IA01

TP03 - Conduite d'expertise d'un SE d'ordre 0+ :

La Roulette du potionniste

Alexis Deseure et Colin Manyri

Table des matières

I. Introduction

Organisation du rapport

Ce rapport a pour objectif de présenter notre choix de Système Expert, l'expertise qui y est associée ainsi que le code LISP associé au TP03. Vous trouverez le détail du code source ainsi qu'une présentation pour chaque fonction. Vous trouverez également un jeu d'exemple pour la fonction principale ainsi que les résultats qui y sont associés. Ce document s'inscrit dans l'archive TP03_Potions_Harry_Potter_AlexisDeseure_ColinManyri où se trouve le fichier source de l'exercice contenant les fonctions ainsi que quelques exemples.

L'archive TP03_Potions_Harry_Potter_AlexisDeseure_ColinManyri.zip contient les documents suivants :

- IA01_TP03_rapport_SE.docx : Ce rapport
- scrapping_potion.py: Programme de "scraping web" en python que nous présenterons en détail
- bdd_potions.csv: Le CSV final, après avoir revu la sortie du programme de scraping et base de données de notre expertise (ne possède pas toutes les informations de la base de règles)
- bdd_ingredients.csv : Le CSV contenant la correspondance des potions qui utilise chacun des ingredients
- TP03_La_Roulette_du_potionniste.lisp : Le système expert à proprement parler, codée en LISP, commenté et avec les jeux de testes associés. Pour lancer, simplement faire un CTRL+A et un CTRL+E ou lancer les jeux de fonction tout en bas. (c'est la fonction fonction-principale pour lancer)
- Soutenance TP.pptx : Le transparent de la soutenance qui a eu lieu le 21/12/2023

Présentation du TP03

Mise en contexte

Pendant les vacances de Noël, comme à son habitude Harry reste à Poudlard tandis que tous ses camarades sont de retour chez eux. Harry qui s'ennuie rentre par hasard dans le bureau de Rogue, le professeur de potions, puis ... BAM !

La porte se ferme violemment et il est impossible de l'ouvrir ! Ennuyé et bloquer dans le bureau, Harry explore la salle et tombe sur la réserve d'ingrédients secrets de Rogue. Ayant de grosses lacunes dans la confection de potions, Harry mélange des ingrédients n'importe comment et boit ses créations comme s'il s'agissais de jus de citrouille.

Que va-il arriver à Harry ?

Va-t-il se transformer ? Subir les effets d'un poison mortel ou s'en sortir ?

Objectifs et définitions

Ce TP a pour objectif de créer un Système expert d'ordre 0+ sur un sujet original et ludique : les potions dans l'univers d'Harry Potter. Les paramètres d'entré du systèmes expert sont les différents ingrédients de la réserve de Rogue et quelques autres paramètres que nous détaillerons par la suite. Le système expert renvoie alors les potions crées par Harry, les effets liés à ses potions que Harry a subit et l'état de santé d'Harry à la fin de l'expérience.

Notre système expert pourra être présenté comme un générateur d'histoire ludique, dans lequel Harry Potter subit les effets de ses potions !

Choix du sujet

Dans un premier temps, nous avions décidée de faire notre système expert sur les champignons, seulement nous trouvions ce sujet peu original. Nous avons donc rapidement abandonné l'idée, et nous sommes rabattu sur le sujet de la saga Harry Potter.

La saga Harry Potter est une série de sept livres écrits par l'auteure britannique J.K. Rowling et qui fut adapté en de nombreux films, séries et même jeux vidéo. Dans l'univers de la saga, il existe un monde magique dans lequel les sorciers vivent caché. C'est dans ce contexte qu'évolue Harry Potter, jeune orphelin au début de la saga, il apprend que c'est un sorcier et qu'il est admis à l'école de magie de Poudlard. Avec ses amis Ron et Hermione, il découvrira les mystères de l'école et affrontera Voldemort, un sorcier maléfique, principal antagoniste de l'histoire.

Qui dit école magique, dit cours de confection de potions. Nous avons à partir de là décidés de nous concentrer sur ce sujet pour effectuer un système expert d'ordre 0+. Nous avons dans un premier temps voulus faire une système expert capable de reconnaitre des potions à partir d'une description d'ingrédients, de couleurs et d'effets, mais nous somme rapidement revenu sur cette idée car nous la trouvions trop simple et peu originale. L'idée de faire boire à Harry Potter chaque potion qu'il réussira à crée nous est ensuite venu. Nous avons alors eu l'idée d'un système expert reconnaissant les potions qu'Harry peut créer et leurs effets, via une liste des ingrédients donnée, et qui fait subir à Harry différents effet. Le choix de ce sujet nous convenait car il comportait des défis techniques, tout gardant un aspect original et ludique.

Nous avons alors trouvé le titre de notre système expert : La roulette du potionniste. En effet, l'état de santé d'Harry étant incertain, cela nous a fortement rappelé le jeu de la roulette russe.

II. Problématique et expertise

Problématique

L'utilisateur va devoir choisir un but au début du programme et le Système Expert va répondre à la problématique : *Harry Potter va-t-il survivre à l'expérience et si oui, dans quels états ?* Comme expliqué précédemment, l'utilisateur va devoir choisir tous les ingrédients que Harry a à sa disposition dans le bureau de Rogue pour tester les combinaisons qui forme des Potions (affiché dans un vieux livre mais sans indications sur les effets). Chacun des ingrédients permettrons de former des potions, elles-mêmes pouvant former d'autres potions combinées avec d'autres éléments tout en ayant un effet propre et des conditions de réalisation particulières. Au temps 0, l'utilisateur choisi en plus des éléments d'état sur Harry : sante, concentration et niveau (dans la discipline de conception de potions). A la fin Harry aura subit pleins d'effets qui auront impacté les états définit initialement. De plus, au cours du processus, certaines potions demandent des conditions particulières pour être faites : santé, concentration minimale ou niveau minimum. Certaine potion très difficilement réalisable (relativement grande profondeur demandant un total important d'ingrédient) peuvent être décisif dans l'issu de la partie et donc dans la réalisation du but fixé par l'utilisateur. Il y en a 2 types : la survie avec un minimum de potions créés (car il y a un compteur s'incrémentant à chaque potion réalisée) ou bien simplement la mort d'Harry.

Expertise

Bien que le thème d'Harry Potter soit purement fictif, la saga est suffisamment populaire pour que de nombreuses informations soient disponibles, il est donc possible de mettre en place un véritable travail d'expertise sur ce sujet. Nous avons utilisé comme principale source le Wiki Harry Potter fandom , le site qui répertorie le plus d'information sur la saga en particulier, les potions et qui est mis à jour par des fans. Sur ce site, plus de 120 potions sont répertoriés, la plupart ont un nom unique, une liste d'ingrédient parmi les plus de 180 possibles et des effets uniques.

Face à la grande quantité d'informations et comme nous ne voulons pas nous limiter aux potions les plus connues, nous avons décidé d'automatiser la récupération de ces données et donc de développer un programme de scraping web. Nous avons choisi le langage Python pour ce code car nous n'avions pas les connaissances des fonctions pour faire ceci en LISP. Nous trouvons néanmoins intéressant de développer ce point de notre travail.

Programme de scraping en Python

Le programme de scraping en Python commenté est consultable dans l'archive, sous le nom de scrapping_potion.py. Pour résumer, il utilise les bibliothèques beautifulSoup(pour parser le contenu des pages web et en extraire les informations utiles), pandas(pour le traitement du dataframe, conversion en csv), requests (pour faire une requête get sur l'url des pages web) et re (pour certaines opérations basées sur des expressions régulières) afin de récupérer tous les liens des potions sur la page principale du wiki potions puis de parcourir chacune de leur page pour récupérer leurs contenus. La difficulté a résidé dans la sélection des éléments pertinents car étant écris par des utilisateurs différents, les balises html ne sont pas toujours les mêmes et il est trop long de parcourir les 120 pages manuellement pour avoir toutes les syntaxes. Mais on a toutefois pu retrouver des paterns similaires qui nous ont permis d'avoir un résultat relativement précis et pertinent.

Suite à cela, nous avons récupéré toutes les informations sur chaque potion sur un fichier CSV. Le CSV réunit alors pour chaque potions les informations suivantes :

- Nom de la potion
- Le lien vers la page du wiki dédiée à la potion
- Description globale de la potion, donnée par le Wiki
- Effets de la potion
- Description visuelle de la potion
- L'inventeur de la potion
- Les ingrédients nécessaires à la création de la potion
- Première apparition dans le monde imaginaire
- Catégories de l'élément (Potion, médication, poison, etc)
- La difficulté de préparation de la potion
- Un lien vers une image de la potion, qui est parfois donnée dans le Wiki

A partir du premier CSV, on a remarqué qu'il manquait des données ce qui nous amène à la prochaine section.

Nettoyage de la BDD

Par la suite nous nous sommes rendu compte que la base de données était très hétérogène, ceci est due au fait que le Wiki est rempli pas des fans et donc ne dispose pas d'une rédaction standardisée des informations sur les potions. Nous avons donc harmonisé la base de données (en remplacent les ingrédients similaires par le même nom) pour obtenir une liste d'environ 180 ingrédients (184 exactement), tous utilisés dans les différentes potions.

De plus, certaines informations ne sont pas importantes ou sont incomplètes, par exemple, toutes les potions n'ont pas d'ingrédients car elles n'apparaissent qu'une seule fois brièvement

dans l'univers d'Harry Potter, de même certains niveaux ne sont pas renseignés. A l'inverse certaines informations n'étaient pas vraiment pertinentes pour notre étude comme par exemple la première apparition de la potion ou son inventeur. Par conséquent, en utilisant notre imagination et la description des potions nous avons repris des ingrédients et des potions mais combinés différemment entre eux pour former les prémisses des règles de notre base. Certains champs ont été conservés afin d'avoir des pistes d'amélioration pour notre programme pour par exemple afficher les informations et l'image des potions faites à l'issu du programme. Nous avons attribué aussi à toutes les potions un aspect physique et visuelle à partir de leur image/description même si cela ne nous a pas servis dans notre programme faute de pertinence et de diversité (presqu'uniquement des liquide simple et couleurs assez aléatoires car peu étaient renseignées sur le wiki).

Une potion ne nécessite pas uniquement des ingrédients pour être crée, mais aussi le talent d'un magicien. En effet, les plus grands maitres des potions ont des dizaines d'années d'expérience dans le domaine. Et la création de potion à Poudlard est considéré comme l'une des matières les plus difficile. Nous avons donc décidé d'ajouter ces aspects aux différentes potions, via deux critères :

- La concentration : qui mesure le niveau de concentration d'Harry Potter au moment de créer la potion, et donc ces chances de faire des erreurs ou non. Cette valeur peut être modifié si Harry subit les effets néfastes d'une potion (la valeur qu'on a défini sur chaque potion est assez arbitraire et dépend en partie de la difficulté de la potion, de sa description, et de la puissance de son effet).
- Le niveau : qui mesure le niveau l'expérience d'Harry dans la création de potions, et qui augmente naturellement quand Harry réussi à créer une potion (nous l'avons défini grâce au champ difficulté issue du scraping initial et de notre connaissance/description).

L'objectif du système expert est de faire subir à Harry les effets de différentes potions. Seul problème, les effets possibles pour chaque potion sont très variés : cela va d'un poison mortel, jusqu'un soin le plus puissant, en passant par des effets de métamorphoses, de résistance, voir même de pousse de cheveux ! Nous avons donc dû encore harmoniser notre base de données pour prendre en compte tous les cas possibles. Nos potions ont alors été classé dans différentes catégories d'effets en fonction de leurs effets propres. Nous avons défini les catégories d'effet de la manière suivante :

- Les Poisons, ont un effet d'intoxication, néfaste pour la personne qui le boit.
- Les Contrôles mentales sont des potions capables d'altérer le jugement de la personne
- Les Hallucinogènes qui provoquent des hallucinations, des vertiges ou des somnolences
- Les Affaiblissant qui affaiblissent la personne qui la boit.
- Les Fortifiant, qui ont l'effet inverse des Affaiblissants
- Les Soins, qui soignent le buveur

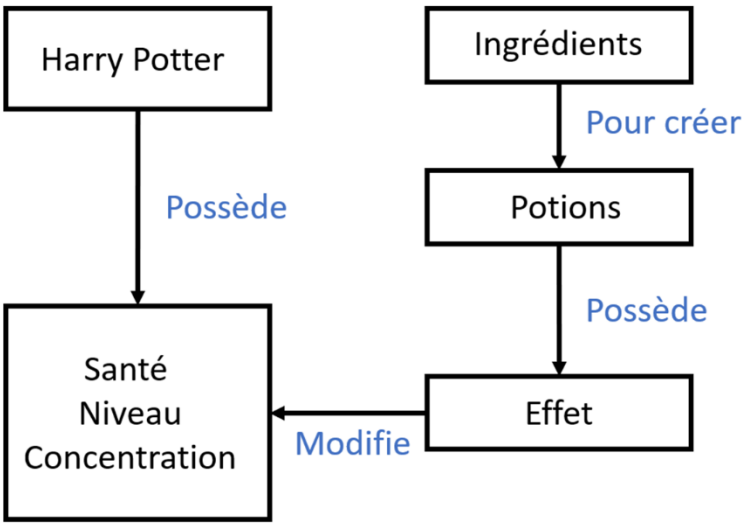
- Les Antidotes, qui peuvent annuler les effets néfastes des potions
- Les outils, qui sont des potions étant destiné à différents autres taches, parmi lesquels on retrouve des produit ménager, explosif, de chance ou de métamorphoses.
- Les potions liées au jardinage (engrais, désherbants,etc), elles ont peu d'effets sur Harry

Dans le CSV disponible dans l'archive à propos des potions on retrouve certains groupes de catégories (outil, métamorphose, pouvoir externe, chance) qu'on a regroupé dans outils ci-dessus car ils n'ont pas de règles propres dans la base de données (pas de compteur) mais un effet défini au cas par cas.

Chaque potion à donc été répertorié dans une de ces catégories en fonction de ses effets. Une potion n'est pas limitée à une seule catégorie, mais peut en cumuler plusieurs. Par exemple : un Poison peut également être Hallucinogène, et Affaiblissant.

Ce système permet par la suite à certaines règles de la base de règles d'appliquer des effets uniques sur Harry en fonction de la potion crée et de sa catégorie.

Voici un résumé schématique de toutes les informations à formaliser dans notre base de règles:



Choix et forme de la base de règles

Forme des prémisses :

Le système expert est d'ordre 0+, il peut donc prendre en compte des prémisses faisant des liens entre des variables et des valeurs fixées. Nous avons donc dû trouver un moyen efficace de représenter tous les types de prémisses possibles.

Nous avons choisi cette représentation :

Prémisse = (Variable Valeur operateur)

La variable est modifiable, la valeur sera toujours un nombre fixe tandis que l'operateur logique lie la variable et la valeur. Nous pouvons choisir comme operateur logique >, >=, <= <, = ou !=.

Cette méthode de représentation permet de représenter aussi bien des prémisses booléennes que des prémisses d'ordre 0+.

Exemple :

Pour un booléen, il suffit de considérer une variable de prémisse booléenne : si variable = 1 alors la prémisse est vraie, sinon elle est fausse. On écrit donc (Variable 1 =) et (Variable 1 !=) ou (Variable 0 =) pour la négation.

Pour la prémisse d'ordre 0+, voici un exemple : (Variable 10 >) ne sera vrai que si « Variable > 10 »

Forme des conclusions :

Nous avons utilisé le même principe pour représenter nos conclusion, seulement cette fois ci, l'opérateur est utilisé à des fins de modification de la variable en fonction de la valeur donnée, operateur peut donc valoir : +, -, /, *, =…

Exemple :

La conclusion (Variable 1 =) effectuera Variable = 1 si la règle est vérifiée.

La conclusion (Variable 10 +) effectuera Variable = Variable + 10 si la règle est vérifiée.

Forme des règles :

Finalement les règles prennent la forme suivante :

(liste_conclusion liste_premisse nom_de_la_règle)

Exemple :

En définissant les variables :

V0 = 1, V1 = 0 , V2 = 0

(((V1 1 +) (V2 10 +)) ((V0 1 =)) « R1 »)

(((V2 10 +)) ((V1 1 =)) « R2 »)

R1 est vérifié car V0 =1. Cela implique que V1 = 1 et V2 = V2 + 10 donc V2 = 10

Puis V1 =1, donc R2 est vérifié. Et finalement : V0 = 1, V1 = 1 et V2 = 20.

Les noms de nos règles sont des strings contenant une action (Harry a fabriqué telle potion + optionnellement une petite phrase humoristique/descriptive de ce qu'il s'est passé ensuite) pour pouvoir directement les afficher à la suite pour former une histoire à la fin du programme.

Nous avons aussi fait le choix que les ingrédients initiés au début ne sont pas en nombre limité car sinon ce serait encore plus difficile pour l'utilisateur d'obtenir un résultat avec un nombre important de potions créés.

III. Connaissances nécessaires au SE

Base de règles

Après avoir définis ces *Choix et forme de la base de règles*, nous pouvons rédiger notre base de règles. Elle comporte des règles pour chaque potion répertoriée de la forme :

(((nb_potion 1 +) (nb_categorie_de_la_potion 1 +)) ((Ingredients 1 =) (Potions 1 =)) « texte d'ambiance »)

Vous pouvez d'ores et déjà remarquer la présence d'un premier compteur nb_potion incrémenté à chaque règle conduisant à la création d'une potion. En effet, cela permet au maximum d'exploiter l'intérêt du 0+ et nous permet de vérifier aisément nos buts davec un minimum de potions confectionnées.

Également des règles liées aux différents effets des potions, par exemple, pour les poisons :

(((nb_Poison 1 -) (sante 2 -)) ((nb_Poison 0 >) (sante 0 >)) "Harry ressent les effets d'un précédent poison et perd de la vie")

Dans cette règle, si le nombre de poisons est positif, Harry a donc créé au moins un poison, et un poison peut être consommé et faire effet sur Harry. Son niveau de vie est donc mis à jour (parfois concentration pour les autres types de potions) et le compteur de potions de ce type est décrémenté. Bien sûr, la plupart des règles ont la prémisse (sante > 0) pour arrêter le moteur d'inférence quand aucune règle n'est vérifiable. Pour éviter d'avoir une concentration négative qui serait trop dur à remonter pour Harry, elle ne peut pas descendre en dessous de 0 donc les règles sur les groupes de potions qui en font perdre vérifient en premier lieu si Harry a une concentration supérieure au nombre qu'ils font perdre et si c'est le cas la concentration est mise directement à 0 :

(((nb_Affaiblissant 1 -) (sante 2 -) (concentration 10 -)) ((nb_Affaiblissant 0 >) (sante 0 >) (concentration 9 >)) "Harry avait bu une potion affaiblissante, il ressent de légères douleurs et est engourdie, sa santé et sa concentration diminue")

(((nb_Affaiblissant 1 -) (sante 2 -) (concentration 0 =)) ((nb_Affaiblissant 0 >) (sante 0 >) (concentration 9 <=)) "Suite à un effet affaiblissant Harry n'est plus du tout concentré et sa santé diminue")

Ensuite, il y a 4 règles de but :

(((but excellent =)) ((nb_potion 40 >) (sante 0 >) (but excellent !=)) "A deux doigts de décéder... Bravo tout de même Harry !")

(((but super =)) ((nb_potion 20 >) (sante 0 >) (but excellent !=) (but super !=)) "Harry est officiellement un maître des potions")

(((but moyen =)) ((nb_potion 10 >) (sante 0 >) (but excellent !=) (but super !=) (but moyen !=)) "Harry a eu de la chance de tenir jusqu'ici !")

(((but mort =)) ((sante 0 <=) (but mort !=)) "Harry est enfin mort !")

Les 3 premières consistent en un nombre minimum de potions à réaliser et la dernière à la mort d'Harry (comme expliqué précédemment), elles ont donc en prémisses une santé positive, le nombre de potions à dépasser et la négation des autres buts supérieurs ou égaux pour ne pas répéter inutilement 2 fois la même règle (qui pourrait conduire à un stackoverflow car la condition d'arrêt ne serait plus respectée si le but est morbide et que Harry n'est pas mort et ne peut plus faire de potions par exemple). On ne veut pas non plus que des buts nécessitant moins de potions « écrasent » à chaque fois les buts supérieures respectés (sauf le but morbide qui est décisif et met nécessairement fin au programme car plus aucune règle n'est vérifiable ensuite avec une sante < 0).

Les règles sur les potions fonctionnent de manière similaire mis à part le fait qu'elles ont des prémisses sur le test de booléens correspondant à la présence ou non d'ingrédients ou de potions et qu'elles vérifient si elles n'ont pas déjà été créées pour ne pas tourner en boucle.

Enfin la prémisse (sante 0 >) est toujours présente, et permet de vérifier que Harry est toujours en vie.

Notre base de règle est définie dans notre programme Lisp dans la variable globale : *bdr*.

Base de faits

La base de fait est essentielle dans la base de connaissances et le SE en général car c'est elle qui contient l'état initial des données et qui va évoluer grâce au moteur d'inférence et au chaînage pour obtenir les informations qu'on souhaite tirer de notre programme.

Pour représenter notre base de faits et pouvoir la passer en paramètres de fonctions, nous avons choisi la représentation sous forme de listes de listes clés – valeurs :

((clé1 valeur1) (clé2 valeur2) (clé3 valeur3) …)

Où la clé représente la variable sous forme de symbole Lisp et la valeur représente son contenu (entier, symbole, …).

On peut considérer plusieurs types de variables que vous avez pu deviner avec les précédentes explications :

- Les variables de types potions qui sont en fait le nom des potions où les caractères problématiques en lisp ont été remplacer par des « _ » pour éviter les problèmes de

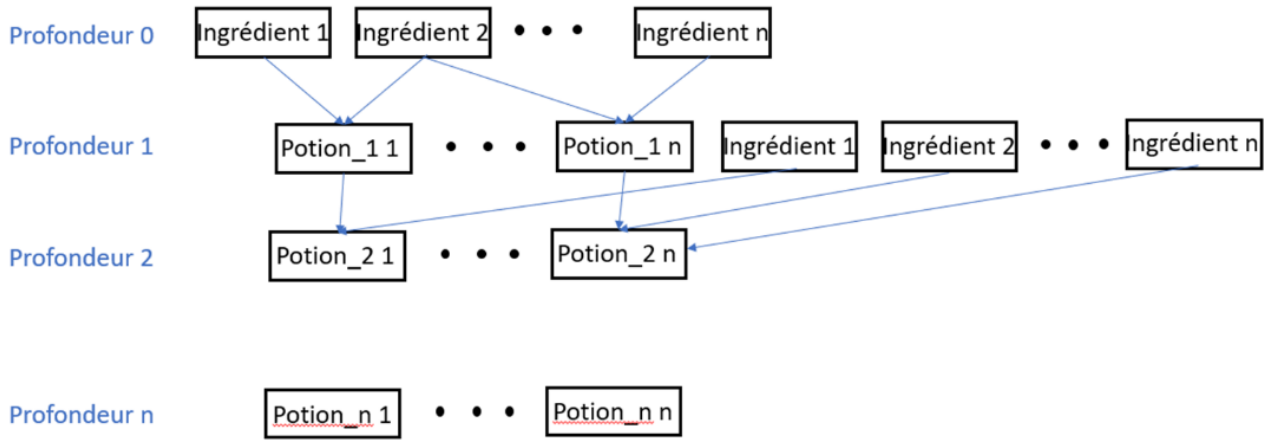
syntaxe sur les symboles. Elles contiennent une valeur booléenne (présence ou non dans l'inventaire d'Harry) qui peut évoluer au cours du programme (fixés à 0).

- Les variables ingrédients qui fonctionnent exactement selon le même principe à ceci près qu'elles sont uniquement définit au début par l'utilisateur puis fixe pendant le déroulement du programme (quantité illimitée de chaque ingrédient s'ils sont fixé à 1)
- Les compteurs de types nb_{nom} qui s'incrémentent et se décrémente au fil du programme et qui sont fixés à 0 initialement.
- Les paramètres de Harry (sante, niveau et concentration), supérieurs à 0 initialement mais qui évoluent (sauf le niveau) avec le programme (niveau peut être définit supérieur à 3 mais ça reviendra exactement au même que s'il valait 3, sante peut devenir négatif auquel cas Harry est mort)
- Le but prenant initialement la valeur de MAUVAIS et qui sera potentiellement modifié pendant l'exécution du programme. Il fait aussi office de condition d'arrêt de notre chaînage.

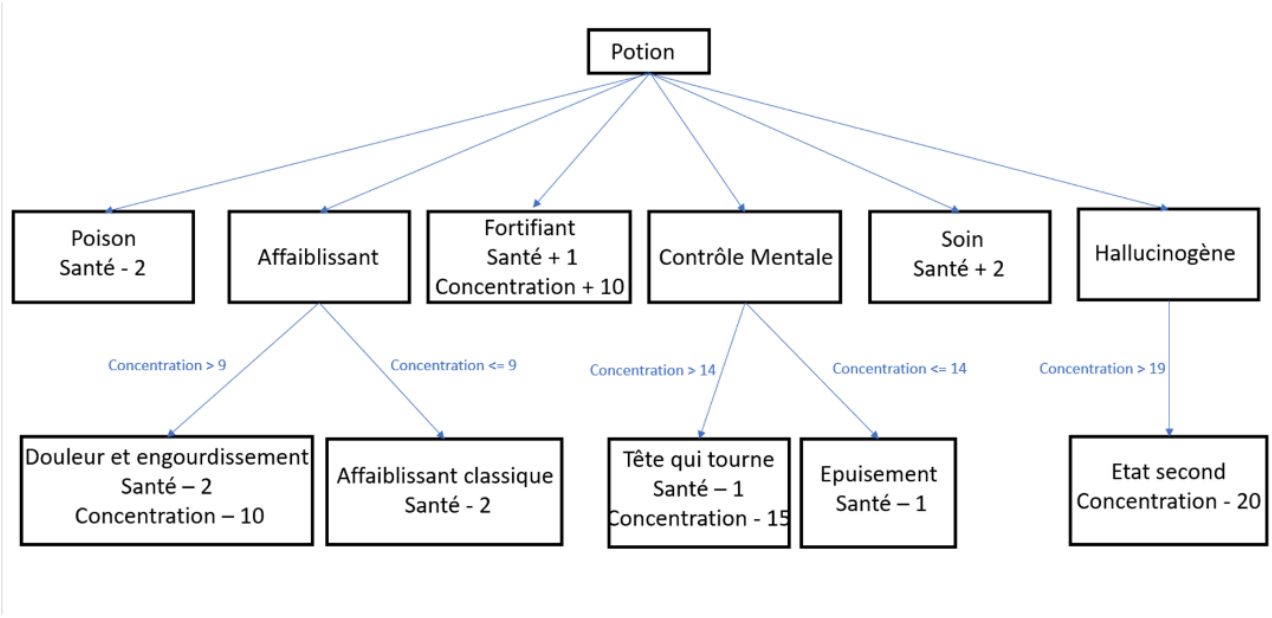
La liste des ingrédients (respectivement potions) et des informations associées est stockée dans la variable globale *bdd_menu* (respectivement *bdd_potions*).

Arbre de déduction :

Voici une manière simplifiée de représenter l'arbre de déduction dans notre cas. Nous avons décidé de séparer l'arbre en deux parties. Tout d'abord, la détermination de la/les potions faisables, puis le choix et l'application de l'effet. Bien entendu, ces deux arbres sont assemblables pour en former un seul, mais par souci de lecture nous avons choisi de le représenter en deux étapes.



Ce deuxième arbre gère la déduction du type d'effet à appliquer et se situe à la suite du premier, pour chaque potion.



Jeux d'essais et exemples

La fonction principale de notre programme qui gère l'interface utilisateur, l'initialisation de la base de faits et le lancement du chaînage prend en paramètre la variable globale des potions, de la base de règles et des ingrédients (*bdd_potions*, *bdr* et *bdd_menu*). Elle prend aussi un paramètre optionnel qui correspond à la liste d'entier des ingrédients que l'on souhaite insérer pour passer cette étape fastidieuse qu'est l'entré des ingrédients. Nous vous proposons, ci-dessous, 3 jeux de test qui peuvent être intéressant à essayer et à faire varier les paramètres de sante, concentration et niveau mais dont les ingrédients sont fixés grâce au paramètre optionnel :

- La liste de tous les ingrédients : (fonction-principale *bdr* *bdd_menu* *bdd_potions* '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158))
- (fonction-principale *bdr* *bdd_menu* *bdd_potions* '(27 92 14 105 43 76 8 150 62 38 121 4 83 52 113 99 18 64 37 55 126 12 95 70 81 33 147 7 136 59 124 28 145 9 100 2 79 49 132 16 69 131 48 98 5 25 141 72 35 153 85 22 139 117 57 142 51 106 91 44 75 20 134 61 153 10 138 30 89 68 146 115 40 60 77 127 24 87 111 74 3 119))
- (fonction-principale *bdr* *bdd_menu* *bdd_potions* '(107 132 28 109 92 121 130 133))

IV. Implémentation

Type de moteur

Pour notre système expert nous avons choisi un moteur en chainage avant et en profondeur d'abord.

Chainage avant

Notre problème est bien plus adapté à un chainage avant qu'a un chainage arrière. En effet, nous voulons dans un premier temps déterminer toutes les potions réalisables par Harry Potter avec les ingrédients donnés dans la base de fait. De plus, dans notre cas, le but étant un nombre de potions à réaliser, il est difficile de remonter la base de règle en partant de ce but, car toutes les potions crée nous rapproche de ce but de manière égale. Il n'y a donc pas d'intérêt de faire un chainage arrière dans cette situation.

En profondeur

Nous avons fait ce choix, afin d'appliquer les effets des différentes potions directement après que celles-ci étaient créées, cela est alors en adéquation avec le fait qu'Harry boit directement les potions après leurs créations. D'un point de vue technique les règles d'application des effets sont représentées de manières plus profondes dans l'arbre de décision que la création des potions elles-mêmes. En progressant en profondeur, on est sûr d'appliquer les effets d'une potion crée avant de s'intéresser aux autres potions réalisables. Le choix d'un parcours en largeur aurait pu être envisageable mais aurai eu pour effet de créer d'abord toutes les options possibles avant d'appliquer les effets de toutes les potions une à une, ce qui est moins conforme à l'histoire racontée.

Algorithme et explications

Nous allons maintenant pouvoir expliquer comment fonctionne notre moteur d'inférence.

Fonctions de services utiles au moteur d'inférence

En premier lieu, pour améliorer la lisibilité du code et éviter la redondance, nous avons créés différente fonctions de services.

Les trois premières sont des fonctions basiques pour sélectionner des parties spécifiques du code :

- nom-regle sélectionne le dernier élément de la liste correspondant à une règle c'est-à-dire son nom ou plutôt la phrase de son action (caddr regle)
- lister-premisses sélectionne les prémisses d'une règle (cadr regle)
- lister-conclusions sélectionne les conclusion d'une règle (car regle)

Ensuite nous avons trois autre fonctions de services qui vont vérifier si prémisses et règles sont correctes e appliquer les conclusions en conséquences.
La fonction verifier-premisse a pour but de vérifier si une seule prémisse est vraie. Pour cela, elle prend en paramètre la prémisse en question (format (variable valeur opérateur)) et la base de fait dans le format vu dans *Base de faits.*
Elle définit simplement les trois variables locales associées aux trois éléments contenus dans une prémisse (variable, valeur et opérateur) puis vérifie à l'aide d'un bloc conditionnel si l'opérateur correspond à un de ceux définit et si pour celui-ci la condition est respectée. Si c'est le cas elle renvoie T sinon nil. Pour obtenir la valeur de la variable elle utilise assoc sur la base de faits)

```
(defun verifier-premisse (premisse bdf)
   (let ((variable (cadr (assoc (car premisse) bdf))) (valeur (cadr premisse)) (operateur (caddr premisse)))
      (cond
         ((and (string-equal operateur '>) (> variable valeur)) t)
         ((and (string-equal operateur '<) (< variable valeur)) t)
         ((and (string-equal operateur '>=) (>= variable valeur)) t)
         ((and (string-equal operateur '<=) (<= variable valeur)) t)
         ((and (string-equal operateur '=) (eq variable valeur)) t)
         ((and (string-equal operateur '!=) (not (eq variable valeur))) t)
      )
   )
)
```

Ensuite, la fonction verifier-regle permet de vérifier si toute les prémisses d'une règle sont vérifiées. Elle prend en paramètre la règle associée et la base de fait. Elle commence par créer la variable locale contenant la liste des prémisses de la règle puis, elle fonctionne de manière récursive pour vérifier à tour de rôle si la prémisse en première position de la liste est vraie grâce à verifier-premisse. Si c'est le cas, elle se rappelle récursivement avec la règle sans la première prémisse de la liste initiale sinon elle renvoie nil. La condition d'arrêt est lorsque la liste des prémisses vaut nil auquel cas on renvoie T.

```
(defun verifier-regle(regle bdf)
   (let ((liste_premisses (lister-premisses regle)))
      (if (null liste_premisses)
         T
         (if (verifier-premisse (car liste_premisses) bdf)
            (progn
               (setq regle (cons (car regle) (append (list (cdr liste_premisses)) (cddr regle))))
               (verifier-regle regle bdf)
            )
         )
      )
   )
)
```

Enfin, la fonction appliquer-conclusion applique les conclusions d'une règle si elle est vérifiée et renvoie la base de faits ainsi modifiée. Elle prend en paramètre une règle et la base de faits. De manière itérative avec un dolist, elle parcourt la liste des conclusions d'une règle et stocke pour chacune d'elles la valeur de la variable, de la valeur et de l'opérateur ainsi lues. Puis à l'aide d'un bloc cond elle vérifie qu'elle opérateur est écrit et applique les modification sur la base de faits en fonction. Dans le cas d'une erreur de syntaxe elle renvoie une erreur.

```
(defun appliquer-conclusion(regle bdf)
    (progn
        (dolist (conclusion (lister-conclusions regle) bdf)
            (let ((variable (car conclusion)) (valeur (cadr conclusion)) (operateur (caddr conclusion)))
                (cond
                    ((string-equal operateur '+) (setf (cadr (assoc variable bdf)) (+ (cadr (assoc variable bdf)) valeur)))
                    ((string-equal operateur '-) (setf (cadr (assoc variable bdf)) (- (cadr (assoc variable bdf)) valeur)))
                    ((string-equal operateur '*) (setf (cadr (assoc variable bdf)) (* (cadr (assoc variable bdf)) valeur)))
                    ((string-equal operateur '/) (when (/= valeur 0) (setf (cadr (assoc variable bdf)) (/ (cadr (assoc
    variable bdf)) valeur))))
                    ((string-equal operateur '=) (setf  (cadr (assoc variable bdf)) valeur))
                    (t (format t "Erreur : Opérateur non pris en charge ou Division par 0 !~%"))
            )
        )
    )
  )
)
```

Fonction principale du moteur d'inférence :

La fonction "moteur-chainage-avant" est le cœur de notre système expert. Ce moteur en profondeur et en chainage avant utilise une méthode récursive pour déterminer toutes les règles vérifiées en fonction de la base de fait donné et de la base de règle.

Ce programme récursif a deux conditions d'arrêts :

- 1er cas : le but fixé par l'utilisateur est atteint, c'est-à-dire, la règle associée à ce but est vérifiée
- 2e cas : un appel récursif ne permet de vérifier aucune règle. Cela implique que la base de faits n'a pas pu être modifiée durant un appel récursif et donc que toutes les règles vérifiables le sont déjà. En effet, sans modification de la base de faits, il est impossible de vérifier de nouvelles règles dans la manière dont notre base de règle est construite.

Voici tout d'abord un algorithme en langage naturel de notre fonction de chainage :

```
Moteur_en_chainage_avant (but bdf bdr &optional (regles_verifie)) :
    Pour toutes les règles de la base de règle :
        Si le but n'a pas déjà été vérifié
            Si la règle est vraie
                Ajouter la règle à la liste regles_verifie
                Modifier la bdf en conséquence
                Appel récursif avec la nouvelle bdf et regles_verifie (parcours en profondeur)
            Fin si
        Sinon
            Fin du programme, renvoyer (bdf regles_verifie) et but atteint
        Fin si
    Fin pour
    Renvoyer (bdf regles_verifie)
Fin du programme
```

Voici maintenant le code lisp associé à cet algorithme :

```
(defun moteur_chainage_avant (but bdf bdr &optional (regles_verifie))
  (let (resultat_recursivite)
    (dolist (regle bdr)
      (if (not (eq (cadr (assoc 'but bdf)) but))
        (if (verifier-regle regle bdf)
          (progn
            (setq regles_verifie (append regles_verifie (list (nom-regle regle))))
            (setq bdf (appliquer-conclusion regle bdf))
            (setq resultat_recursivite (moteur_chainage_avant but bdf bdr regles_verifie))
            (setq bdf (car resultat_recursivite))
            (setq regles_verifie (cadr resultat_recursivite))
          )
        )
        (return-from moteur_chainage_avant (list bdf regles_verifie))
      )
    )
    (list bdf regles_verifie)
  )
)
```

Fonction principale et interface utilisateur

Notre fonction principale comme expliqué précédemment permet l'initialisation de la base de fait, l'interaction avec l'utilisateur, l'affichage et le lancement du moteur d'inférence du SE. Elle s'appuie sur plusieurs foncions de services que l'ont va expliquer brièvement ci-dessous. (les codes sont directement disponibles dans le fichier TP03_La_Roulette_du_potionniste.lisp de l'archive)

- La fonction recup-line-ingredient permet de récupérer toutes les infos sur un ingredient dans la bdd_ing à partir de son index à l'aide d'un simple assoc.

- La fonction recup-nom-ingredient permet à l'aide d'une ligne récupéré au préalable sur une bdd_ing et passée en paramètre sous le nom de ing, de renvoyer le nom en toutes lettres de l'ingredient.
- La fonction recup-var-ingredient permet de la même manière de récupérer cette fois le nom de l'ingredient mais sous le format symbole (espaces et apostrophes ont été remplacées par des « _ ») pouvant être ainsi utilisable plus facilement sur Lisp.

- La fonction recup_line_avec_var_ingredient a exactement le même objectif que recup-line-ingredient mais cette fois ci en utilisant le symbole associé à l'ingrédient plutôt que l'index. Pour cela on place le symbol en première position de chaque sous liste de bdd_ing pour ensuite appliquer un assoc sur le résultat et ainsi obtenir la bonne ligne.

- La fonction afficher-menu-ingredients va simplement afficher ligne par ligne les éléments de la bdd_ing pour permettre à l'utilisateur de voir à quels ingrédients correspondent chaque entier.

- La fonction recap_ingredient prend en paramètres la liste des couples (symbol_ingredient valeur) avec valeur un booléen ainsi que bdd_ing. Elle recherche le nom des ingredients qui ont pour valeur 1 et recherche leur nom dans bdd_ing grâce à recup_line_avec_var_ingredient puis les affiche en console.

- La fonction choisir-ingredient initialise la variable « ingredients » à 0 dans la liste des couples (symbol_ingredient valeur) associée en variable locale. Elle prend en paramètres la potentielle liste d'ingrédients passée en paramètre optionnel par l'utilisateur et n'affiche pas le menu si c'est le cas. Dans ce cas, elle va simplement mettre à 1 la valeur des ingrédients correspondant à chacun des id de la liste. Si l'argument optionnel n'est pas renseigné ou incorrect, alors la fonction va faire appel à afficher-menu-ingredients et vérifier les entrées de l'utilisateur à tour de rôle puis mettre à 1 la valeur dans ingredients correspondant à l'index renseigné puis attendre à nouveau une nouvelle entrée de l'utilisateur jusqu'à qu'il entre 0 en console pour

continuer. A la fin la fonction renvoie la liste ingredients qui constitue en réalité une partie de la base de fait initialisée.

- La fonction choix_paramètre prend en paramètre le couple (symbol_paramètre valeur) puis demande à l'utilisateur d'entrer une valeur strictement positive pour ce paramètre ou de laisser la valeur par défaut. Ensuite il demande confirmation puis modifie la valeur et renvoie la valeur modifiée. La difficulté de ce type de fonctions de types menu dans ce TP, a été la gestion des entrées clavier pour prendre en compte les erreurs de l'utilisateur.

- La fonction ajouter_potions_bdf permet de parcourir la liste des potions contenu dans bdd_potion passée en paramètres puis de les ajouter dans la base de fait en initialisant leur valeur à 0. Elle renvoie la base de faits modifiées.

- La fonction initialiser_params_potions prend en paramètre la base de fait et initialise les différents types de compteurs à 0 avant de renvoyer la base de faits modifiées.

Toutes ces fonctions d'affichage, de menu et d'initialisation simplifie grandement la lisibilité de la fonction-principale par la suite.

Notre fonction principale « fonction-principale » prend en paramètres la base de règle, la base des ingrédients, la base des potions et facultativement la liste des entier correspondant aux ingrédients choisi par l'utilisateur pour éviter de les entrer à la main.
En premier lieu elle donne le contexte à l'utilisateur et initialise les variables niveau, sante et concentration en demandant l'avis de l'utilisateur grâce à choix_paramètre. Ensuite, la fonction initialise la base de faits avec les ingredients par l'appel de la fonction choisir-ingredient. Les niveaux, sante et concentration sont ajoutés dans la base de faits. Le choix du but est fait par l'utilisateur qui est aussi ajouté dans la base de faits. Puis, l'initialisation des potions et des compteurs est fait grâce aux fonctions ajouter_potions_bdf et initialiser_params_potions. Le moteur d'inférence est ensuite lancé par la fonction moteur_chainage_avant dont le résultat est stocké dans result_chainage. Enfin, les règles vérifiées sont imprimées dans le bon ordre en console ce qui affiche l'histoire et la fonction test si le but choisi par l'utilisateur a été vérifié au final.

V. Conclusion

Ce TP03 a été, encore une fois, très enrichissant. Il nous a permis de mettre en application toutes les connaissances en programmation LISP accumulé au cours du semestre en IA01, ainsi que d'utiliser des connaissances d'autre UV avec le développement d'un programme de scraping par exemple. Ce sujet, plus ouverts que les précédents, à pleinement contribués à l'amélioration de notre niveau en LISP, et plus généralement en IA. Le caractère très ouvert du sujet nous a permis de nous concentrer sur un sujet ludique et original, que nous aimons : les potions dans l'univers d'Harry Potter.

Concernant les améliorations possibles, elles sont nombreuses. Nous avions la volonté au départ de faire une interface graphique qui affiche les potions crée par Harry Potter étant donné que nous disposions aussi dans la base de données des photos associé à chaque potion. Malheureusement, cela aurait pris trop de temps à réaliser pour que nous puissions finir le travail dans les temps. Nous pouvons citer comme autre amélioration possible, l'amélioration et simplification globale du code ou encore l'amélioration de la base de règles par l'ajoute de règles (par exemple la prise en compte de la quantité d'ingrédient dont dispose Harry Potter). Nous estimons néanmoins avoir rendu le travail, le plus abouti possible, compte tenu du temps dont nous disposons, ce qui est le rôle principal d'un ingénieur.

Enfin, nous avons bien entendus rencontrés de nombreuses difficultés durant la réalisation de ce projet, mais nous en sommes venus à bout le plus souvent. Nous pouvons par exemple citer, le choix d'un sujet suffisamment original, avec le besoin d'une véritable expertise et des défis techniques. Une autre difficulté a été le débogage du code, nous avons fait face à de nombreuses erreurs de syntaxe, explicable par le fait que notre base de règle et le nombre de variable était conséquent, cela nous a fait perdre beaucoup de temps. En revanche rédaction du moteur et des fonctions de service n'as pas posée trop de problème.

Nous vous remercions pour votre lecture et vous souhaitons une bonne fin de semestre et de bonnes fêtes.