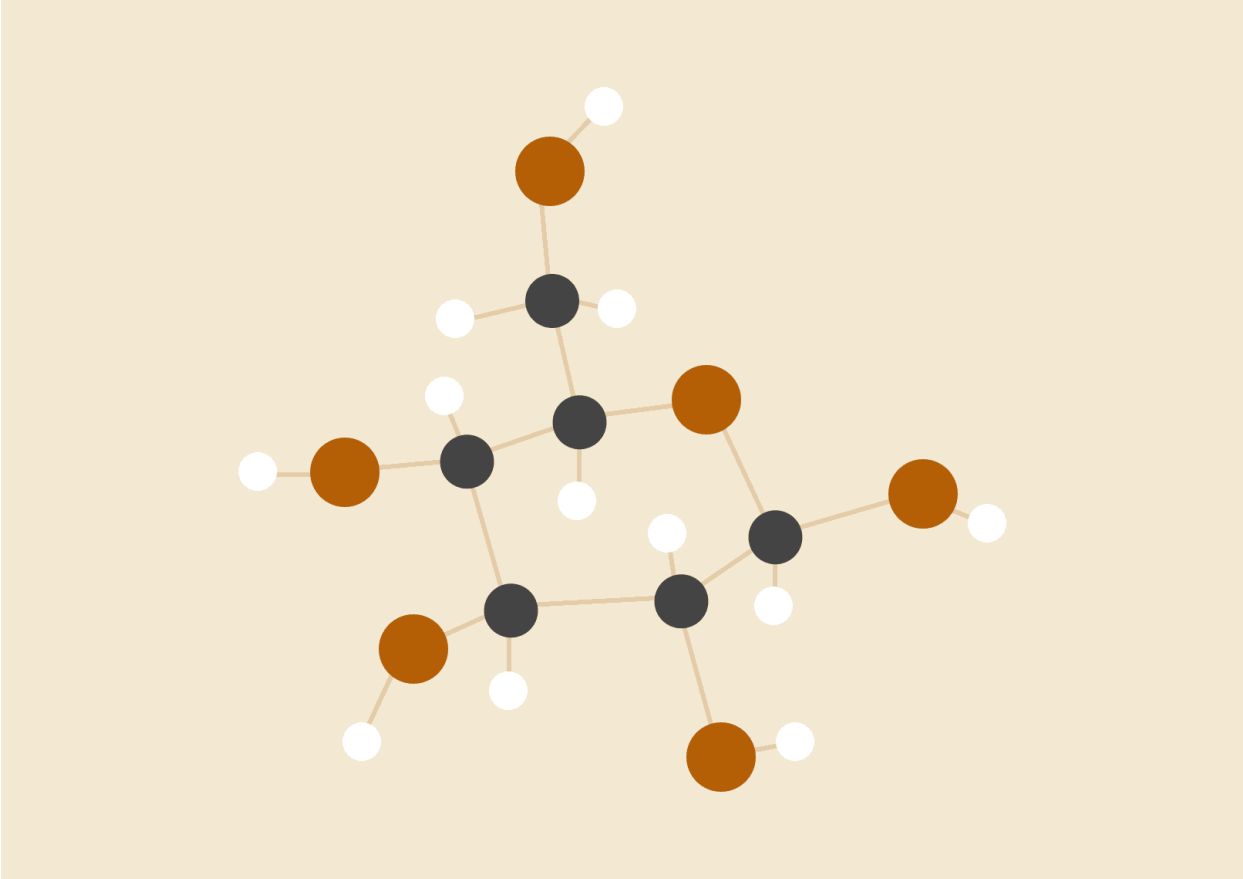
SYSTÈME EXPERT

rapport de projet



THOMAS Yann

A23
LO21

SOMMAIRE

INTRODUCTION

Ce projet consiste à développer un système expert en langage C. Un système expert est, par définition, “un outil informatique d'intelligence artificielle, conçu pour simuler le savoir-faire d'un spécialiste, dans un domaine précis et bien délimité, grâce à l'exploitation d'un certain nombre de connaissances fournies explicitement par des experts du domaine”. Notre système expert sera donc composé d’une base de connaissances, qui représente l’expert à qui on pose les questions, une base de faits, qui correspond à nos questions pour l’expert et à un moteur d’inférence, qui fait le lien entre les deux.

CHOIX D'IMPLÉMENTATIONS

Dans un premier temps, j'ai commencé par créer les structures qui étaient nécessaires à la réalisation du programme. J'ai d'abord créé une structure Proposition avec une valeur (dans ce cas je prends un caractère pour représenter une proposition) et un pointeur sur la proposition suivante. Puis j'ai créé une structure Regle avec une liste de propositions (la prémisse), une conclusion (également un caractère) et un pointeur sur la suivante. Ensuite j'ai créé ma Base de Connaissance et ma Base de Faits, respectivement une liste de règles et une liste de propositions.

J'ai par la suite créé toutes les fonctions demandées puis j'ai réfléchi à une manière de faire fonctionner le tout. Premièrement, j'ai voulu faire quelque chose de légèrement différent du sujet en ne supprimant pas les propositions mais en comparant la règle avec la base de fait pour voir si toutes les propositions de la règle était dans la base de faits. Cela aurait évité d'avoir à supprimer les propositions des règles et aurait pu permettre une utilisation plus poussée du programme avec par exemple plusieurs base de faits pour plusieurs utilisations d'affilés. Pour ce faire j’ai créé la fonction toutesLesPremissesVraies qui regardait si toutes les propositions de la prémisse d'une règle étaient présentes dans la base de faits.

J'ai ensuite traduit tout cela en C puis créé des fonctions pour libérer la Base de connaissance et la base de faits. Après la résolution des quelques problèmes que j'avais, notamment au niveau de la libération de mémoire, je me suis rendu compte qu'un problème était toujours présent et après quelques recherches, j'ai compris que le problème venait du nombre trop important de boucle de récurrence dans le programme, du fait que le programme parcourait la liste de règles un grand nombre de fois pour

vérifier si une règle de la base de fait se trouvait dans la base de connaissance et de même pour les nouveaux faits. Après m'être rendu compte de ça, j'ai essayé de corriger le problème mais après de nombreuses tentatives de correction avec toujours un dépassement de mémoire, j'ai décidé de presque tout reprendre de zéro en changeant le principe de fonctionnement du programme, en suivant celui indiqué par le sujet du projet. J'ai donc façonné ces algorithmes en faisant en sorte qu'une fois un fait reconnu dans une règle de la base de connaissance, la proposition associée dans la règle en question est supprimée. Une fois une règle vide, la conclusion de cette règle est ajoutée à la base de faits. Pour faire cela j'ai pu reprendre mes définitions de structure et quelques fonctions de ma première version en les adaptant à la nouvelle.

Pour ce qui est du moteur d'inférence j'ai donc aussi eu plusieurs versions. Ma première version, comme je l'ai précisé plus tôt, avait pour principe de vérifier si toutes les propositions d'une prémisses étaient vraies pour ajouter sa conclusion à la base de faits. Pour ce faire j'avais plusieurs fonctions. Tout d'abord une fonction globale appliquerMoteurJusquaConvergence() qui copiait la base de fait, appliquait le moteur d'inférence puis comparait la base de fait à la sortie avec celle copiée au début. Si les bases de faits était différentes, cela voulait dire qu'un nouveau fait avait été déduit et qu'il fallait donc relancer le moteur d'inférence avec la nouvelle base de fait. La fonction s'arrêtait quand la base de fait copiée en entrée et la base de fait finale étaient identiques. Dans cette fonction appliquerMoteurJusquaConvergence(), on pouvais retrouver un appel à une fonction basesIdentiques() qui comparait deux bases de faits en sortant un booléen et une autre fonction appliquerMoteurInference() qui appliquait le moteur d'inférence d'une base de fait sur une base de connaissance. Voici donc les algorithmes de la première version (non fonctionnelle) de mon moteur d'inférence :

```
Fonction appliquerMoteurJusquaConvergence(baseDeConnaissance : BC, baseFaits : BF)
DEBUT
      baseFaitsCopie <- BF
      Faire
            baseFaitsCopie <- baseFaits
            appliquerMoteurInference(baseDeConnaissances, propositions(baseFaits))
      Tant que basesIdentiques(baseFaits, baseFaitsCopie) N'EST PAS VRAI
FIN

Fonction appliquerMoteurInference(baseDeConnaissance : BC, baseDeFaits : baseFaits)
DEBUT
      regleCourante <- pointeur sur règle(baseDeConnaissance)
      Tant que regleCourante N'EST PAS nulle faire
            Si toutesLesPremissesVraies(regleCourante, baseDeFaits) alors
                  -> Ajouter la conclusion de la règle à la base de faits
            FinSi
            regleCourante <- suivante(regleCourante)
      Fait
FIN
```

Après avoir changé de stratégie pour le programme, j'ai donc repris de zéro le moteur d'inférence et certaines fonctions pour faire quelque chose qui pouvait coller au mieux au sujet, bien que j'aurais pu reprendre et modifier celui de la première version mais j'ai voulu le reprendre de zéro. J'ai eu également plusieurs versions de ce deuxième modèle de mon moteur d'inférence, suite aux corrections qui ont suivi la résolution de bugs. Le principal problème que j'ai rencontré avec ce nouveau moteur était lié à la libération de mémoire. J'ai pu m'en apercevoir en utilisant le debug proposé par CLion. J'ai pu observer que mon programme plantait après avoir tourné, pendant la libération de mémoire. J'ai donc cherché le problème pendant un bon moment, avant de me rendre compte que j'avais fait une erreur dans mon algorithme et que j'avais libéré la règle avant la prémisse, juste une ligne de code à modifier. Après cela j'ai testé mon programme avec différents sets de valeurs pour vérifier son bon fonctionnement ce qui était le cas et ce fut un grand soulagement et une grande satisfaction que de voir le programme fonctionner après toutes ces difficultés.

DÉROULEMENT DU PROGRAMME

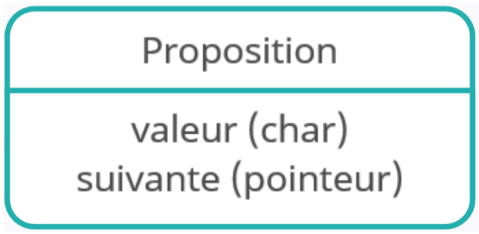
Dans un premier temps, dans le main, on va créer notre base de connaissance et lui ajouter des règles, puis on va créer notre base de faits et y ajouter des faits. On affiche le tout puis on va lancer notre moteur d'inférence sur nos bases de connaissances et de faits.

Le moteur d'inférence va parcourir la base de connaissance règle par règle. Pour chaque règle, on commence par vérifier si la conclusion est déjà dans la base de faits. Ensuite on va regarder la tête de la règle, si la proposition qui est en tête de la règle est déjà dans la base de faits, on la supprime de la règle. On vérifie ensuite si la règle actuelle est vide, et si elle l'est on va ajouter sa conclusion à la base de faits. On passe ensuite à la règle suivante et on répète cette méthode pour toutes nos règles.

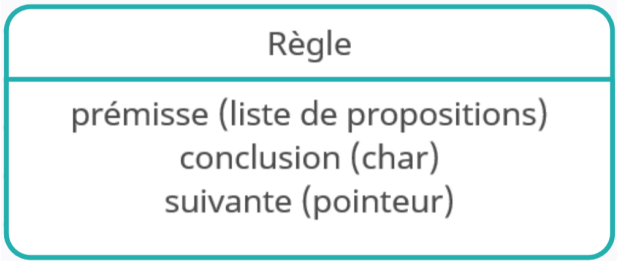
Après l'application du moteur d'inférence on affiche la nouvelle base de faits qui contient donc tous les nouveaux faits qui ont été déduits grâce au moteur d'inférence. On termine le programme en libérant les structures qui ont été créées lors de l'exécution du programme.

STRUCTURES

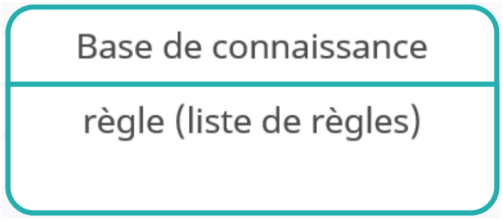
—Structure pour représenter une proposition—



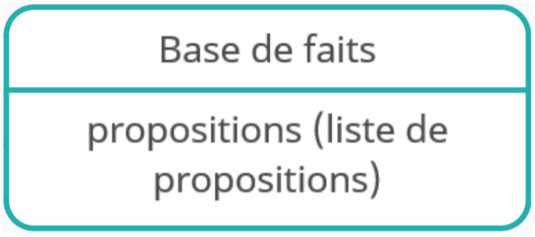
—Structure pour représenter une règle—



—Structure pour représenter une base de connaissances—



—Structure pour représenter une base de faits—



ALGORITHMES

RÈGLE / PROPOSITIONS

- **creerRegle**, une fonction pour créer une nouvelle règle vide. Elle ne prend pas d'éléments en entrée et ressort une règle. nouvelleRegle est une règle de type règle.

```
Fonction creerRegle()->Règle
DEBUT
        nouvelleRegle <- allouer nouvelle règle
        Si nouvelleRegle N'EST PAS NULLE alors
                suivante(nouvelleRegle) <- NULLE
                prémisse(nouvelleRegle) <- NULLE
                conclusion(nouvelleRegle) <- O
        FinSi
        creerRegle <- nouvelleRegle
FIN
```

- **ajouterProposition**, une fonction pour ajouter une proposition (représenté par un caractère) à la fin d'une liste de propositions. Il prends en entrée une règle et un caractère et ne ressort rien. nouvelleProposition est la proposition d'une règle et courant est un pointeur sur une proposition d'une règle.

```
Fonction ajouterProposition(règle : règle, valeur : char)
DEBUT
        nouvelleProposition <- créer proposition
        Si nouvelleProposition N'EST PAS nulle alors
                valeur(nouvelleProposition) <- valeur
                suivante(nouvelleProposition) <- nulle
                Si prémisse(règle) EST nulle alors
                        prémisse(règle) <- nouvelleProposition
                Sinon
                        courant <- pointeur SUR prémisse(règle)
                        Tant que suivante(courant) N'EST PAS nulle faire
                                courant <- suivante(courant)
                        Fait
                        suivante(courant) <- nouvellePorposition
                FinSi
        FinSi
FIN
```

- **supprimerProposition**, fonction servant à supprimer une proposition d'une prémisse et d'une valeur donnée. Elle prend en entrée une proposition et un caractère et ne ressort rien. Courant est un pointeur sur la proposition d'entrée et suivante est un pointeur sur nulle.

```
Fonction supprimerProposition(premisse: Proposition, valeur: char)
DEBUT
        courant <- pointeur SUR premisse
        precedente <- pointeur SUR nulle
        Tant que courant N'EST PAS nul faire
                Si valeur(courant) = valeur alors
                        Si precedente EST nulle alors
                                suivante(courant) <- premisse
                        Sinon
                                suivante(precedente) <- suivante(courant)
                        FinSi
                        liberer(courant)
                FinSi
        precendente <- courant
        courant <- suivante(courant)
        Fait
FIN
```

- **creerConclusion**, fonction servant à remplir la conclusion d'une règle donnée. Elle prend en entrée une règle et un caractère en ne ressort rien.

```
Fonction creerConclusion(règle: regle, valeur: char)
DEBUT
        conclusion(règle) <- valeur
FIN
```

- **conclusionRegle**, fonction servant à renvoyer la conclusion d'une règle donnée. Elle prend en entrée une règle et ressort un caractère qui est la conclusion de cette règle.

```
Fonction conclusionRegle(règle: regle)->char
DEBUT
        conclusionRegle <- conclusion(règle)
FIN
```

-

- **tetePremisse**, fonction servant à renvoyer la tête de la prémisse d'une règle donnée. Elle prend en entrée une règle et ressort un caractère qui est la tête de la prémisse de la règle donnée.

```
Fonction tetePremisse(règle: regle)->char
DEBUT
        Si premisse(règle) N'EST PAS nulle alors
                tetePremisse <- valeur(premisse(règle))
        Sinon
                tetePremisse <- F
        FinSi
FIN
```

- **propositionDansPremisse**, fonction servant à vérifier récursivement si une proposition se situe dans la prémisse donnée d'une règle. Elle prend une proposition et un caractère en entrée et ressort un booléen.

```
Fonction propositionDansPremisse(premisse: proposition, valeur: char)->booléen
DEBUT
        Si premisse EST nulle alors
                propositionDansPremisse <- FAUX
        Si valeur(premisse) = valeur
                propositionDansPremisse <- VRAI
        FinSi
        propositionDansPremisse <- propositionDansPremisse(suivante(premisse), valeur)
FIN
```

- **premisseEstVide**, Fonction servant à vérifier si la prémisse d'une règle est vide. Elle prend une règle en entrée et ressort un booléen.

```
Fonction premisseEstVide(règle: regle)->booléen
DEBUT
        premisseEstVide <- premisse(règle) = nulle
FIN
```

- **creerEtAjouterRegle**, Fonction servant à créer une règle et à l'ajouter à une base de connaissance (en utilisant d'autres fonctions). Elle prend en entrée une base de connaissance, une proposition et un caractère et ressort une règle. nouvelleRegle est la nouvelle règle créée par la fonction.

```
Fonction creerEtAjouterRegle(baseConnaissances: BC ; propositions,conclusion: char)->Regle
DEBUT
        nouvelleRegle <- creerRegle()
        Pour i=0 DANS proposition PAR PAS DE 1 faire
                ajouterProposition(nouvelleRegle, proposition en i)
        Fait
        creerConclusion(nouvelleRegle, conclusion)
        ajouterRegle(baseConnaissances, nouvelleRegle)
        creerEtAjouterRegle <- nouvelleRegle
FIN
```

BASE DE CONNAISSANCES / FAITS

- **creerBaseConnaissances**, fonction servant à créer une base de connaissance vide.Elle ne prend rien en entrée et ressort une base de connaissance nouvellement créée dans nouvelleBase.

```
Fonction creerBaseConnaissances()->Base De Connaissance
DEBUT
        nouvelleBase <- créer base de connaissance
        Si nouvelleBase N'EST PAS nulle alors
                regle(nouvelleBase) <- nulle
        FinSi
        creerBaseConnaissances <- nouvelleBase
FIN
```

- **ajouter Regle**, Fonction servant à ajouter une règle à une base de connaissance. Elle prend en entrée une base de connaissance et une règle et ne ressort rien. courant est un pointeur sur une règle de la base.

```
Fonction ajouterRegle(base: BC, règle: regle)->
DEBUT
        Si regle(base) EST nulle alors
                regle(base) <- règle
        Sinon
                courant <- pointeur sur regle(base)
                Tant que suivante(courant) N'EST PAS nulle faire
                        courant <- suivante(courant)
                Fait
                suivante(courant) <- règle
        FinSi
FIN
```

- **teteBaseConnaissances**, fonction servant à renvoyer la tête de la base de connaissance. Elle prend en entrée une base de connaissance et ressort une règle.

```
Fonction teteBaseConnaissances(base: BC)->Règle
DEBUT
        teteBaseConnaissances <- regle(base)
FIN
```

- **creerBaseFaitsVide**, fonction servant à créer une base de faits vide. Elle ne prend rien en entrée et ressort une base de faits dans la variable baseFaits.

```
Fonction creerBaseFaitsVide()->Base de Fait
DEBUT
        baseFaits <- une base de faits
        propositions(baseFaits) <- nulle
        creerBaseFaitsVide <- baseFaits
FIN
```

- **ajouterFait**, fonction servant à ajouter un fait à une base de faits. Elle prend en entrée une base de faits et un caractère et ne ressort rien. nouvelleProposition est une variable représentant une proposition.

```
Fonction ajouterFait(baseFaits: base de faits, valeur: char)->
DEBUT
        nouvelleProposition <- créer propositions
        valeur(nouvelleProposition) <- valeur
        suivante(nouvelleProposition) <- propositions(baseFaits)
        propositions(basefaits) <- nouvelleProposition
FIN
```

- **faitExiste**, fonction servant à vérifier si un fait existe dans une base de faits donnée. Elle prend en entrée une base de faits et un caractère et ressort un booléen. courante est un pointeur sur une proposition de la base de faits.

```
Fonction faitExiste(baseFaits: base de faits, valeur: char)->booléen
DEBUT
        courante <- pointeur sur propositions(baseFaits)
        Tant que courante N'EST PAS nulle faire
                Si valeur(courante) = valeur alors
                        faitExiste <- VRAI
                FinSi
                courante <- suivante(courante)
        Fait
        faitExiste <- FAUX
FIN
```

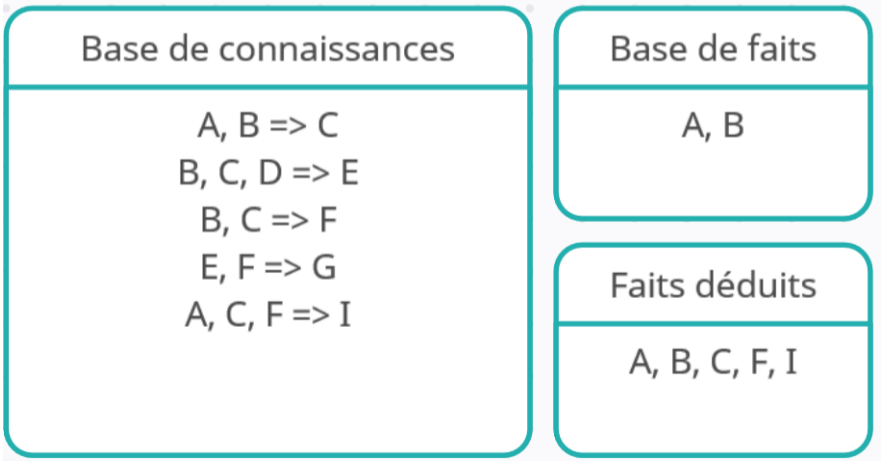
- Fonctions servant à libérer les différentes structures créées lors du programme.
    - Fonction **libererBaseConnaissances**(*baseConnaissances* : BC)
    - Fonction **libererBaseFaits**(*baseFaits* : baseDeFaits)
    - Fonction **libererProposition**(*propositions* : proposition)
    - Fonction **libererRegle**(*règle* : regle)

- L'algorithme du moteur d'inférence servant à relier la base de connaissance et la base de faits. Il prend en entrée une base de connaissance et une base de faits et ne ressort rien. modification est un booléen qui indique si la base de faits à été modifiée au cours de l'exécution du moteur. regle est un pointeur sur une règle de la base de connaissance. tete est un pointeur sur la tête de cette règle.

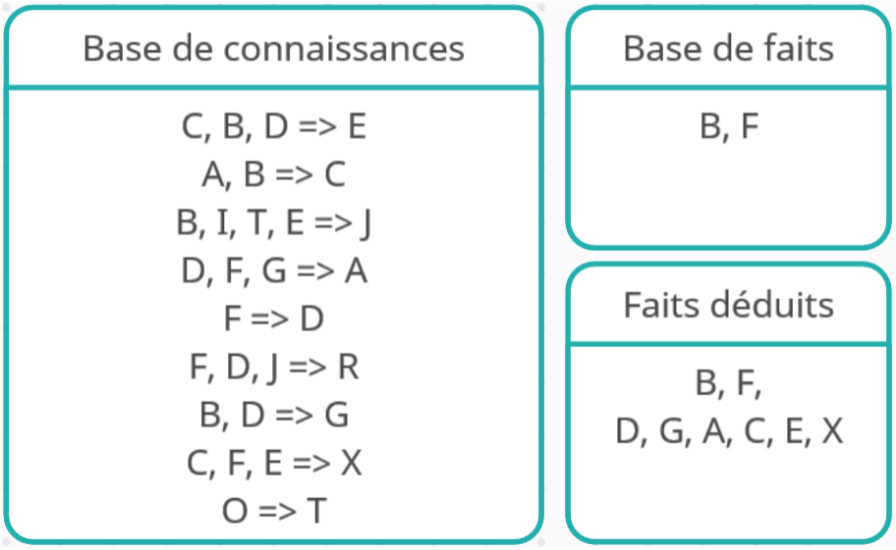
```
Fonction moteurInference(baseConnaissances : BC, baseFaits : baseDeFaits)
DEBUT
        modification <- VRAI
        Tant que modification EST VRAI faire
                modification <- FAUX
                Pour UNE regle <- teteBaseConnaissances(baseConnaissances) AVEC regle NON NULLE ET regle <- suivante(regle) faire
                        Si faitExiste(baseFaits, conclusionRegle(regle)) EST VRAI alors
                                continue
                        FinSi

                        tete <- tetePremisse(regle)

                        Si propositionDansPremisse(propositions(baseFaits), tete) EST VRAI alors
                                supprimerProposition(premisse(regle), tete)
                                modification <- VRAI
                        FinSi

                        Si (premisseEstVide(regle)) EST VRAI alors
                                ajouterFait(baseFaits, conclusionRegle(regle))
                                modification <- VRAI
                        FinSi

                Fait
        Fait
FIN
```

JEUX D'ESSAIS

Voici les jeux d'essais sur lesquels j'ai pu tester mon programme pour vérifier son bon fonctionnement. Le premier set de jeu était simple et à servi de test du programme et pour vérifier son fonctionnement général.



Le deuxième set de jeu à été complexifié pour vérifier si le programme marchait dans tous les cas possibles. J'ai donc créé un set de jeu avec des règles plus complexes qui ne peuvent se déduire que si l'une des règles suivantes à été déduite. Cela permet donc de tester si le programme repasse bien dans la base de connaissance avec les faits nouvellement déduits. Il y a également des règles qui ne peuvent pas être déduites testant le programme la dessus pour éviter d'éventuels faits perdus.



COMMENTAIRES

J’ai pu rencontrer quelques difficultés au cours de ce projet qui m’ont à un moment obligé de presque reprendre de zéro, mais dans l’ensemble le développement du programme à été plutôt aisé, notamment grâce à l’écriture des algorithme en amont qui ont permis une avancé plus méthodique du développement et qui ont été d’une grande aide plutôt que de coder directement en C. J’ai choisi de faire ce projet seul comme un défi pour moi même car jusqu'à présent, j’ai toujours été en groupe avec des personnes plus doué que moi en terme de code pour les projet de groupes donc je ne pouvais pas forcement mettre mon plein potentiel dans les projets, mes idées étant rapidement remplacés par d’autres plus pratique/performantes de mes camarades. C’est pourquoi j’ai voulu faire ce projet seul pour avoir la main sur l'entièreté du développement du projet et chercher par moi même des solutions pour l'améliorer de plus en plus. Je suis finalement plutôt content et satisfait de ce que j’ai pu produire pour ce projet de système expert, concept que je ne connaissais pas et qui m’a beaucoup plu. Cela m’a donné envie de m'intéresser plus en détail à l’histoire des systèmes experts et aux innovations qui ont suivi.