fonctionnalité essentielle à mettre en place.
5	Animation des fleurs	Non-essentiel, mais apporterait un plus au projet.
6	Fonctionnalité de retour en arrière	Non-essentiel, mais apporterait un plus au projet.

FIGURE 2.2 – Tableau récapitulatif des tâches

2.2.2 Organisation du code

Dans le cas où l'analyseur syntaxique détecterait une erreur de formule, celui-ci ne retournera pas d'objets mais notifiera le bloc de communication de l'erreur. Ce dernier remontera donc le soucis au bloc GUI qui l'affichera à l'utilisateur.

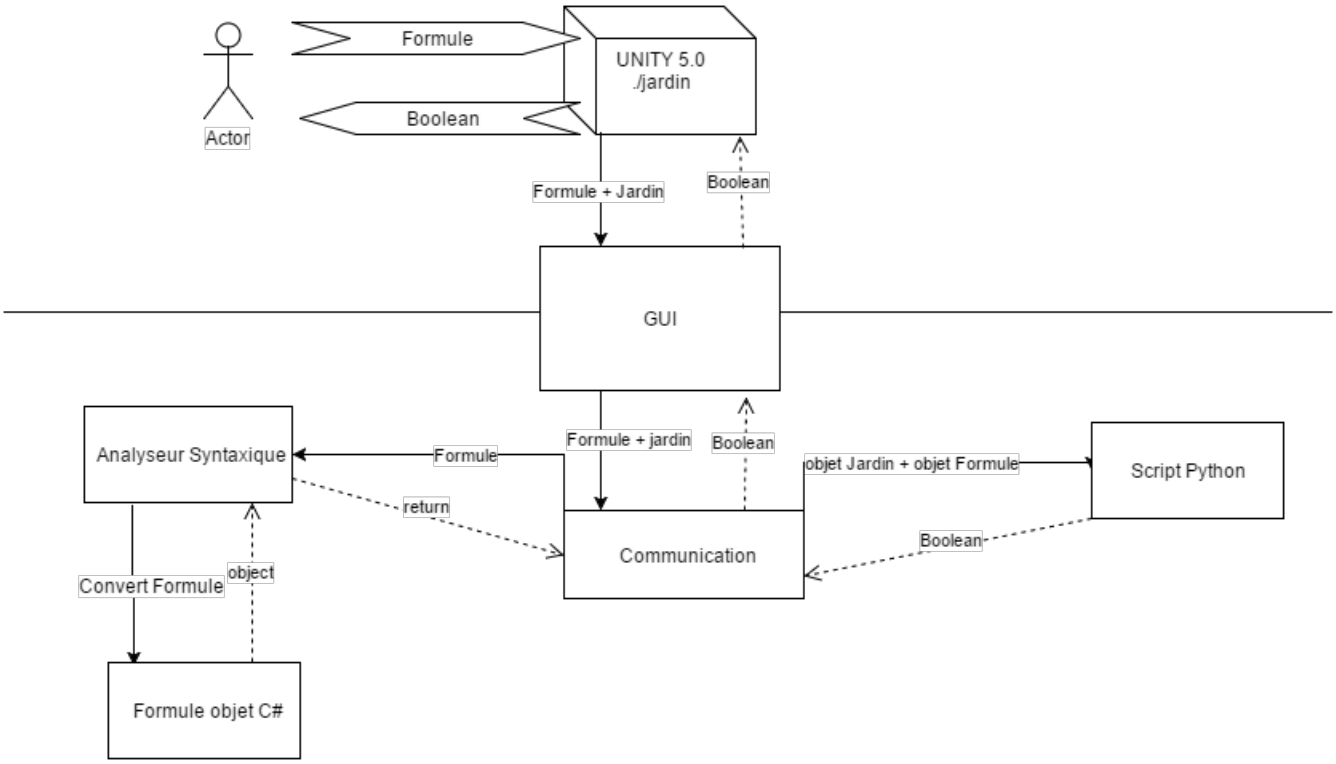

FIGURE 2.3 – organisation des blocs

Chapitre 3

Choix techniques

Dans cette partie nous cherchons à décrire dans un premier temps les divers choix techniques et spécificités du projets plus en détail.

3.1 Analyse Syntaxique

3.1.1 Génération du parseur avec Grammatica

Comme présenté dans la première partie, l'outil utilisé pour réaliser l'analyse syntaxique d'une formule est Grammatica 1.6. Ce programme écrit en Java est un générateur de parseur en Java et C#, autrement dit il prend en entrée un fichier texte décrivant les règles d'une syntaxe à suivre et produit 4 classes vers le langage cible. Ce fichier texte décrivant la grammaire (fournis en annexe) est constitué de deux blocs :

— Les tokens, ce sont toutes les chaînes de caractères qui seront acceptées par le parseur.

— La production qui est l'ensemble des règles qui articulent les tokens définis précédemment.

Les règles définies dans la production doivent être judicieusement écrites afin que la rédaction d'une formule soit suffisamment souple (éviter de forcer l'utilisateur à écrire trop de parenthèse inutile par exemple) mais respecte toujours les règles des formules de la logique. De plus Grammatica, comme beaucoup d'autre générateurs de parseur libres de droits, examine la chaîne de caractère de gauche à droite lors de l'analyse. Cette contrainte empêche d'écrire des règles directement récursives. Par exemple, bien qu'elle soit la plus intuitive on ne peut pas écrire une telle règle :

$$Formule \ = \ Formule \ ET \ Formule$$

car Grammatica bouclera lors de l'analyse. La première solution pour contourner le problème est d'introduire un caractère intermédiaire :

$$Formule \ = \ '(' \ Formule \ ET \ Formule \ ')'$$

Cette solution bien qu'efficace oblige l'utilisateur à écrire un nombre inutile de parenthèses dans certaines situations, par exemple la formule logique :

$$Rose(x) \wedge est\_grand(x) \wedge a\_l\_est(x)$$

devra s'écrire :

$$(Rose(x) \wedge (est\_grand(x) \wedge (a\_l\_est(x))))$$

Pour cela on définit une règle spécialement dédiée aux prédicats. Les prédicats étant des cas terminaux de notre récursion on peut écrire :

$$Formule \ = \ Predicat \ ET \ Predicat$$

Une fois que la grammaire est correctement définie, on lance Grammatica afin de générer les quatre classes C# suivantes :

- **LogicalConstants.cs** qui est une énumération de tous les tokens.
- **LogicalTokenizer.cs** qui fait l'association entre la chaîne de caractères d'un token et la représentation interne propre à Grammatica.
- **LogicalParser.cs** et **LogicalAnalyzer.cs** qui permettent d'effectuer l'analyse proprement dite d'une chaîne de caractères.

Ces quatre classes permettent d'obtenir un arbre syntaxique à partir d'une chaîne de caractères. Cependant cet arbre est représenté selon la bibliothèque de Grammatica et ne correspond pas exactement à la représentation dont on en a besoin. Il est donc nécessaire de parcourir une fois l'arbre afin de le reconstruire avec la représentation propre à notre projet, afin de pouvoir l'utiliser avec un jardin et vérifier l'intégrité de la formule. C'est le rôle des classes **Formule.cs** et **FormuleFactory.cs**. Pour résumer, l'analyse syntaxique d'une formule se fait comme suit :

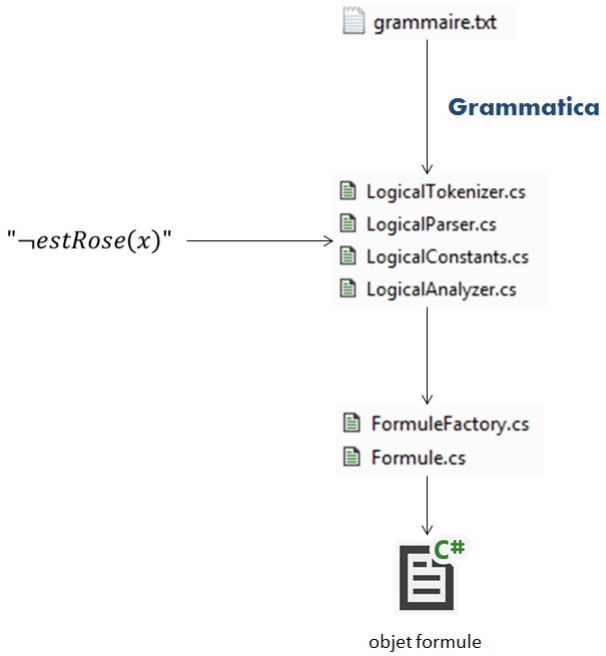

FIGURE 3.1 – Vérification syntaxique avec Grammatica

3.1.2 Communication avec le script Python

Comme évoqué précédemment, la brique de base du projet se séparait en deux problématiques distinctes, la grammaire et la communication inter blocs. Nous allons aborder ici ce second point.

Fonctionnement

La première problématique amenée par le projet était de pouvoir faire communiquer deux langages différents ensemble. Après quelques recherches, il s'est avéré qu'une implémentation de Python avait été réalisée spécifiquement pour le développement .NET et Mono, écrit en C# et donc parfaitement compatible avec Unity 5 : IronPython. Le but était de pouvoir embarquer la solution dans Unity afin que celui-ci puisse gérer par la suite la conversion du projet pour les plateformes Linux, Mac et Windows. IronPython permet l'utilisation de toute la bibliothèque Python mais avec les avantages du .Net, il est donc possible de faire des appels de fonction directement depuis le C# sans avoir à passer par des sockets ou autre connecteur réseau. De même il est également possible de passer des objets C# vers Python et inversement, il est alors possible de faire des appels de méthode sur les objets C# depuis le Python. Une fois le problème du choix de technologie réglé, il est temps de se questionner sur la manière dont la brique de communication doit agir. Dans un premier temps, elle est composée d'une méthode principale visée à être appelée par des événements placés sur l'interface utilisateur. Son but et de faire le lien entre les différentes étapes de vérification, en premier lieu le but est d'envoyer la formule à l'analyse syntaxique qui étudie si oui ou non, la formule peut être envoyée à la vérification par le script Python. Dans le cas où la formule est juste, la brique de communication récupère de l'analyseur la formule modifiée en objet C# et la renvoie dans une méthode intermédiaire récursive, visant à la retransformer pour qu'elle puisse être lue par le script Python, auquel elle est envoyée au final. La brique de communication fini son travail en retournant à l'interface utilisateur le résultat du traitement de la formule. Soit un booléen, soit un message d'erreur dans le cas où un problème serait survenu au cours du traitement (mauvaise syntaxe, objet inexistant dans le jardin de base, etc.)

Problèmes rencontrés

Lors de la réalisation du cahier des charges et de l'analyse de l'existant, il a été fourni aux deux groupes travaillant sur ce projet deux scripts distincts. Un script en Python et un script en Ocaml. Notre groupe a choisi de prendre le second. Le premier gros problème rencontré sur la partie communication a été de réussir à trouver comment faire une liaison entre du C# et de l'Ocaml. Une seule API fournissant la solution a été trouvée : CSML. Il s'est avéré que la solution n'était plus maintenue depuis plusieurs années et que le projet n'était pas fonctionnel sous des machines en 64 bit. Les nombreuses contraintes et bugs rencontrés lors des tests nous ont fait abandonner très vite cette solution. Il a alors été abordé l'idée de communiquer via Sockets, cependant nous avons rencontré des erreurs dans la documentation même d'Ocaml nous avons préféré abandonner totalement l'idée de continuer avec Ocaml et nous rabattre sur Python, plus simple à prendre en main rapidement et avec une communauté plus nombreuse derrière.

3.2 L'interface graphique avec Unity

L'interface graphique est le point central de ce projet. Aussi, le choix du bon langage et/ou du bon outil pour la concevoir fut un choix crucial pour mener à bien ce projet.

FIGURE 3.2 – Rendu final de l'interface graphique

3.2.1 Présentation et architecture générale

Notre choix s'est porté sur le moteur de jeux Unity 5 qui, grâce à de nombreux outils performants permet de créer une interface graphique élégante et robuste. Bien que Unity prenne en charge le temps réel, pour notre application nous avions seulement besoin d'utiliser la programmation par évènements. Le bloc qui effectue la liaison entre le code nécessaire à la vérification d'une formule avec un jardin et l'interface graphique en elle-même est donc constitué d'un ensemble de scripts. Ces scripts sont attachés à différents éléments de l'interface graphique (bouton, fleur, formulaire) et sont appelés lorsque l'utilisateur déclenche un évènement attendu. Un script, une fois activé, peut alors soit modifier directement un élément de l'interface graphique (par exemple déplacer ou modifier l'aspect d'une fleur) ou faire appel à un autre script encapsulant le code nécessaire au traitement d'une formule et d'un jardin (vérification, sauvegarde ou chargement de formules et de jardins). De plus on peut diviser les scripts en trois catégories :

— Les scripts qui n'interagissent qu'avec le clavier et les formulaires, s'occupant de faciliter la rédaction d'une ou plusieurs formules.
— Les scripts propres au jardin et aux fleurs permettent de positionner et modifier des fleurs dans le jardin.
— Les scripts liants les deux groupes précédents sont notamment les scripts attachés aux boutons 'Vérifier', 'Vérifier un', 'Sauvegarder', 'Charger' qui sont en charge de récupérer le jardin et les formules pour effectuer différents traitement.

L'architecture globale de l'interface graphique peut être schématisée comme suit :

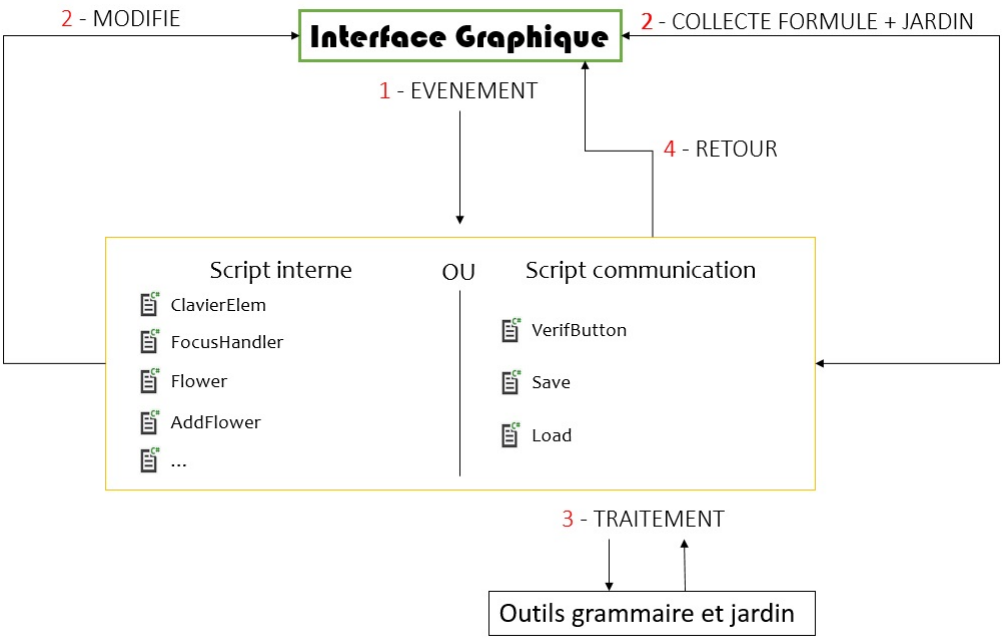

FIGURE 3.3 – Vérification syntaxique avec Grammatica

3.2.2 Fonctionnement du clavier et des formulaires

Pour écrire une formule l'utilisateur a 30 formulaires à sa disposition. Ces formulaires fonctionnent avec un clavier virtuel permettant à l'utilisateur de saisir les caractères spéciaux représentant les opérateurs ainsi que les constantes, variables et prédicats afin d'éviter les erreurs de syntaxe bête lors de la réalisation d'une formule. Bien que le fonctionnement du clavier puisse sembler trivial à première vue quelques problèmes furent posés lors de sa conception. En effet, quand l'utilisateur manipule le clavier, au-delà du fait d'ajouter le ou les caractère(s) au formulaire, il faut se souvenir quel formulaire a été sélectionné en dernier et pour chaque formulaire quelle était la position du curseur. Ceci est le rôle des scripts $CaretPos$ et $FocusHandler$. Chaque formulaire possède une instance de $CaretPos$ qui, dans une variable locale, sauvegarde chaque changement du curseur pour se formulaire. Le curseur peut aussi bien être déplacé par un clic de souris que par une édition. $FocusHandler$ au contraire n'est instancié qu'une seule fois dans un panel encapsulant l'ensemble des formulaires. Quand un formulaire est sélectionné à la souris, celui-ci se charge d'appeler une fonction de $FocusHandler$ pour indiquer qu'il est le nouveau formulaire actuellement sectionné. Ainsi, quand l'utilisateur appuie sur un des boutons du clavier virtuel, ce bouton demande au $FocusHandler$ à quel formulaire envoyer la chaîne de caractères. Le formulaire ajoute ensuite les caractères à la bonne position grâce à $CaretPos$.

3.2.3 Le jardin et les fleurs

Le soucis de cette partie était d'arriver à gérer trois comportements différents de l'application sur une même zone, à savoir l'ajout de fleur, la modification de fleurs et leur suppression. Comme évoqué plus haut, Unity offre une méthode *Update*, si celle-ci est vide, elle est ignorée par le moteur graphique. Cependant si elle est utilisée, elle est alors appelée une fois par *frame* et simule donc une boucle. Nous avons donc utilisé le plateau afin d'y placer un script de gestion des modes, vérifiant à chaque frame l'état de la boucle et la position de la souris. Récupérer l'objet que la souris survole est en général simple depuis l'objet lui même, Unity offrant une méthode toute faite. Cependant récupérer ce même objet depuis un autre script, non directement lié, est légèrement plus complexe. Dans ce cas il faut utiliser un *Raycast*, qui est un rayon envoyé depuis la souris sur une ligne droite dont l'angle et la distance max sont fixés dans la méthode offerte par le moteur de jeu. Tout objet n'appartenant pas à la classe d'interface graphique de Unity est alors retourné : c'est le cas des tuiles de notre jardin.
Les différents modes vont donc offrir un panel de fonctionnalités différent. Le principe de base étant le même pour l'ajout et la modification d'une fleur. Le but est de récupérer les différentes informations des boutons et de les enregistrer dans un objet. Ensuite la boucle principale analyse l'endroit que le pointeur vise et récupère l'objet en question si il fait partie du jardin. On demande alors à la case de créer une visualisation de la fleur que l'on veut. Dans le cas où l'utilisateur est d'accord sur l'emplacement et qu'il clique, on set alors la fleur à l'endroit voulu. Si jamais l'utilisateur préfère une case voisine, alors la visualisation est détruite et reconstruite dans la nouvelle case choisie.
Chaque ajout et chaque modification est enregistré dans le script du jardin destiné à l'analyse finale.
L'idée est ici d'utiliser la puissance du moteur graphique, initialement conçu pour les jeux vidéos, capable de gérer la modélisation et la suppression rapide d'objets 3D.

Problèmes rencontrés

Dans un premier temps il avait été évoqué de pouvoir générer dynamiquement un aperçu de la fleur dans un menu, et de glisser avec la souris celle-ci du panel vers le plateau. Le soucis de cette méthode était que nous n'avions pas pris en compte la difficulté de pouvoir jongler sur plusieurs plans, les panel étant considérés comme des éléments graphiques 2D contrairement au plateau considéré comme un élément 3D. Arriver à jongler entre les deux axes ainsi que de passer d'un objet 2D à un 3D était trop compliqué. Il a donc été décidé de se rabattre sur une solution plus simple : installer un système de modes via un menu. Un autre soucis a été d'arriver à prendre en

compte le fait que l'on ne peut pas avoir une même constante sur plusieurs fleurs, il a donc fallut ajouter un système de vérification à la structure déjà en place.

3.2.4 Options supplémentaires

En plus des fonctionnalités de base qui permettent de manipuler un jardin et d'écrire des formules, il est intéressant que l'utilisateur puisse sauvegarder ces deux éléments pour pouvoir les réutiliser plus tard. Dans un premier temps il a fallut définir un format de sauvegarde. Notre choix s'est porté sur un unique fichier contenant deux parties : une pour le jardin et l'autre pour les formules. Le jardin est sauvegardé comme une liste de quintuplets suivant :

$$coordX, coordY, espece, taille, couleur, nom$$

et les formules sont simplement des chaînes de caractères recopiées telles quelles. Unity ne possédant pas d'utilitaire graphique pré-fait pour parcourir une arborescence de fichier, il a été nécessaire de reconstruire la notre. Elle se compose simplement de deux champs, l'un indiquant le chemin complet et le nom du fichier destination et l'autre permettant de se déplacer dans l'arborescence du répertoire courant. La figure suivante présente l'utilisation de cet utilitaire :

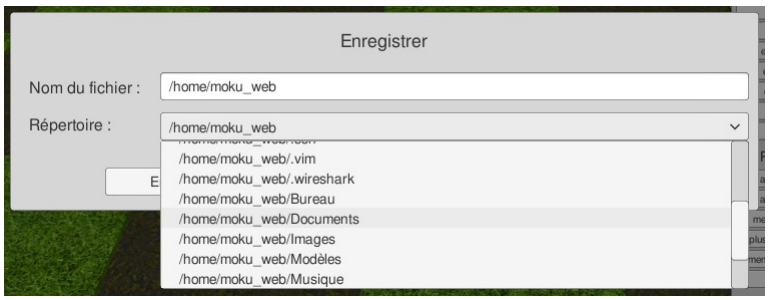

FIGURE 3.4 – Exemple d'utilisation de la fenêtre d'enregistrement

Chapitre 4

Résultats

Pour conclure il est de mise de faire un état des fonctionnalités finales du programme et de les comparer à ce qui était attendu initialement.

4.1 Évolution des fonctionnalités

Les objectifs concernant l'analyse syntaxique et la communication avec le script de base ont bien été atteints conformément aux pré-requis établis pour pouvoir avoir une interface graphique fonctionnelle.
Cependant nous avons du ajuster quelques technologies choisies lors de la réalisation du cahier des charges. Comme évoqué dans le chapitre 3.1.2 sur la Communication avec le script Python, nous avions à l'origine choisi de faire la liaison avec un script Ocaml qui s'est avéré être trop compliquée à mettre en place.
Concernant l'interface graphique, de nombreux points avaient été mis en avant lors de la réalisation du cahier des charges et rappelés dans les chapitres précédents (voir chapitre 2.1 Besoins). Tous les objectifs principaux ont été atteints. Lors de l'avancée du projet, nous avions prévu de faire un système de *Drag & Drop* depuis un panel vers le jardin (voir chapitre 3.2.3 Le Jardin et les fleurs), nous avons du revoir nos exigences à la baisse devant le temps de programmation restant. Nous avons d'ailleurs, et ce tout du long de ce projet, tenu nos engagements par rapport au diagramme de Gantt fixé dans le cahier des charges (voir annexes). Bien que nous ayons espéré finir plus tôt afin de pouvoir ajouter des fonctionnalités supplémentaires.
Dans les causes de petit retards, initialement pris en compte dans la réalisation du planning, on pourra noter la perte de temps avec le premier script Ocaml, les soucis avec GitHub qui bloque les commit de trop grande taille, le passage à un git sur un serveur privé qu'il a fallut monter nous même et donc impliquant des petits soucis de configuration à ajuster et enfin la configuration de Unity à ajuster pour pouvoir y faire passer des librairies externes.
Toutes les fonctionnalités principales nécessaires pour le bon fonctionnement et la bonne utilisation du logiciel sont donc opérationnelles. Mais il est bien entendu toujours possible d'apporter de nouvelles améliorations aussi bien graphiques (améliorer et personnaliser l'aspect graphique de l'interface, création de nouveaux environnements, etc.) que fonctionnelles (ajouter un outil de "retour en arrière", permettre à l'utilisateur de définir des raccourcis pour l'utilisation du clavier, etc.).

Annexes

Vocabulaire

— Panel : Élément graphique de Unity 5 représenté par un panneau configurable sur lequel placer des éléments héritant de la classe UI.
— UI : *User Interface*, classe fournie par Unity comprenant les Panels, boutons, textes, éléments d'interface graphique.
— Drag & Drop : Glisser-déposer en français, manière de gérer une interface en permettant le déplacement de certains éléments vers d'autres conteneurs.
— Tile : Tuile en français, correspond à une case du jardin.

Grammaire pour la vérification syntaxique

```
/*
 * grammar.grammar
 *
 * Grammaire pour la logique du premier ordre
 *
 */

%header%

GRAMMARTYPE = "LL"

DESCRIPTION = "Grammaire pour la logique du premier ordre"


/** Tokens **/
%tokens%

WHITESPACE              = <<[ \t\n\r]+>> %ignore%
LEFT_PAREN              = "("
RIGHT_PAREN             = ")"
COMMA                   = ","

NOT                     = "¬"
AND                     = "∧"
OR                      = "∨"
IMPLY                   = "⇒"
FORALL                  = "∀"
EXISTS                  = "∃"

CSTE                    = <<[a-t]>>
VAR                     = <<[v-z]>>

/* Predicat */
ROSE                    = "Rose"
PAQUERETTE              = "Paquerette"
TULIPE                  = "Tulipe"
EST_GRAND               = "est_grand"
EST_MOYEN               = "est_moyen"
EST_PETIT               = "est_petit"
EST_ROUGE               = "est_rouge"
EST_ROSE                = "est_rose"
EST_BLANC               = "est_blanc"
A_L_EST                 = "a_l_est"
A_L_OUEST               = "a_l_ouest"
AU_SUD                  = "au_sud"
AU_NORD                 = "au_nord"


A_L_EST_DE              = "a_l_est_de"
A_L_OUEST_DE            = "a_l_ouest_de"
AU_SUD_DE               = "au_sud_de"
AU_NORD_DE              = "au_nord_de"
MEME_LATITUDE           = "meme_latitude"
MEME_LONGITUDE          = "meme_longitude"
PLUS_GRAND_QUE          = "plus_grand_que"
PLUS_PETIT_QUE          = "plus_petit_que"
MEME_TAILLE_QUE         = "meme_taille_que"
MEME_COULEUR_QUE        = "meme_couleur_que"
EGAL                    = "egal"

EST_ENTRE               = "est_entre"
```

```
/** Production **/
%productions%

FORMULE = NOT FORMULE
                | FORALL VAR FORMULE
                | EXISTS VAR FORMULE
                | PREDICAT FORMULE_PRED*
                | "(" FORMULE FORMULE_BIN ;

FORMULE_PRED = AND FORMULE
                         | OR FORMULE
                         | IMPLY FORMULE ;

FORMULE_BIN = AND FORMULE ")"
                        | OR FORMULE ")"
                        | IMPLY FORMULE ")"
                        | ")" ;

ATOM = CSTE | VAR ;

PREDICAT = PRED1 | PRED2 | PRED3 ;

PRED1 = ROSE "(" ATOM ")"
                | PAQUERETTE "(" ATOM ")"
                | TULIPE "(" ATOM ")"
                | EST_GRAND "(" ATOM ")"
                | EST_MOYEN "(" ATOM ")"
                | EST_PETIT "(" ATOM ")"
                | EST_ROUGE "(" ATOM ")"
                | EST_ROSE "(" ATOM ")"
                | EST_BLANC "(" ATOM ")"
                | A_L_EST "(" ATOM ")"
                | A_L_OUEST "(" ATOM ")"
                | AU_SUD "(" ATOM ")"
                | AU_NORD "(" ATOM ")" ;

PRED2 = A_L_EST_DE "(" ATOM "," ATOM ")"
                | A_L_OUEST_DE "(" ATOM "," ATOM ")"
                | AU_SUD_DE "(" ATOM "," ATOM ")"
                | AU_NORD_DE "(" ATOM "," ATOM ")"
                | MEME_LONGITUDE "(" ATOM "," ATOM ")"
                | MEME_LATITUDE "(" ATOM "," ATOM ")"
                | PLUS_GRAND_QUE "(" ATOM "," ATOM ")"
                | PLUS_PETIT_QUE "(" ATOM "," ATOM ")"
                | MEME_TAILLE_QUE "(" ATOM "," ATOM ")"
                | MEME_COULEUR_QUE "(" ATOM "," ATOM ")"
                | EGAL "(" ATOM "," ATOM ")" ;

PRED3 = EST_ENTRE "(" ATOM "," ATOM "," ATOM ")" ;
```

Bibliographie

[1] Grammatica documentation. http://grammatica.percederberg.net/. Accessed : 2016-04-26.

[2] IronPython documentation. http://ironpython.netl. Accessed : 2016-04-26.

[3] MSDN csharp documentation. https://msdn.microsoft.com/fr-fr/library/67ef8sbd.aspxl. Accessed : 2016-04-26.

[4] Unity 5 Asset Store 3d material et références. https://www.assetstore.unity3d.coml. Accessed : 2016-04-26.

[5] Unity 5 Documentation manuel. http://docs.unity3d.com/Manual/index.html. Accessed : 2016-04-26.

[6] Unity 5 Documentation script manuel. http://docs.unity3d.com/ScriptReference/index.html. Accessed : 2016-04-26.

[7] Unity 5 Forum forum et communauté unity. http://unity3d.com/community. Accessed : 2016-04-26.