# Rapport Projet LO21 : système expert

Cahier des charges

Contextes et définition du problème

Le sujet de ce projet est de réaliser un système expert fonctionnel. D'après la définition donnée par la page Wikipédia sur ce sujet, un système expert est : "un logiciel capable de répondre à des questions, en effectuant un raisonnement à partir de faits et de règles connues."(source : https://fr.wikipedia.org/wiki/Syst%C3%A8me\_expert). Pour cela le système expert va se comporter comme un humain expert dans un domaine d'activité comme la médecine ou l'automobile. En effet pour établir un diagnostic ou une analyse l'humain va faire des observation et à partir de ces observation il va appliquer des règles qu'il a apprise lors de sa formation ou par son expérience pour en tirer des déduction, comme par exemple la maladie dont souffre un patient dans le cas d'un médecin ou le problème qu'a la voiture pour un garagiste. Le système expert est donc composé de trois élément principaux, le premier est la base de connaissances. La base de connaissance permet de stocker toutes les règle qui permettront au programme de faire des déductions. Le second élément est la base de fait, elle permet de stocker les propositions qui sont considérée comme vrai au lancement du programme. Le dernier élément est le moteur d'inférence, c'est lui qui est chargé à partir de la base de connaissances et de la base de fait de déduire toute les proposition qui sont vraies.

Objectif du projet

L'objectif de ce projet est de développer une programme capable de remplir les fonction d'un système expert. Le projet doit pouvoir permettre à l'utilisateur de créer ou importer une base de connaissances, puis après avoir répondu à des questions permettant d'établir la base de fait de connaitre les résultats qui peuvent en être déduit. On souhaiterait permettre à un utilisateur n'ayant pas de connaissances en informatique de pouvoir utiliser notre projet une fois qu'il a été configuré en fonction de la situation.

Périmètre du projet

Le projet devra être réaliser en langage C et s'exécuter dans la console. Il sera au maximum portable c'est à dire capable de fonctionner sous différent systèmes d'exploitation (linux et Windows au moins). La vitesse d'exécution et la place prise en mémoire sera optimiser afin d'obtenir un programme le plus efficace possible. Nous limiterons les règle contenue dans la base de connaissances à des règle du type "A et B et ... implique C" et nous utiliserons uniquement les implication directe, pas de réciprocité ou de contraposée.

Description fonctionnelle des besoins

* Définir des structure de données abstraite pour les objets suivant : Proposition, Prémisse, Règle, Base de connaissance, Base de fait.
* Implémenter les sous programmes permettant de manipuler les structure
* Permettre à l'utilisateur de créer et de stocker une base de connaissances
* Permettre à l'utilisateur de répondre au question et de créer la base de fais en fonction des réponses.
* Implémenter le sous-programme qui remplit la fonction du moteur d'inférence
* Afficher à l'utilisateur le contenue de la base de connaissance et les résultat du moteur d'inférence

Contrainte/ Normalisation et documentation du code

* Le programme doit respecter la norme C99
* Chaque fonction ou structure doit être documentée selon la formalisation imposée par doxigène. Ainsi une documentation du code sera générée par doxigène
* Les fonction seront dans la mesure du possible implémenté de manière récursive
* Les variable sont en normalisation CamelCase et les nom de structure commences par une majuscule
* Le projet est versionner avec git et héberger sur git hub
* La génération des make file et la compilation du programme ce fait avec Cmake
* Le temps d'exécution et l'espace mémoire utilisé par le programme devront rester acceptable pour que le programme soit utilisable sur le plus grand nombre de machine

## Réalisation pratique

### Structures de donnée

* La structure de donnée abstraite qui va nous permettre représenter une Règle est composée de 2 sous éléments : la prémisse et la conclusion.
  + Premisse est du type Premisse que nous définirons ci-dessous.
  + Conclusion est de type pointeur sur une Proposition que nous définirons ci-dessous.

typedef struct Regle

{

    Premisse       premisse;

    Proposition    \*conclusion;

}Regle;

* La structure de donnée abstraite qui va nous permettre de représenter une Premisse est une liste chainée de Proposition. La Premisse est donc un pointeur sur le premier élément de la liste qui est de type PremisseElem que nous définirons ci-dessous.

typedef PremisseElem\* Premisse;

* La structure de donnée abstraite PremisseElem est constituée de 2 sous variables : la valeur de l’élément et le pointeur sur l'élément suivant de la liste chainée :
  + valeur est de type pointeur sur une Proposition que nous définirons ci-dessous
  + elemSuivant est un pointeur sur PremisseElem

typedef struct PremisseElem

{

    Proposition         \*valeur;

    struct PremisseElem \*elemSuivant;

}PremisseElem;

* La structure de donnée abstraite qui va nous permettre de stocker les Proposition est composée de 2 sous variable : la description et la validité :
  + description est une chaine de caractère (tableau de char en C) qui correspond à ce que veut dire la proposition en lagunage naturel
  + validite est un booléen qui vaut true si la proposition est vrai et false si la proposition est fausse ou si son état est inconnue.

typedef struct Proposition

{

    char \*description;

    bool validite;

}Proposition;

La base de connaissances est une liste chainée de Règle, nous avons déjà définit la structure de donnée règle ainsi que toutes les fonction qui lui sont associer, il faut maintenant rajouter une structure permetant de stocker un élément de cette liste chainée et la structure qui fera office de tête de liste

* La structure de donnée abstraite BDConnaissancesElem permet de représenter un élément de la liste chainée de règles. Elle est composée de deux sous-variables, un pointeur sur Règle ainsi qu’un pointeur sur l'élément suivant dans la liste chainée

typedef struct BDConnaissancesElem

{

    Regle\* valeur;

    struct BDConnaissancesElem\* suivant;

}BDConnaissancesElem;

* La structure de donnée abstraite BDConnaissances permet de représenter la base de connaissance, c'est un pointeur sur le premier élément de la liste chainé de règle.

typedef BDConnaissancesElem\* BDConnaissances;

### Algorithme des fonctions :

* newRegle : créer une nouvele règle vide
  + donnée : rien
  + résultat : un pointeur sur la règle qui vient d'être créée

fonction : newRegle(): \*Regle

    Soit nouvelRegle une Regle

    nouvelRegle.premise <-- NULL

    nouvelRegle.conclusion <-- NULL

    newRegle <-- nouvelRegle

fin fonction

* deleteRegle : supprimer une règle et toutes ces composantes
  + donnée : regleToDelete, un pointeur sur règle, la règle que l'on veut supprimer
  + résultat : supprime de la mémoire la règle

procédure : deletRegle(Regle \*regleToDelete)

    deletePremisse(regleToDelete.premisse)

    libère(regleToDelete)

fin procédure

* deletePremisse : supprimer de la mémoire une prémisse
  + donné : un pointeur sur la prémisse à supprimer
  + résultat : supprime de la mémoire la prémisse

procédure : deletePremisse(Premisse prem)

    si (premisseNonVide(prem))

        deletePremisse(suivant(prem))

        libère(prem)

    fin si

fin procédure

* Ajouter une Proposition à la prémisse d'une règle en queue
  + donnée : la proposition à ajouter et la Règle à laquelle l'ajouter
  + résutat : ajoute la proposion à la liste chainé en queue en faisant le lien entre les élément

fonction :  insertTailPremisseRegle(Regle\* regle, Proposition \*prop)

    premisse(regle) <-- inseretTailPremisse(premisse(regle), prop)

    insertTailPremisseRegle <-- regle

fin fonction

* insertTailPremisse : ajouter une proposition à une premisse en queue
  + donnée : la premisse à laquel on veut ajouter la proposition et la proposition à ajouter
  + résultat : renvoie la valeur de la premmise et ajoute la propositon à la prémisse

fonction : insertTailPremisse(Premisse prem, Proposition\* prop)

    si premisseVide(prem)

        soit newElem un PremisseElem

        valeur(newElem) <-- prop

        elmSuivant(newElem) <-- NULL

        insertTailPremisse <-- newElem

    sinon si premisseVide(elemSuivant(newElem))

        soit newElem un PremisseElem

        valeur(newElem) <-- prop

        elmSuivant(newElem) <-- NULL

        elemSuivant(prem) <-- newElem

        instertTailPremisse <-- prem

    sinon

        insertTailPremisse(elemSuivant(prem), prop)

        insertTailPremisse <-- prem

    fin si

fin fonction

* addConclusion : Créer la conclusion d'une règle
  + donné : la règle que l'on veut modifier et la proposition à donner à la conclusion
  + résultat : modifie ou initialise la conclution de la règle

procédure : addConclusion(Regle\* regle, Proposition\* prop)

    conclusion(regle) <-- prop

fin procédure

* propositionDansPremisse: Tester si une Proposition appartient à la prémisse d'une règle récursivement
  + donnée : un pointeur sur le premier élément de la prémisse dans laquel on veut rechercher, la propositon à rechercher
  + résulat : renvoie vrai si la proposion à été trouvée dans la prémisse et faux sinon

fonction : propositionDandPremisse(Premisse prem, Proposition\* prop)

: booléen

    si premisseVide(prem)

        propositionDansPremisse <-- faux

    sinon si valeur(prem) = prop

        propositionDansPremisse <-- vrai

    sinon

        propostionDansPremisse <--

        propostionDansPremisse(elemSuivant(prem))

    fin si

fin fonction

* rechercheSupprimePremisse :Supprimer une Proposition de la prémisse d'une règle
  + donnée : la prémisse dans laquelle on veut supprimer, la proposition à supprimer
  + résultat : supprime la proposions si elle a été trouvée et renvoie la prémisse sur laquelle on travail

fonction : rechercheSupprimePremisse(Premisse prem,

Propostion\* prop) : Premisse

    si premisseVide(prem)

        rechercherSupprimePremisse <-- NULL

    sinon si premisseVide(elemSuivant(prem)) et description(prop) =

description(valeur(prem))

        libère prem

        rechercherSypprimePremisse <-- NULL

    sinon si description(prop) = description(valeur(elemSuivant(prem)))

        Soit toDelete une Premisse

        toDelete <-- elemSuivant(prem)

        elemSuivant(prem) <-- elemSuivant(toDelete)

        libérer toDelete

        rechercherSupprimePremisse <--  prem

    sinon

        rechercheSupprimePremisse(elemSuivant(prem), prop)

        rechercheSupprimePremisse <-- prem

    fin si

fin fonction

* reglePremisseIsEmpty :Tester si la prémisse d'une règle est vide
  + Donées : un pointeur sur la règle à tester
  + Resultat : renvoie vrai si la prémisse est vide et faux sinon

fonction : reglePremisseIsEmpty(Regle \*regle) : booléen

    reglePremisseIsEmpty <-- premisseIsEmpty(premisse(regle))

fin fonction

* returnHeadPremisse : Accéder à la proposition ce trouvant en tête d'une prémisse
  + donnée : la premisse sur laquelle on veut travailler
  + renvoie : pointeur sur la proposition en tête de la premisse

fonction : returnHeadPremisse(Premisse prem) : Proposition \*

    returnHeadPremisse <-- valeur(prem)

fin fonction

* conclutionRegle : Accéder à la conclusion d'une règle
  + donnée : un pointeur sur la règle dont on veut connaitre la conclusion
  + résultat : renvoie un pointeur sur la proposition qui est la conclusion de la règle

fonction : conclutionRegle(Regle\* regle) : Proposition\*

    conclutionRegle <-- conclution(regle)

fin fonction

* isEmptyBDC : tester si la base de connaissances est vide
  + donnée : la base de connaissances que l'on veut tester
  + résultat : renvoie vrai si la basse de connaissance est vide et faux sinon

fonction : isEmptyBDC(BDConnaisances bdc)

    si bdc = NULL

        isEmptyBDC <-- vrai

    sinon

        isemptyBDC <-- faux

fin fonction

* addHeadBDC : insérer en tête une nouvelle règle
  + donnée : la base de connaissance avec laquelle on travail et un pointeur sur la règle à ajouter
  + résultat : ajoute la règle à la base de connaissance et renvoie la base de connaissance

fonction : addHeadBDC(BDConnaissance bdc, Regle regle): BDConnaissances

    soit newElem un pointeur sur BDConnaissanceElem

    valeur(newElem) <-- regle

    suivant(newElem) <-- bdc

    addHeadBDC <-- newElem

fin fonction

* deleteHeadBDC : supprimer en tête une règle
  + donnée : la base de connaisance dont on veut supprimer le premier élément
  + supprime l'élément en tête de la liste chainée et renvoie la base de connaissance

fonction : deleteHeadBDC(BDConnaissances bdc) : BDConnaissances

    si isEmptyBDC(bdc) = faux

        soit toDeleteNext un pointeur sur un BDConnaissancesElem

        toDeleteNext <-- suivant(bdc)

        deleteRegle(valeur(bdc))

        libère bdc

        deleteHeadBDC <-- toDeleteNext

    sinon

        deleteHeadBDC <-- NULL

fin fonction

* returnHeadBDC : Accéder à la règle se trouvant en tête de la base
  + donné : la base de connaissances dont on veut connaitre la prémière règle
  + résultal : renvoie un pointeur sur la règle en tête

fonction : returnHeadBDC(BDConnaissances bdc) : Regle\*

    returnHeadBDC <-- valeur(bdc)

fin fonction

Lecture et écriture dans les ficher

(Carlo c’est pour toi)

Gestion de la mémoire