Rapport de projet LO21 : Système Expert

Membres du binôme : CONSTANT Julien et ECHARD Noé

Table des matières

I. Nos choix de conception, d'implémentation et démarche adoptée

Le sujet de notre projet de LO21 porte sur un "système expert". Un système expert est composé de trois partie :

- Une base de connaissance : elle est composée d'une liste de règle.
- Une base de fait : une liste de proposition considérée comme vraie
- Un moteur d'inférence qui permet de déduire une conclusion vraie en comparant la base de fait avec les règles de la base de connaissance.

Pour faire cela, nous avons défini trois types "principaux" de variables. Le type "Regle" est composé de deux types de variables "annexes" : le type conclusion qui est une chaine de caractère et le type prémisse qui est une liste chainée. Le type "BF" (base de fait) est une liste chainée. Enfin, le type "BC" (base de connaissance) est une liste de Regle.

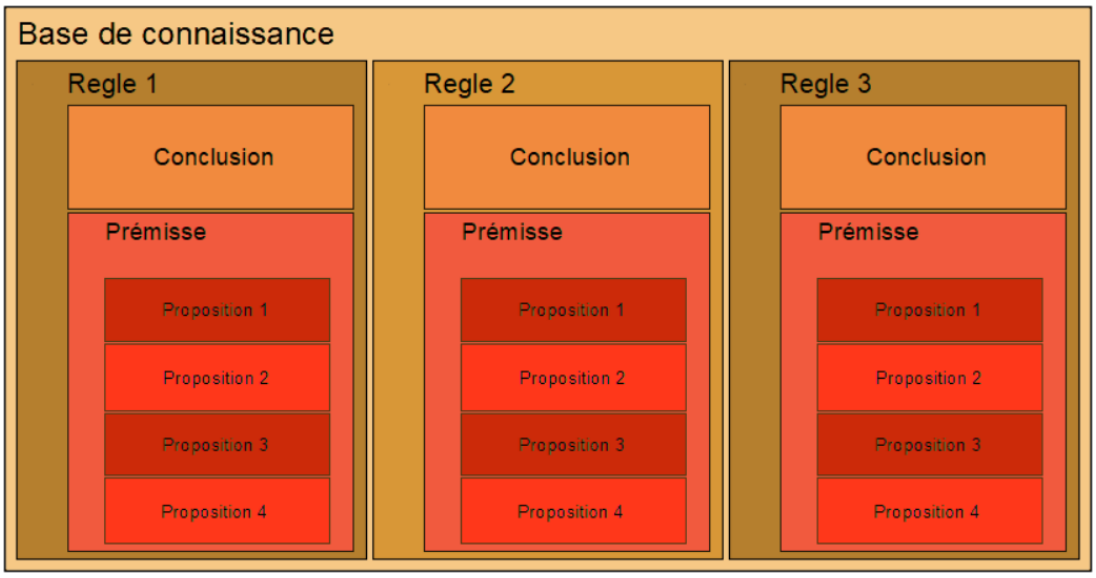
a. Déclaration en détail du type Regle et du type proposition :

```
typedef struct proposition {
        char* content;
        struct proposition* next;
} Proposition;

typedef Proposition* Premisse;
typedef char* Conclusion;

typedef struct regle {
        Premisse premisse;
        Conclusion conclusion;
} Regle;
```

b. Déclaration en détail du type BC (base de connaissance) :

```
typedef struct BC{
    Regle head;
    struct BC* next;
}ElemBC;

typedef ElemBC* BC;
```

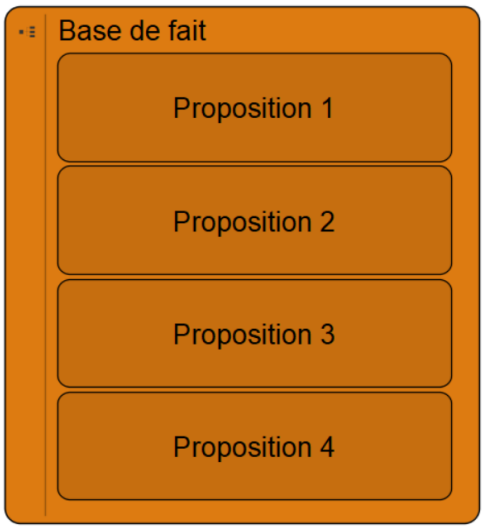
Pour être plus clair, les variables de types BC peuvent être représentées par une classification en groupes emboîtés, ce qui donne ceci :



c. Déclaration du type BF (base de fait) en détail :

```
typedef Proposition* BF;
```

Et la représentation par classification en groupes emboités donne ceci :



II. Algorithmes des sous-programmes

1. Algorithmes des règles

a. Créer une règle vide

On alloue la mémoire nécessaire pour une variable de type règle que l'on retourne ensuite.

```
Fonction créer_règle
Données : pas de données
Résultat : Regle nouvelleRegle
Debut
        nouvelleRegle <- new(regle)
        retourner nouvelleRegle
Fin
```

b. Tester si la règle/prémisse/conclusion est vide

Les trois algorithmes fonctionnent de manière similaire : si la règle, la conclusion, la prémisse ne contient rien, retourner VRAI, sinon retourner FAUX.

```
Fonction est_vide_conclusion
Données : Regle r
Résultat : VRAI si conclusion(r) est vide
Debut
        si conclusion(r) est indéfini alors
                retourner VRAI
        sinon
                retourner FAUX
        fin si
Fin
```

```
Fonction est_vide_premisse
Données : Premisse p
Résultat : VRAI si p est vide
Debut
        si p est indéfini alors
                retourner VRAI
        sinon
                retourner FAUX
        fin si
Fin
```

```
Fonction est_vide_regle
Données : Regle r
Résultat : VRAI si r est vide
Debut
        si r est indéfini
                        retourner VRAI
        sinon
                retourner FAUX
        fin si
Fin
```

c. Ajout d'une proposition à la prémisse (ajout en queue)

On teste si la règle est vide. Si c'est le cas, la prémisse prend la valeur de la proposition. Sinon, on teste si la proposition est déjà dans la prémisse (au cas où). Si ce n'est pas le cas, on ajoute la proposition en queue de prémisse.

```
Fonction ajouter_proposition
Données : Regle r
                        string proposition)
Résultat : Regle r
Debut
        si est_vide_regle(r) est VRAI alors
                nouvelleProposition(proposition)
                premisse(r) <- proposition
                suiv(premisse(r)) <- indéfini
        sinon si est_proposition(premisse(r), proposition) est FAUX alors
                Regle l = r
                        tant que est_vide_premisse(suiv(l)) est FAUX faire
                                l <- suiv(l)
                        fin tant que
                        nouvelleProposition(nouvProposition)
                        contenu(nouvProposition) <- proposition
                        suiv(nouvProposition) = indéfini
                        suiv(l) <- nouvProposition
        sinon
                retourner ERREUR
        fin si
        retourner l
Fin
```

d. Créer la conclusion d'une règle

On teste si la règle est vide. Si c'est le cas, on ne peut rien conclure. Sinon, la conclusion prend la valeur de la proposition.

```
Fonction creer_conclusion
Données : Regle r
                        string proposition
Résultat : Regle r
Debut
Regle l = r
        si est_vide(l) est VRAI alors
                conclusion est indéfini
        sinon
                conclusion(l) <- proposition
Fin
```

e. Tester si une proposition appartient à une prémisse (récursivement)

On teste si la prémisse est vide. Si c'est le cas alors on retourne *FAUX*. Sinon si la proposition de la prémisse est la même proposition que celle que l'on veut tester, alors on retourne *VRAI*. Si ce n'est pas le cas, on retourne la fonction pointant sur l'élément suivant de la prémisse.

```
Fonction est_proposition
Données : Premisse p
                        string proposition
Résultat : VRAI si la proposition appartient à la prémisse
Debut
        si est_vide_premisse(p) est VRAI alors
                retourner FAUX
        sinon
                si contenu(p) = proposition
                        retourner VRAI
                sinon
                        retourner est_proposition(suiv(p), proposition)
                fin si
        fin si
fin
```

f. Retirer une proposition d'une prémisse

Si p est vide, on retourne indéfini. Si *suiv(p)* est indéfini et si le seul élément de p est le même que la proposition alors *libérer(p)* et retourner indéfini, sinon retourner p. (Si p et *suiv(p)* sont définis) Si *contenu(p) = proposition* alors on retourne une prémisse "*résultat*" qui vaut *suiv(p).* Sinon on retourne *retirer_proposition* pour l'élément suivant de p.

```
Fonction retirer_proposition
Données : Premisse p
                        string proposition
Résultat : Premisse p
Debut
        si p est inédini alors
                retourner indéfini
        si suiv(p) est indéfini alors
                si contenu(p) = proposition alors
                        libérer(contenu(p))
                        libérer(p)
                        retourner indéfini
                sinon
                        retourner p
        si contenu(p) = proposition alors
                Premisse résultat = retirer_proposition(suiv(p), proposition)
                libérer(p)
                retourner résultat
        sinon
                suiv(p) = retirer_proposition(suiv(p), proposition)
                retourner p
Fin
```

g. Accéder à l'élément de tête d'une prémisse

```
Fonction acces_tete_premisse
Données : Regle r
Résultat : proposition en tête de prémisse (string) de la regle
Debut
        si est_vide_premisse(premisse(r)) est VRAI alors
                retourner indéfini
        sinon
                retourner contenu(premisse(r))
        fin si
fin
```

On teste si la prémisse de la règle est vide. Si c'est le cas, on retourne indéfini. Sinon, on retourne le contenu du premier élément de la prémisse de r.

h. Accéder à la conclusion d'une règle

```
Fonction acces_conclusion
Données : Regle r
Résultat : conclusion de la règle (string)
Debut
         si est_vide_conlcusion(r) est VRAI alors
                retourner indéfini
        sinon
                retourner conclusion(r)
        fin si
fin
```

On teste si la conclusion de la règle est vide. Si c'est le cas, on retourne indéfini. Sinon on retourne le contenu de la conclusion de r.

2. Algorithmes de la base de connaissance

a. Créer une base vide

On alloue de la mémoire pour une variable de type BC que l'on retourne ensuite :

```
Fonction créerBaseVide
Données : pas de données
Résultat : BC new_basis
Debut
                new_basis <- new(BC)
                retourner new_basis
Fin
```

b. Ajouter une règle à la base de connaissance

On regarde si la base est vide, si tel est le cas, on copie simplement la règle de la base de connaissance, sinon on parcourt la base à l'aide d'une base tampon, puis à l'aide d'un autre tampon on place la règle en queue de la première base tampon.

```
Fonction ajouterRegleBase
Données : BC knowledge_basis
                                Regle r
Résultat : BC knowledge_basis
Debut
        si knowledge_basis est indéfini alors
                        knowledge_basis = r
        sinon
                        BC tampon = knowledge_basis
                        tant que tampon != indéfini
                                        tampon <- suiv(tampon)
                        fin tant que
                        créerBaseVide(tampon2)
                        element_tete(tampon2) = r
                        suiv(tampon) = tampon2
        fin  si
Fin
```

c. Accéder à la valeur de tête de la base de connaissance

On retourne la règle en tête de la base de connaissance

```
Fonction valeurTeteBase
Données : BC konwledge_basis
Résultat : variable de type Regle
Debut
        return valeur_tete(knowledge_basis)
Fin
```

3. Algorithmes du moteur d'inférence

a. Rechercher une UV

On crée plusieurs variables tampons afin de ne pas corrompre nos données, ensuite tant que la base de fait tampon n'est pas vide et tant que la base de connaissance tampon n'est pas vide, on ajoute à une base de faits connus (known_fact) la règle comprenant le fait donné. On retourne alors la base de fait reconnu, celle qui comporte l'ensemble des règles dont la prémisse correspond avec le fait donné.

```
Fonction rechercherUV
Données : BC knowledge_basis
          BF fact_basis
Résultat : BC known_fact
Début
  BC knowldge_buffer = créerBaseVide()
  BC know_fact = créerBaseVide()
  BF fact_buffer = créerListe()

  knowledge_buffer = knowledge_basis
  fact_buffer = fact_basis

  tant que est_vide(fact_buffer) est FAUX faire
    tant que est_vide(knowldge_buffer) est FAUX faire
      Premisse premisse_buffer = premisse(tete(suiv(knowledge_buffer)))
      tant que premisse_buffer != indéfini
        si contenu(premisse_buffer) = contenu(fact_buffer) alors
          known_fact = ajouterRegleBase(know_fact, tete(suiv(knowledge_buffer))
        fin si
        premisse_buffer = suiv(premisse_buffer)
      fin tant que
    knowldge_buffer = suiv(knowldge_buffer)
    fin tant que
    fact_buffer = suiv(fact_buffer)
  fin tant que
  retourner known_fact

Fin
```

b. Moteur d'inférence

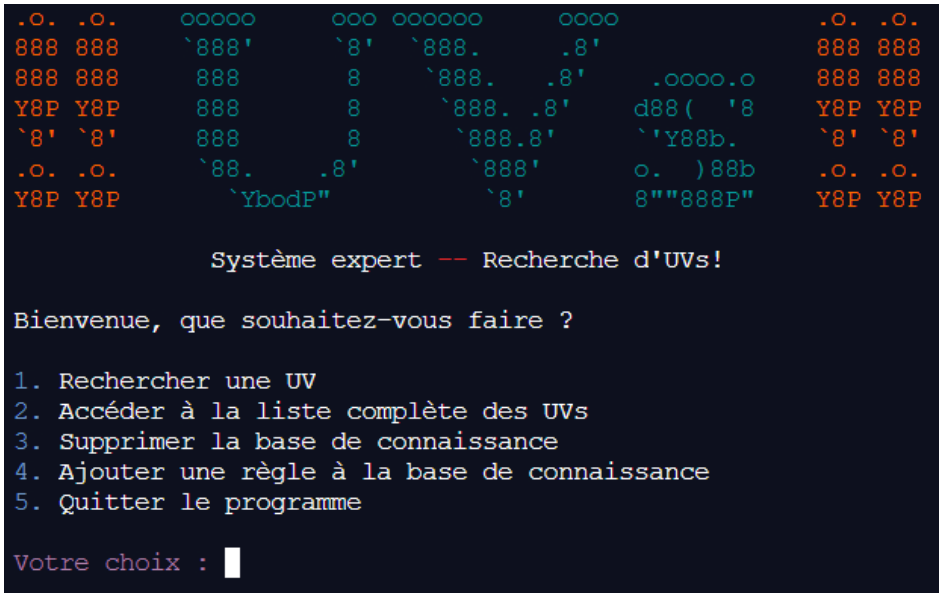
Si la base de connaissance est vide, alors on indique que rien ne peut en être tiré, sinon avec l'aide de variable tampons, tant que la base de fait tampon est définie, on applique rechercherUV à la base de connaissance tampon, puis on avance sur la base de fait, une fois la boucle finie, si la base de connaissance tampon est vide, on indique que l'on n'a rien trouvé, sinon on indique ce que l'on a trouvé.

```
Fonction moteurInference
Données : BC knowledge_basis
          BF fact_basis
Résultat : pas de résultat renvoyé
Début
        Si est_vide(knowledge_basis) est VRAI alors
                afficher("Base de connaissance vide, on ne peut rien déduire")
        sinon
                BC knowledge_buffer = créerBaseVide()
                BF fact_buffer = créerListe()

                knowledge_buffer = knowledge_basis
                fact_buffer = fact_basis

                tant que fact_buffer != indéfini faire
                        knowledge_buffer = rechercherUV(knowledge_buffer, fact_buffer)
                        fact_buffer = suiv(fact_buffer)
                fin tant que

                Si est_vide(knowledge_buffer) est VRAI alors
                        afficher("Aucune UV ne correspond")
                sinon
                        afficher(conclusion(tete(suiv(knowledge_buffer))))
                fin si
        Fin si
Fin
```

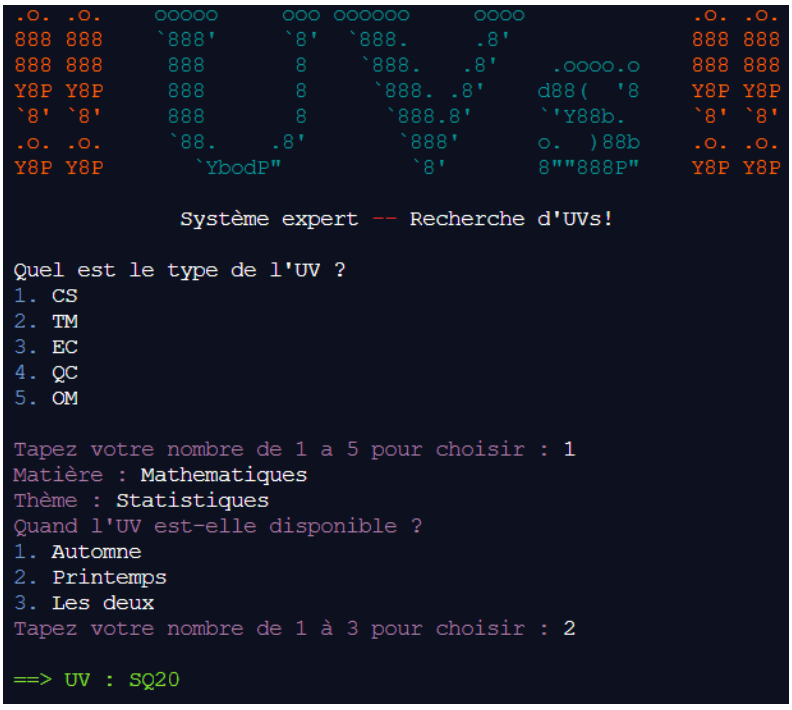
III. Jeu d'essais

Pour le jeu d'essai, nous avons choisi de faire un guide des UV. L'idée est de pouvoir rechercher une UV à partir de plusieurs propositions, comme sa catégorie (CS, TM, etc.), son type (Maths, physique, électricité), un thème particulier ou encore les semestres où l'on peut l'étudier (automne, printemps, ou les deux). Le système peut aussi faire une recherche dans l'autre sens : on lui donne une UV, et il nous renvoie ses caractéristiques.

Capture d'écran du menu principal :

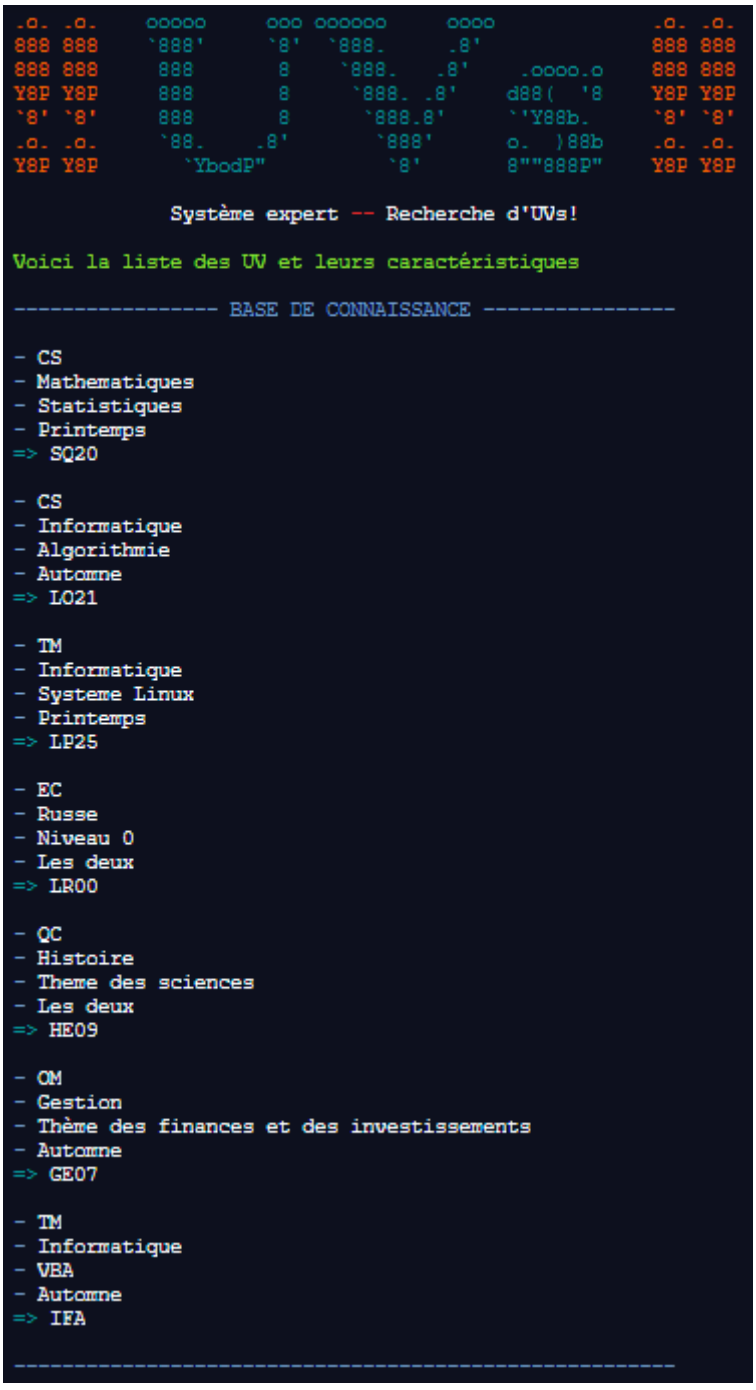


Le menu principal nous propose 5 possibilités d'actions. Si l'on tape 1, on accède au menu de recherche d'UV :



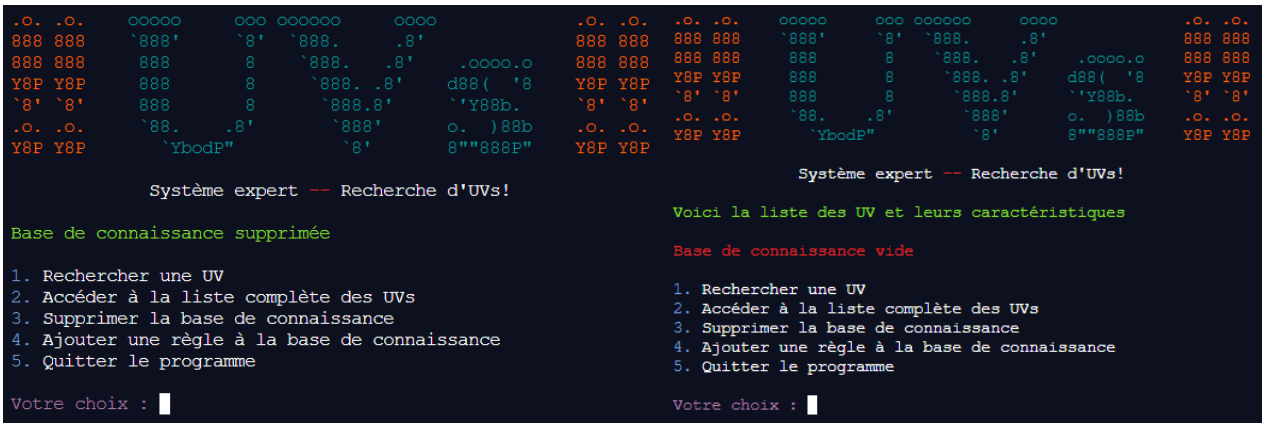
Comme on peut le voir, pour obtenir une UV la recherche s'effectue autour de 4 questions : quel est le type d'UV ? Quelle est la matière ? Quel est le thème ? Et, quand peut-on l'étudier ? Toutes les UV ont une prémisse de la même forme, ce qui permet une recherche précise.

Si l'on tape deux dans le menu principal, on accède à la liste complète des UV, c'est-à-dire, la base de connaissance :



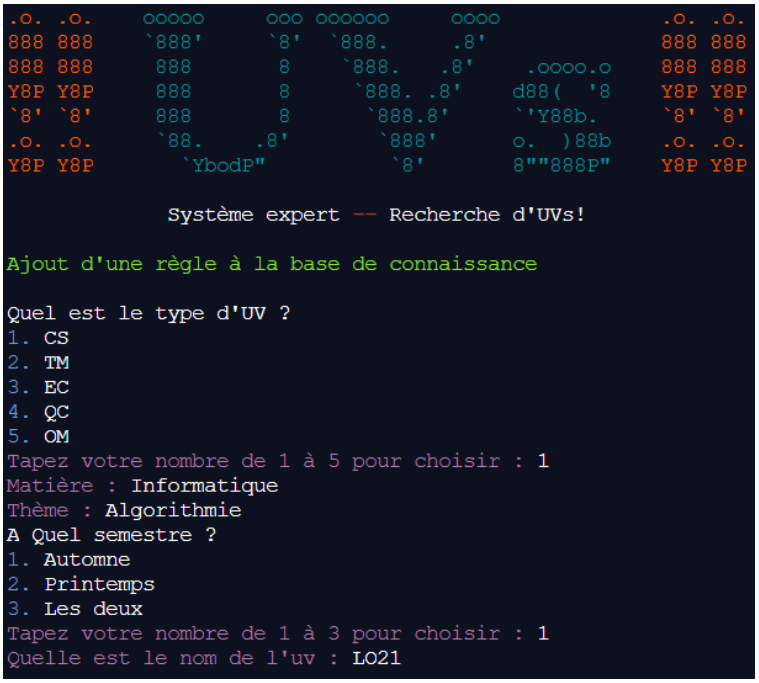
Dans notre cas, nous avons prérempli la base de connaissance de 7 règles pour pouvoir illustrer notre travail avec différentes UVs ayant différentes caractéristiques.

Ensuite, si l'on choisit la 3ème option dans le menu, on réinitialise le système, c'est-à-dire que l'on supprime la base de connaissance. (Voir ci-dessous)



Lorsque l'on supprime la base de connaissance, un message s'affiche nous indiquant que la base a été vidée. Ensuite, si l'on veut afficher la base de connaissance en tapant 2 dans le menu principal, on obtient un message d'erreur indiquant que la base de connaissance est vide.

Enfin, la 4ème option du menu nous premet d'ajouter une règle à la base de connaissance :



Dans notre exemple, nous voulons ajouter l'UV LO21 à notre base de connaissance. Pour cela un menu apparait pour nous demander ses caractéristiques.

Pour finir, taper 5 dans le menu principal permet de quitter le programme.

IV. Commentaires

Nous sommes satisfaits des menus et l'interaction que l'on peut avoir avec ceux-ci. Le moteur d'inférence fonctionne comme demandé dans le sujet, on lui transmet une base de fait et en ressort une conclusion. Le seul défaut que nous avons pu constater dans ce que l'on a fait se trouve dans la fonction qui permet d'ajouter une règle à la base de connaissance : en effet, nous nous sommes rendu compte au dernier moment que l'on ne vérifiait pas si la règle que l'on voulait intégrer faisait déjà partie de la base de connaissance ou pas. La conséquence est que l'on peut mettre plusieurs fois la même règle dans la base de connaissance.

D'un point de vue plus global, ce projet a été pour nous une bonne manière de tester nos compétences en programmation et en algorithmie, et plus précisément notre capacité à concevoir des algorithmes abstraits. Malgré la situation actuelle plutôt atypique et discordante avec le travail, nous avons fait tout notre possible pour aller le plus loin dans ce projet, et cela a été pour nous un défi imposant. Nous sommes plutôt fiers de ce que nous avons pu faire, c'est-à-dire tout ce à quoi nous avions pensé lorsque nous avons commencé le projet.