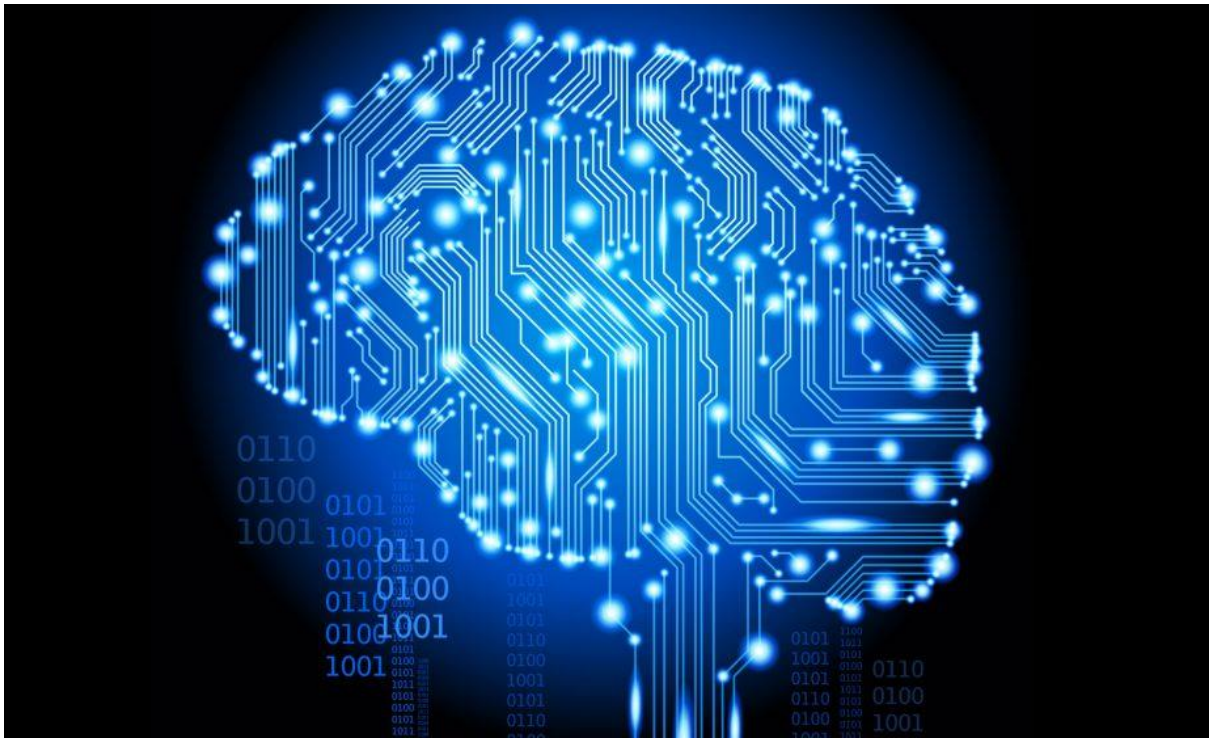
Projet LO21 A2020:

Système expert



PETRY Jules
SZYMANSKI Maxime

Langage C

Table des matières

Durant le premier semestre d'étude de la deuxième année de tronc commun à l'UTBM, dans le cadre de l'UV LO21, les étudiants doivent réaliser un système expert. Ce travail s'effectue en binôme.

1) Présentation du projet et de l'équipe

Notre équipe est composée de Jules PETRY et Maxime SZYMANSKI. Ce projet a été conceptualisé à l'aide d'algorithme et implémenté en langage C, le langage de programmation de l'UV LO21 . Notre projet réside en la création d'un système expert. Notre système expert permet de déduire des faits à partir de connaissance et de faits observés.

2) Concept du projet :

1. Proposition, règle et base de connaissance :

Notre projet consiste en la création d'un système expert. Il est constitué de deux parties : les connaissances et les faits observés. Les connaissances sont une liste de règles. Une règle est elle-même une liste de proposition et une conclusion. Pour ce projet, nous allons considérer les propositions comme des objets qui peuvent valoir « vrai » ou « faux » et qui comportent un nom. Si toutes les propositions d'une règle sont vraies, alors la conclusion de la règle est vraie. Elle repose sur le concept d'implication en mathématiques : *A et B implique C*. La liste des propositions d'une règle est appelée prémisse. Ici, *A* et B sont deux propositions de la prémisse, et *C* la conclusion. Les connaissances sont donc une liste de règle considéré comme vrai et acquise. Nous appellerons l'ensemble des règles « base de connaissance ».

2. Base de fait :

La base de fait est une liste de fait constaté. Le système expert va déduire des faits vrais à partir de la base de fait et de la base de connaissance. Les faits déduits seront alors ajoutés à leur tour dans la base de fait.

3. Exemple d'utilisation de système expert :

Le concept de système expert est fortement utilisé dans beaucoup de domaine. Par exemple, dans le domaine médical, il permet de déduire certaines maladies à partir de faits constatés sur un patient et de connaissances rentrées dans la machine par un médecin expert. *Par exemple « Si le patient a des rougeurs ET des plaques sur le visage ET a de la fièvre ALORS il a contracté la maladie A. ».*
Le système expert que nous avons réalisé est très basique, mais ce concept peut bien sûr se complexifier afin de pouvoir déduire avec plus d'exactitude certains faits. Ce système expert peut aussi, par exemple, être utilisé dans un traitement de données pour des recettes de cuisines : *« Si eau ET sucre ET farine ALORS tarte ».*
Le système expert est un fonctionnement très utilisé et très concret pour traiter des données.

3) Choix de conception du projet

Maintenant que nous avons définies le vocabulaire de la tache à réaliser, nous allons voir comment nous avons concrétisé ce projet.

1. La base de connaissance :

Les faits seront matérialisés par des propositions. Une proposition est un objet qui comporte un nom et une valeur booléenne (VRAI ou FAUX). Une règle est quant à elle un objet qui comporte une liste de propositions, appelé « prémisse » de la règle, une proposition qui représente la conclusion ainsi qu'une chaine de caractère qui représente le nom de la règle. Enfin, la base de connaissance est une liste de règles. Nous avons décidé de séparer la conclusion de la prémisse pour pouvoir manipuler les données avec plus d'aisance. Nous avons aussi mis un nom pour chaque propositions et règles afin de pouvoir traiter les données correctement. Nous pouvons donc avoir facilement accès à la prémisse ainsi qu'à la conclusion d'une règle en ayant son nom.

2. La base de fait :

La base de fait est une liste de propositions. Les propositions de la base de fait seront donc toutes considérées comme vrai. Lorsqu'une proposition est déduite, elle se rajoute dans la base de fait. Nous avons aussi fait une liste de proposition appelée « déductions » qui sert à afficher les faits déduis.

3. Le moteur d'inférence :

C'est l'élément principal de notre programme. En effet, c'est le moteur d'inférence qui va déduire des faits vrais à partir des faits constatés et de la base de connaissance. Son fonctionnement est simple :

Au début du lancement du moteur, nous mettons toutes les prémisses des règles a faux. Ensuite, nous déduisons les faits vrais. Nous parcourons donc chaque règle. Si un fait de la base de fait est dans la règle et que la conclusion de celle-ci est fausse, alors nous mettons la proposition de la prémisse associée au fait à vrai. Sinon, on passe à la règle suivante. Après avoir traité une règle, nous regardons si toutes les propositions de la prémisse de celle-ci sont vraies. Si oui, alors la conclusion de la règle passe à vrai et elle est ajouté à la base de fait. Si non, on passe à la règle suivante. Nous ajoutons aussi la déduction a la liste déduction afin de pouvoir les afficher si nécessaire. Une fois toutes les règles traitées, nous regardons si une des conclusions est passé a vrai dans l'itération. Si oui, alors on refait une itération du moteur d'inférence. Sinon, tous les faits déductibles ont été traité, donc le moteur d'inférence peut s'arrêter.

4) Choix d'implémentation du projet :

Nous avons implémenté ce projet en langage C. Nous avons donc réalisé plusieurs choix :
Les objets propositions, règles, base de connaissance et base de fait ont été traités grâce à des structures. Nous avons également choisi de représenter l'ensemble des listes par des listes chainées, afin d'avoir des listes modulables en taille et en contenue :

Proposition :

La structure proposition contient un booléen (sa valeur) et un pointeur sur un caractère (la chaine de caractère qui représente son nom). Nous avons décidé de mettre un booléen car nous eûmes pensée que c'est la structure de données la plus adaptée au besoin du projet, à savoir prendre la valeur vrai ou faux. Nous avons aussi choisi d'utiliser un pointeur afin de faire une allocation dynamique de la mémoire pour le nom. Nous aurions pu faire un tableau, mais nous avons préféré l'allocation dynamique qui permet d'une part d'éviter les erreurs de mémoires et d'autre part de pouvoir modifier le nom et changer sa taille facilement. Cette implémentation facilite grandement la gestion des propositions, élément centrale du système expert.

Elem :

La structure de données elem est la définition d'un élément de la liste chainée de proposition, utilisé pour représenter la prémisse. Cette structure contient une proposition (la valeur de l'élément de la liste chainée) et un pointeur sur l'élément suivant de la liste chainée. Nous avons défini l'objet « pointeur sur elem » comme l'objet « liste ». Nous avons décidé de faire une liste chainée de proposition afin de pouvoir ajouter ou retirer très facilement des propositions de la prémisse pour modifier des règles.

Regle :

La structure regle contient un pointeur sur un caractère, qui représente le « nom » de la regle et qui permettra d'allouer dynamiquement la mémoire pour pouvoir changer a notre guise le nom de la regle. Une regle contient également un champ conclusion, qui est une structure « proposition ». Et enfin une « liste », qui est une liste chainée de proposition représentant la prémisse de la regle. Nous avons choisi de séparer la conclusion de la prémisse afin de pouvoir accéder à la conclusion plus facilement et a fortiori la modifier aisément. Cette structure va donc nous permettre de grandement faciliter la gestion des regles dans le moteur d'inférence.

Base de connaissance :

La structure BC est un élément de la liste chainée de regle que constitue la base de connaissance. Cette structure est constituée d'une « regle », qui est la valeur de l'élément de la liste et d'un pointeur sur une BC qui est l'élément suivant de la liste de regle. L'implémentation en liste chainée de la base de connaissance est tres intéressante pour modifier les règles et pour accéder successivement a chaque regle de la base de connaissance afin de les traiter avec plus de facilité.

Nous avons défini l'objet « pointeur sur BC » comme l'objet « liste_regle » afin de manipuler la liste chainée plus facilement.

Base de fait :

La structure Base_fait est un élément de la liste chainée qui représente la liste de fait. C'est une liste de proposition. La structure contient donc une « proposition », l'élément de la liste et un pointeur sur l'élément suivant. Nous avons dissocié la structure « Liste » et la structure « Base_fait » afin d'améliorer la clarté du code. Nous avons ensuite défini l'objet « liste_fait » comme un pointeur sur « Base_fait ». L'utilisation de liste chainée est encore une fois très utile car elle permet de modifier facilement la liste ou ses éléments.

Nos choix dans l'implémentation ont été motivés d'une part par les consignes du projet et d'autre part pour la simplicité d'utilisation. En effet, les listes chainées facilitent la gestion des éléments et les structures nous permettent de manipuler les données avec plus de facilité, donc permettent au code d'être plus clair et compréhensible.

5) Algorithmes des sous programmes :

Les types abstraits de données :

Type **Proposition** :

Data : booléen
Nom : chaîne de caractères

Type **Regle** :

Premisse : Liste de proposition
Conclusion : Proposition
Nom_regle : chaîne de caractères

Type **BC** :

Valeur : Regle
Suivant : BC

De plus, nous avons rajouté le type abstrait de donné Liste lors l'implémentation en C afin de faire une liste chainée de proposition pour la prémisse de la regle :

Type **Liste** :

Valeur : proposition
Suivant : Liste

Ainsi que le type BF qui représente la base de fait comme une liste chainée de proposition :

Type **BF** :

Valeur : proposition
Suivant : BF

Fonction : creer_Regle

Données : Ø
Résultat : r Regle
Variables : nv_Regle Regle

Début

premisse(nv_Regle) <- indéfini
data(conclusion(nv_Regle)) <- FAUX
nom(conclusion(nv_Regle)) <- indéfinit

Retourner nv_Regle;

Fin

Lexique : Cette fonction créer une règle vide. Elle ne prend rien en paramètre. Le type Regle étant composée d'une liste du type proposition, cette dernière est initialisée comme indéfini, c'est-à-dire une liste de proposition vide. La donnée booléenne du type Regle est initialisé à **faux** et la donnée **nom** est initialisé à indéfini. La fonction retourne finalement la Regle qui vient d'être créée.

Fonction : Ajout_proposition_premisse

Données : Regle Regle , **prop** Proposition
Résultat : Regle Regle
Variable : temporaire Liste , **nouveau** Liste

Début
nouveau <- créer_liste()
valeur(nouveau) <- prop
suivant(nouveau) <- est indéfini
Si premisse(Regle) = indéfini **Alors**
    Premisse(Regle) <- nouveau
Sinon
    temporaire <- premisse(Regle)
    Tant Que (suivant(temporaire) <> indéfini) **Faire**
        temporaire <- suivant(temporaire)
    Fin Tant Que
    Suivant(temporaire) <- nouveau
Fin Si

Retourner r

Fin

Lexique : Cette fonction prend en paramètre une Proposition ainsi qu'une Règle (qui contient une liste de Propositions). Elle ajoute la Proposition prise en paramètre en queue dans la liste de Propositions de la Règle. Elle retourne ensuite la Règle prise en paramètre à laquelle la Proposition a été ajoutée.

Fonction : creer_Conclusion

Données : nom_CCL chaine de caractere
Regle Regle
Résultat : Regle Regle
Variables : conclusion Proposition

DEBUT

conclusion <- créer_Proposition()
data(conclusion) <- FAUX
nom(conclusion) <- nom_CCL
conclusion(Regle) <- conclusion

Retourner Regle

FIN

Lexique : Cette fonction prend en paramètre une chaine de caractère ainsi qu'une Règle. Elle créée ensuite un élément du type Proposition qui constituera la conclusion de la Règle prise en paramètre. Le nom de l'élément créé est la chaine de caractère prise en paramètre. La donnée booléenne de la conclusion est initialisée à Faux. Finalement, la fonction retourne la Règle avec la conclusion ajoutée

Fonction : appartient_premisse

Données : **premisse** Liste
Prop Proposition
Résultat : appartient Booléen

DEBUT

Si nom(valeur(premisse)) = nom(Prop) **Alors**
Retourner VRAI
Sinon Si est_vide(premisse) **Alors**
Retourner FAUX
Sinon retourner **appartient_premisse**(reste(premisse), Prop)

Fin Si

FIN
Lexique : Cette fonction prend en paramètre une Liste de Propositions ainsi qu'une Proposition. Elle parcoure ensuite la Liste de manière récursive en comparant l'élément actuelle (une Proposition) de la Liste avec la Proposition a trouver dans la Liste. Si un élément appartenant à la Liste est le même que celui recherché, la fonction retourne Vrai, sinon elle retourne Faux (si aucun élément ne correspond à la Proposition recherchée)

Fonction : Test_premisse _vide

Données : **premisse** Liste
Résultat : **est_vide** Booléen

DEBUT

Retourner estvide(premisse)

FIN

Lexique : Cette fonction prend en paramètre une Liste de Propositions. Elle appelle la fonction estvide() et retourne la valeur booléenne retournée par cette dernière fonction. (La fonction estvide() retourne Vrai si la Liste est vide (si la tête de la Liste est indéfinie) et FAUX sinon).

Fonction : tete_premisse

Données : premisse Liste
Résultat : **tete** Proposition

DEBUT

Si estvide(premisse) = VRAI **Alors**
Retourner indéfini
Sinon
Retourner valeur(premisse)
Fin Si

FIN

Lexique : Cette fonction prend en paramètre une Liste de Propositions. Elle appelle ensuite la fonction estvide() et retourne une Proposition indéfinie si la fonction estvide() retourne vrai et sinon retourne la donnée **valeur** de la tête de la Liste prise en paramètre.

Fonction : conclusion_regle

Données : **Regle** Regle
Résultat : **conclusion** Proposition

DEBUT

Si conclusion(Regle) = indéfini
Retourner indéfini
Sinon
Retourner conclusion(Regle)
Fin Si

FIN

Lexique : Cette fonction prend en paramètre une Regle et retourne une Proposition indéfinie si la donnée **conclusion** de la Regle est indéfinie, sinon elle retourne la Proposition **conclusion** de la Regle.

Fonction suppression_Proposition

Résultat: **r** Regle
Données: **r** Regle, **prop** Proposition
Variable: **temp** Liste, **temp2** Liste;

DEBUT

```
si(appartient_premisse(premisse(r), prop) est VRAI) alors
   temp <- premisse(r);
   si (nom(valeur(premisse(r))) = nom(prop)) alors
      si(premisse(r) =est indéfinie) alors
         temp <- indéfini
      sinon
         temp <- suivant(premisse(r))
         liberer(premisse(r))
      Fin Si
   premisse(r) <- temp

   sinon
      Tant que (nom(valeur(suivant(temp))) <> nom(prop) ET suivant(temp) <> indefini) faire
         temp <- suivant(temp)
      temp2 <- suivant(suivant(temp))
      liberer(suivant(temp))
      suivant(temp) <- temp2
      Fin tant que
    Fin Si
sinon
   afficher(« La proposition n'est pas dans la liste »)
Fin Si
FIN
```

Lexique : Cette fonction prend en paramètre une Regle ainsi qu'une Proposition. Elle vérifie ensuite que la Proposition prise en paramètre (qui est vouée à être supprimé par cette fonction) soit bien dans la liste Regle. Pour cela la fonction appartient_premisse() est appelée. Sinon un message d'erreur est affiché dans l'interface utilisateur. Si la Proposition est bien dans la Liste, et que c'est la tête de la Liste, une suppression en tête est effectuée. Sinon la Liste est parcourue jusqu'à ce que l'élément suivant soit celui a supprimé. L'élément suivant est donc changé par celui qui suit celui qui va être supprimé. Enfin, la Proposition prise en paramètre est supprimée. Finalement la fonction retourne la Regle prise en paramètre avec la Proposition supprimé (ou non si rien n'a été supprimé).

Fonction Creer_base_vide

Données : Ø
Résultat : **nouvelle_base** BC
Variables : **nouvelle_base** BC

DEBUT

Suivant(nouvelle_base) <- indefini
Valeur(nouvelle_base) <- indefini
Retourner nouvelle_base

FIN

Lexique : Cette fonction créée un élément du type BC puis initialise sa donnée **suivante** et sa donnée **valeur** à indéfinie. Elle retourne finalement l'élément de type BC créé.

Fonction : Ajout_regle_base

Données : B_C_entre BC
Nouvelle_regle Regle
Résultat : **BC_add** BC
Variables : **nouveau** BC
Temporaire BC

DEBUT

Nouveau <- Creer_base_vide
Valeur(nouveau)<- Nouvelle_regle
Suivant(nouveau) <- indefini

Si estvide(B_C_entre) est vrai **Alors**
B_C_entre <- nouveau
Sinon
Temporaire <- B_C_entre
Tant Que (suivant(temporaire) <> indéfini) **Faire**
Temporaire <- suivant(temporaire)
Fin tant que
Suivant(temporaire) <- nouveau

Fin si

Retourner B_C_entre

FIN

Lexique : Cette fonction prend en paramètre une base de connaissance (Une liste de type BC) ainsi qu'une Regle a ajouté à la base de connaissance. Un nouvel élément du type BC est créer avec la fonction creer_base_vide(). La Regle prise en paramètre est injectée dans la donnée valeur de l'élément BC créé. La donnée suivant de l'élément de type BC créé prend la valeur indéfinie. Finalement un ajout en queue l'élément BC créé est fait dans la liste d'élément de type BC pris en paramètre.

Fonction : tete_base

Données : BC Base_de_connaissance
Résultat Tete Regle

DEBUT

Retourner valeur(BC)

FIN

Lexique : Cette fonction prend en paramètre une liste d'élément de type BC. Elle retourne la valeur de la tête de cette liste (la valeur retournée étant un élément de type Regle).

Fonction : Ajouter_fait
Données : **Nom_fait** chaîne de caractère
Liste_fait BF
Resultat : Liste_fait BF

Variable : Nouveau Liste

DEBUT :

Nouveau **<-** créer_liste()
Data(valeur(nouveau)) <- faux
Nom(valeur(noveau)) <- Nom_fait
Ajouter_queue(BF,valeur(nouveau))

Retourner BF

FIN

Lexique : Cette algorithme ajoute un fait dans la base de fait. Elle prend en entré le nom du fait a ajouter et la base de fait dans laquelle il faut incorporer le nouveau fait. Elle retourne ensuite la base de fait avec le nouvel élément chainé en queue de liste.

Fonction : Ajouter_ccl_fait
Données : **Regle1** Regle
Liste_fait BF
Repetition booléen
Resultat : Liste_fait BF

DEBUT :

Si data(conclusion(Regle1)) est vrai et **fait_deja_existant**(nom(conclusion(Regle1)),BF) est faux **Alors**

BF <- **ajouter_fait**(nom(conclusion(Regle1)),BF)
Repetition <- vrai
Sinon
Repetition <- faux
Fin Si

Retourner BF

FIN

Lexique : Cette algorithme lance la procédure d'ajout d'une conclusion d'une regle a une base de fait si la conclusion de la regle est vrai et qu'elle n'est pas déjà dans la base de fait. Nous avons trois variables d'entrées, Regle1 qui est la regle dont on veut extraire la conclusion, Liste_fait qui est la base de fait ou ajouter la conclusion ainsi que répétition qui vaux vrai si l'on a eu une déduction et faux sinon. La fonction retourne une base de fait qui est en fait la base de fait d'entrée avec, si la condition est vraie, le nouveau fait en queue de liste.

Fonction : Tester_ccl
Données : **Regle1** Regle

Resultat : Regle1 Regle

Variable : temporaire Liste
Test booléen

DEBUT :

Test <- vrai
Temporaire <- premisse(Regle1)

Tant que est_defini(temporaire) **ET** test est vrai **Faire**

Test <- data(valeur(temporaire))
Temporaire <- suivant(temporaire)

Fin tant que

Si test est vrai **Alors**

Data(conclusion(Regle1)) <- vrai

Sinon

Data(conclusion(Regle1)) <- faux

Fin Si

Retourner Regle1

FIN

Lexique : Cette fonction test la valeur de la conclusion. Si toutes les valeurs de la prémisse d'une regle sont vraies, alors la conclusion vaut vrai, sinon elle vaut faux. L'unique paramètre d'entré de la fonction est la regle dont on veut connaitre la valeur de la regle. La valeur de renvoie de la fonction est la regle d'entrée, dont la conclusion a été modifié si nécessaire. Nous avons ajouté une variable temporaire pour parcourir la prémisse et une variable booléenne pour arrêter la fonction si l'on reconnait une proposition de la prémisse qui est fausse.

Fonction : Activer_prop

Données : **Regle1** Regle
Prop Proposition

Resultat : Regle1 Regle

Variable : temporaire Liste

DEBUT :

Temporaire <- premisse(Regle1)

Tant que nom(Prop) <> nom(valeur(temporaire)) **Faire**

Temporaire <- suivant(temporaire)

Fin tant que

Data(valeur(temporaire)) <- vrai

Retourner Regle1

FIN

Lexique : Cette fonction met la valeur d'une proposition de la prémisse a vrai a partir du nom de celle-ci. On avance dans la prémisse jusqu'à trouver la proposition qui a le nom d'entrée, puis on met sa valeur a vrai. Nous n'avons pas besoin de vérifier si la proposition est bien dans la regle, nous supposons que oui lors de l'exécution de ce sous-programme. Nous avons, en entré, la regle dont on veut mettre à vrai une de ses propositions, et la proposition associé. Cette fonction retourne la regle d'entrée, avec la prémisse modifiée. Nous avons utilisé ici aussi une variable temporaire afin de parcourir la prémisse.

Fonction : Inférence_regle

Données : **Regle1** Regle
Base_fait BF

Resultat : Regle1 Regle

Variable : temporaire BF

DEBUT :

Temporaire <- base_fait

Tant que est_defini(temporaire) **Faire**

Si appartient_premisse(premisse(Regle1),Valeur(temporaire)) est vrai **Alors**

Regle1 <- activer_prop(Regle1,Valeur(temporaire)
Fin Si

Temporaire <- suivant(temporaire)

Fin tant que

Retourner Regle1

FIN

Lexique : Cette fonction lance la procédure qui met les propositions constatées, celles de la base de fait, a vrai dans la prémisse de la regle. On parcourt la base de fait, et, si le fait est aussi dans la prémisse de la regle, alors on lance la fonction qui le met à vrai dans la prémisse. Nous avons une regle et une base de fait en entré. On retourne la regle d'entré, a laquelle on a modifié la prémisse en activant les propositions qui sont dans la base de fait. Nous avons derechef utilisé une variable temporaire afin de parcourir la base de fait.

Fonction : Fait_deja_existant

Données : **nom_ccl** Chaine de caractère
Base_fait BF

Resultat : Test Booléen

Variable : temporaire BF
Bool_temp Booléen

DEBUT :

Bool_temp <- faux

Temporaire <- base_fait

Si Test_premisse_vide(Base_fait) est vrai **Alors**

Retourner faux
Sinon Tant que est_defini(temporaire) et bool_temp est faux **Faire**

Si nom(valeur(temporaire)) = nom_ccl **Alors**

Bool_temp <- vrai
Sinon

Temporaire <- suivant(temporaire)
Fin Si

Fin Tant que

Fin Si

Retourner Bool_temp

FIN

Lexique : Cette fonction test si un fait est déjà dans la base de fait. Elle renvoie vrai si oui et faux sinon. Nous avons en données une base de fait dans laquelle nous allons rechercher l'existence d'un fait et une chaine de caractère qui est le nom du fait à rechercher. Nous avons utilisé une variable temporaire pour parcourir la base de fait. Nous avons aussi introduit une variable booléenne car la fonction doit donner l'information « vrai » ou « faux » seulement.

Fonction : Moteur_inference

Données : **BC1** BC
Base_fait BF

Resultat : **Base_fait** BF

Variable : **temporaire** BC
Repetition booléen

DEBUT :

Faire

Repetition <- faux
Temporaire <- BC1

Tant que temporaire <> indéfini **Faire**

Si nom(conclusion(tete_base(temporaire))) <> indéfini **Alors**
Si fait_deja_existant(nom(conclusion(Tete_base(temporaire))), Base_fait) est faux **Alors**

Valeur(Temporaire) <- inference_regle(valeur(temporaire),Base_fait)
Valeur(Temporaire) <- tester_conclusion(valeur(temporaire))
Base_fait <- ajouter_ccl_fait(valeur(temporaire), Base_fait, repetition)
Fin Si
Fin Si
Temporaire <- suivant(temporaire)

Fin tant que

Tant que repetition = vrai

Retourner base_fait

FIN

Lexique : Cette fonction réalise le moteur d'inférence. Elle ajoute dans la base de fait entrée en donnée de la fonction toutes les déductions faites avec les règles qui sont dans la base de connaissance qui est elle aussi en entrée de la fonction. Il y a aussi un booléen qui regarde si on a eu une déduction. Le moteur s'arrête quand l'on a parcouru toutes les règles et qu'il n'y a eu aucune déduction lors du parcours. Elle retourne ensuite la base de fait qui comporte, en plus des faits précédent, les nouvelles déductions du moteur d'inférence.

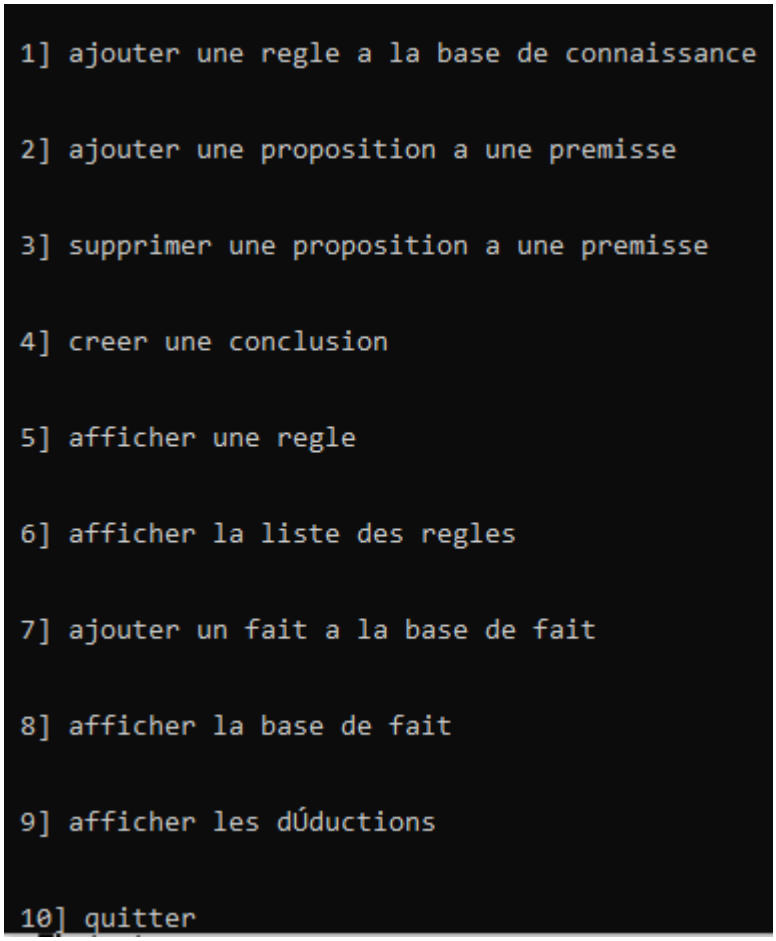
6) Jeu d'essai :

Pour tester notre projet, nous avons réalisé un programme d'essai. Il sagit en fait d'un menu ou l'on peut :

- Ajouter une règle a la base de connaissance
- Ajouter une proposition a une prémisse
- Supprimer une proposition d'une prémisse
- Créer une conclusion
- Afficher une règle
- Afficher la liste de règles
- Ajouter un fait à la base de fait
- Afficher la base de fait
- Afficher les déductions du moteur d'inférence
- Quitter le menu

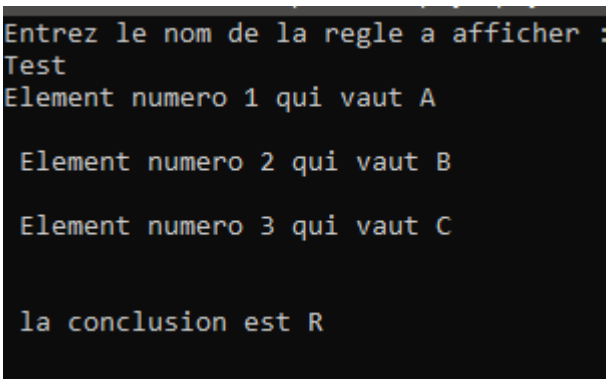
Ce jeu d'essai nous a permis de procéder a l'élaboration de l'implémentation en C du moteur d'inférence. Nous avons séparé, dans le menu, chaque étape du système Expert afin de pouvoir tester toutes les fonctionnalités une par une. Nous savons donc exactement d'où vient le problème quand celui-ci se pose.

Nous avons aussi contrôlé toutes les entrées de l'utilisateur. Ainsi, le programme ne peut pas s'interrompre à cause d'une mauvaise entré. Lorsque l'utilisateur rentre une donnée qui dépasse une certaine taille en mémoire ou est tout simplement incorrect, le programme lui redemande de rentrer cette information jusqu'à ce que la donnée soit conforme.

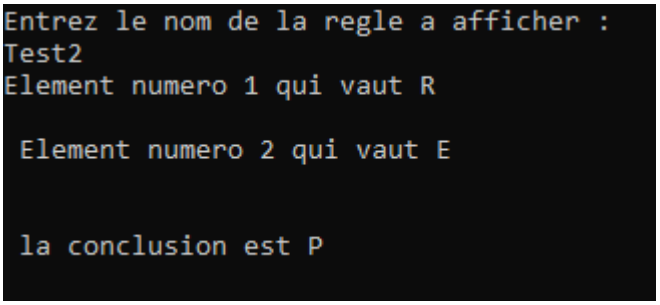


Menu du jeu d'essai

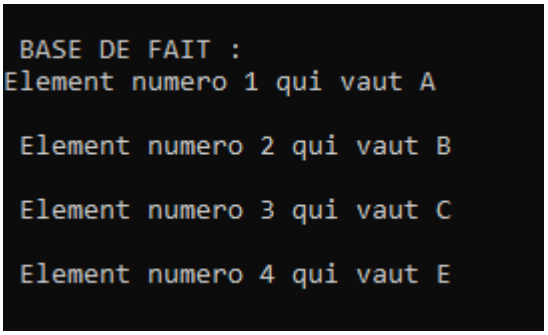
Exemple :



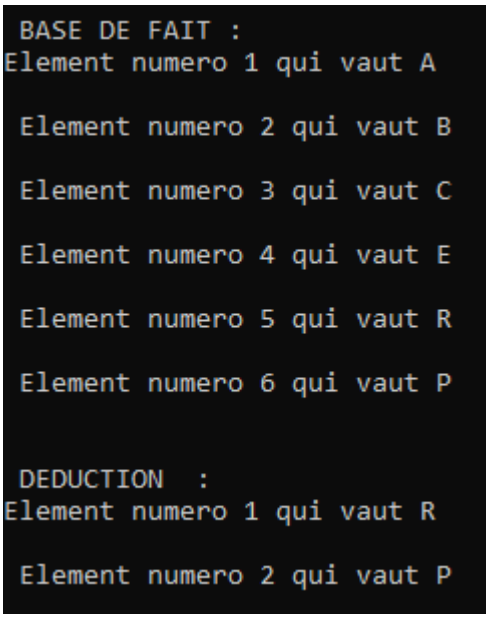
On construit la règle « Test » suivante : A ^ B ^ C => R



Puis une deuxième règle « Test2 » : R ^ E => P



Enfin nous ajoutons A B C et E dans la base de fait



Après exécution du moteur d'inférence, nous avons bien déduis les propositions P et R qui ont été ajouté à la base de fait

7) Exemple d'application du système expert :

Choix du concept :

Lorsque nous avons réfléchi a un exemple d'application, nous avons immédiatement pensé à une aventure textuelle. Nous avons également cherché des applications pour le moteur d'inférence et nous nous sommes rendu compte que la « table de Craft » du célèbre jeu Minecraft de Microsoft est une très bonne application pour le moteur. En ayant certains matériaux on peut en concevoir de nouveaux et avec ces derniers matériaux nouvellement créés, nous pouvons en fabriquer d'autres.
Nous sommes donc partis sur ces deux idées qui nous tenaient à cœur, tout comme le fait d'exploiter toutes (ou presque toutes) les compétences en C que nous avons acquise au cours du temps, notamment en suivant les UVs de IFB/IFE et LO21.

Principe et but du jeu :

Le principe du jeu est simple, vous incarnez un personnage qui est capable de se déplacer dans un monde fait de plusieurs environnements. Vous pouvez interagir avec ces environnements afin de récupérer des matières premières qui vous permettrons de fabriquer des objets. Ces objets serviront à en fabriquer d'autres ou bien acquérir des bonus. Pour réaliser toutes ces choses, vous devrez entrer des commandes dans la console.
Le but du jeu est d'avoir un maximum de points de score à la fin de la partie. Vous pourrez en gagner en explorant la carte du jeu, en tuant des monstres et en fabricant des objets. La partie se termine lorsque vous n'avez plus de vie ou quand vous abandonnez en quittant le jeu.

Comment le jeu fonctionne en C ?

Joueur : Le joueur est une structure composée du nom du joueur (saisie par le joueur lui-même), du lieu du joueur (toujours à 0 en début de partie), de ses points de vie, de son inventaire ainsi que de son score.

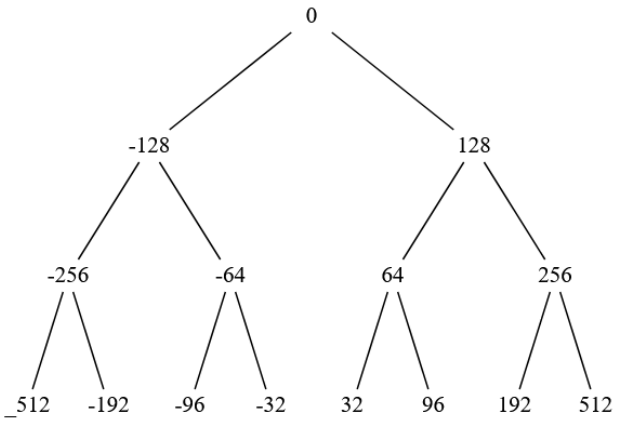
Inventaire du joueur : L'inventaire du joueur est une liste chainée d'élément composé d'une structure ayant pour données, le nom de l'item et sa quantité. Ainsi, lorsqu'on fait la commande « inventaire », la liste chainée est affichée en console.

Monstre : Chaque donjon comporte 1 monstre. Chaque monstre est une structure composée du type de monstre et de ses points de vie.

Lieu : Chaque lieu est composé de son nom (son type), de son identifiant unique, d'un booléen qui permet de savoir s'il comporte un monstre ou non et d'un monstre.

La carte du jeu : C'est un arbre de recherche. Lorsque le jeu commence, l'arbre est composé d'une racine qui contient la valeur 0, d'un fils gauche qui contient la valeur -128 et d'un fils droit qui contient la valeur 128. Chaque nœud (racine et feuilles comprises) représente un lieu. Chaque lieu est caractérisé par un Identifiant unique. Lorsque le joueur se déplace dans un nouveau lieu, un fils droit et un fils gauche sont créés et ajouter au nœud ou le joueur s'est déplacé. Cela permet de créer un arbre infini (dans la limite du nombre d'Integer).

Le type de lieu (foret, grotte, désert ou donjon) est ensuite déterminer aléatoirement pour chaque nouveau lieu. La commande « aller » permet de déplacer le joueur d'un lieu à un autre en changeant la donnée lieu de la structure joueur par l'identifiant unique du lieu où le joueur veut aller.



Représentation de l'arbre binaire de recherche

La commande « craft » : Lorsque le joueur saisie cette commande, il fait appel au moteur d'inférence. Un fichier .txt regroupe l'ensemble des règles. Ce fichier est composé de la manière suivante. Chaque mot suivi du caractère « * » est un nom de règle. Ensuite, tous les mots suivant le nom de la règle et ayant le caractère « ; » juste à côté sont des éléments de la prémisse de la règle. Enfin, le mot suivant les éléments de prémisse et ayant le caractère « : » juste à côté est la conclusion de la règle. Exemple d'une fabrication possible : une épée composée d'une bûche et d'un fer.
En modifiant le fichier .txt, on peut ainsi modifier les Craft disponibles pendant le jeu.

```
epee *
fer ;
buche ;
epee :
```

La commande « regarder » : Cette commande réalise simplement une lecture du fichier .txt correspondant au lieu où le joueur se trouve. Le fichier contient un dessin en ASCII ainsi qu'une petite description du lieu.

8) Conclusion :

A l'issu de la réalisation de ce projet de LO21, nous avons pu créer le moteur d'inférence, et l'utiliser dans un petit jeu. Ce projet nous a permis de découvrir les bases de l'intelligence artificiel. Le jeu d'essai est concluant. En effet, nous sommes très satisfaits du résultat obtenu, c'est exactement ce que nous avions prévu de faire au départ. Le jeu d'essai est parfaitement fonctionnel et les differents test nous ont permis d'optimiser le codeS. Nous pouvons encore améliorer le moteur d'inférence, en mettant par exemple plusieurs possibilités dans les propositions plutôt qu'un simple booléen.
Ce projet nous a permis de consolider nos bases en algorithmie et en C, ainsi que les méthodes de travail de groupes.