# IA01 Rapport Projet n°3: Système Expert

*Pour quelle association de l'UTC êtes- vous fait?*

**Par Victoria BOURGEAIS, Martin HEBANT et Emmanuelle LEJEAIL**

# Table des matières

# Introduction

Ce dernier projet réalisé dans le cadre de l'UV IA01 a pour sujet les Système Experts. Le principe d'un système expert est de fournir un diagnostic, des connaissances sur un domaine bien particulier. Il existe plusieurs types de systèmes experts que l'on regroupe par "ordre". Le premier d'ordre 0 repose sur un système de faits à valeur booléenne. L'ordre 0+ est dans la continuité de l'ordre 0, mais les faits sont représentés par des couples ou des triplets. C'est le type de système qu'il nous a été demandé de réaliser. Cependant, étant en trinôme nous avons choisi de réaliser un système d'ordre 1 dont le fonctionnement est légèrement plus complexe.

Notre système d'ordre 1 possède des règles dont la valeur des triplets (attribut objet valeur) sont des variables. Nous détaillerons toutes les subtilités de notre programme dans les parties suivantes.

# Choix du sujet

Nous sommes tous les 3 à l'UTC depuis 2 ans et demi. Au cours de ces deux années, nous avons tous les trois eu l'occasion de nous engager auprès de diverses associations de l'école. Nous avons également remarqué la disparition de certaines associations faute de personne pour les reprendre. C'est ainsi que nous avons eu l'idée de créer un système expert qui serait capable de conseiller les étudiants dans leur choix afin de faciliter et de relancer l'engagement des étudiants.

Le principe de notre système est le suivant:

- Poser des questions à l'utilisateur concernant ses goûts et ses centres d'intérêt.
- L'enchaînement des questions est conditionné par les réponses données précédemment
- Fournir un diagnostic en fin d'exécution lorsque l'on posé suffisamment de questions.

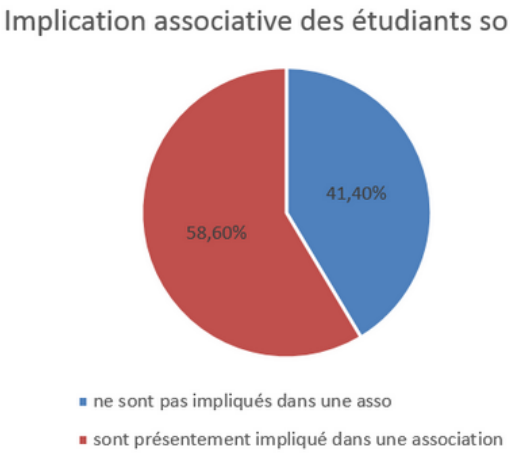
# Domaine d'Expertise

Afin de récolter les informations nécessaire à l'établissement de nos règles et de notre Base de Faits/Connaissances, nous avons décidé de soumettre à sondage les étudiants de l'UTC pour connaître leurs habitudes associatives. Nous nous sommes également basés sur nos connaissances personnelles et sur le guide des associations produit par le BDE chaque semestre.

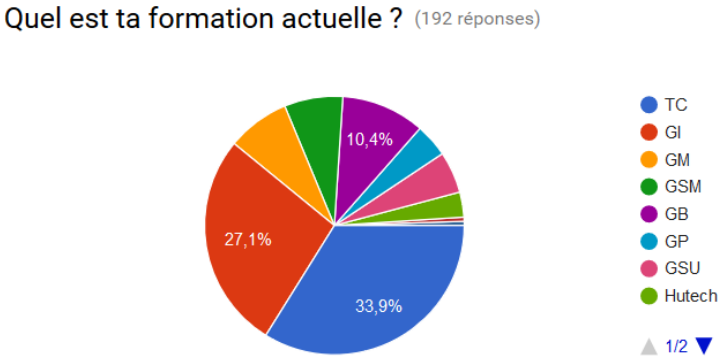
Notre sondage était structuré de la façon suivante:

- Une première partie pour connaître le niveau de formation des sondés.
- Une seconde pour séparer les étudiants impliqués dans les associations de ceux qui ne le sont pas.
- Dans le cas où la personne sondée ne participe à aucune association, on essaye d'en connaître les raisons, sinon on continue le sondage sur une partie contenant des questions sur l'implication de l'étudiant.
- Des questions ciblées pour les associations de musique et d'humanitaire.
- Une question sur les centres d'intérêt.
- Une question sur le poste occupé dans l'association.
- Pourquoi ce choix de poste.
- Le poste demande-t-il des compétences particulières.
- Combien de temps par semaine, prend l'activité au sein de l'association.

Ci-dessous le graphique obtenu concernant le taux de participation des étudiants à une activité associative:




Notre échantillon aurait pu être plus représentatif cependant, certaines branches de l'UTC comptent moins d'étudiants que d'autres, voici ci-après les formations suivies par les sondés:



Voici également la répartition du nombre de réponses par association:


Il s'y dégage nettement que nous avons eu plus de réponses pour des associations comme l'Imaginarium Festival, l'USEC ou Décibels qui sont des associations à gros effectif.

Nous avons également eu des réponses sur des petites associations comme Dada (asso d'arts plastiques), Piano'UT (asso de piano), ou Ocata (asso de musique jazz), ce qui nous a apporté de meilleures connaissances sur les compétences requises pour prendre part aux activités de ces associations.

Le détail des motivations des étudiants nous a permis de définir les "goûts" qui poussent un étudiant vers telle ou telle association. C'est ainsi que nous avons pu catégoriser les associations dans différents groupes aux caractéristiques bien particulières qui conditionneraient l'enchaînement des questions posées.

Au-delà de notre sondage, la plaquette des associations nous a elle aussi été très utile. Grâce à la description plus ou moins détaillée des activités proposées, elle nous a permit de connaître les valeurs des attributs des associations.


Les descriptions données sont aussi celles-utilisées par notre système expert au moment où l'on recommande une association à un étudiant, afin que celui-ci puisse prendre connaissance dans une meilleure mesure de ce que nous lui proposons. Si l'association ne lui convient pas, il est possible de continuer à explorer d'autres centres d'intérêt.

# Choix d'un moteur d'inférence d'ordre 1

Lorsqu'on a commencé à chercher des règles, on est rapidement arrivé à des règles comme:
"Si l'étudiant possède telle caractéristique et que tel asso possède aussi tel caractéristique alors recommander cette asso"

En ordre 0+, il nous faudrait énormément de règles. Alors que l'utilisation d'un ordre 1 nous permet l'utilisation de moins de règles, d'une base de connaissance sur les associations plus maintenable, et plus de flexibilité pour l'ajout d'un système de score. On a donc choisi d'utiliser un moteur d'inférence d'ordre 1.

# La Base de faits *BF*

Tout d'abord, un fait est représenté sous la forme d'un triplet (objet attribut valeur).
Notre base de faits est composée de deux parties :

- une partie statique qui contient les connaissances concernant les assos de l'UTC. Exemple : (objet asso Larsen) (Larsen style rock)
- une partie créée dynamiquement à partir des réponses aux différentes questions représentant les faits relatifs à l'étudiant.

Les associations partageant un même goût possèdent les mêmes attributs afin de faciliter la sélection d'une association en fonction des réponses de l'étudiant.
Ainsi, les associations de musique partagent les attributs : instrument, niveau, façon_pratique, style.

Si un étudiant est amené à donner comme centre d'intérêt la musique, plusieurs questions lui seront posées pour créer des couples attribut - valeur où chaque attribut de chaque couple correspond aux mêmes attributs que les associations de musique.

Ainsi, les associations événementiels partagent les attributs : type et standing.

Les associations vie de campus partagent les attributs : but et type.

Les associations de danse partagent les attributs : style, niveau, pratique.

Les associations sportives partagent l'attribut : sport.

Les associations de mécanique partagent l'attribut : domaine_méca.

Les associations de voyage partagent les attributs : but, lieu.

Les associations humanitaires partagent les attributs : lieu.

Certaines sont limitées qu'au centre d'intérêt car elles sont seules dans leur domaine.

C'est le cas d'Orion pour l'astronomie, VeloUTc pour le cyclisme, UTCiel pour l'aviation, TUC pour citoyenneté.

Une même association peut être caractérisée par des goûts différents donc devra présenter les attributs relatifs à chaque goût : c'est le cas par exemple de Décibels ou SurfUT.

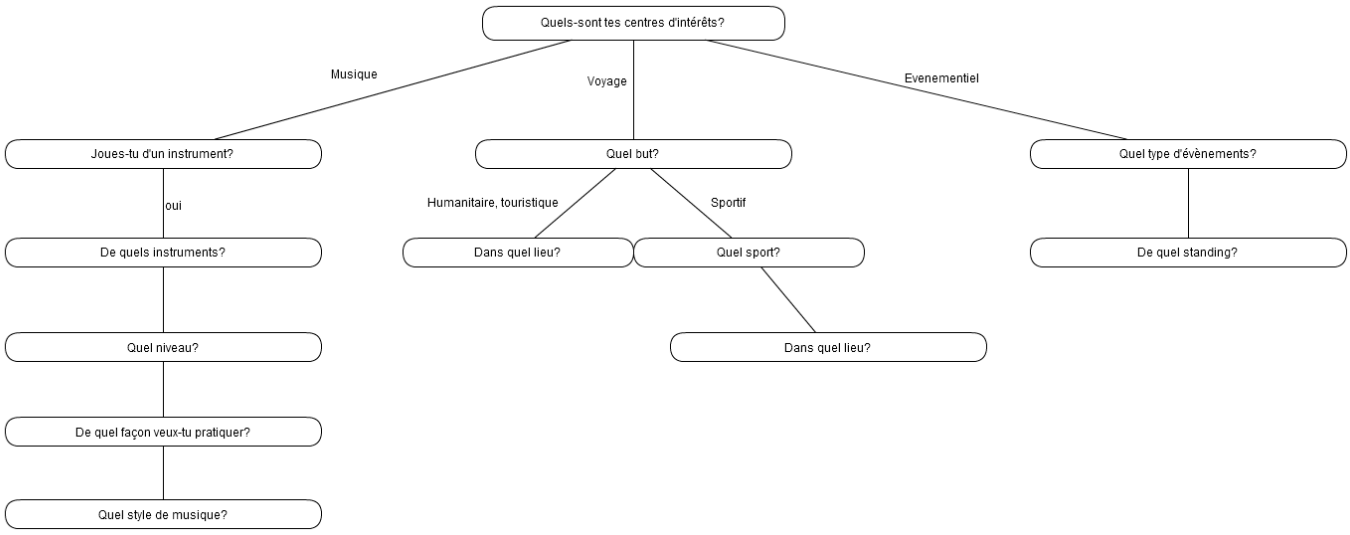
Nous nous sommes limitées à l'implémentation de 31 associations sachant qu'il en existe plus d'une cinquantaine:

**Comet,SDF,etuville,IF,Stravaganza,Ocata,Larsen,Décibels,PianoUT,Choruts, AlaRUtc, Acoustique, UTCIEL,VeloUTC, Orion, Club_bioméca, coincidence, PIC, Integ, Utécia, Picnrock, breakdance, UTRIP, SurfUT, SkiUT, BreakDance, CABARET, TUC, SoleilsenInde, ToitpourleNépal, et ToitPourLeNepal.**

Mais notre système peut supporter l'implémentation de plus d'associations, il suffit de les catégoriser et évaluer les similitudes avec d'autres associations et bien sûr créer de nouvelles règles.

Nous avons comme même essayé de catégoriser quasiment toutes les associations, ce qui n'a pas été une tâche facile. Beaucoup de paramètres doivent être pris en compte : un centre d'intérêt peut être nommé de plusieurs manières par exemple.

Voici l'arbre de nos questions :



# La Base de règles *BR*

**Cadre général :**

La base de règles *BR* est représentée comme une liste de règles allant de $R_1$ à $Q_N$. Une règle se présente de la manière suivante : (nom_règle ($cond_1$ $cond_2$ … $cond_n$) ($action_1$ $action_2$ … $action_n$))

Chaque condition est un quadruplet (opérateur $objet attribut $valeur), où objet et valeur sont des variables, l'attribut étant fixé pour chaque règle. Seulement pour la première condition de toutes les règles, l'objet sera lui aussi fixé à etudiant. L'opérateur peut prendre comme valeur **=** ou **!**.

**!** signifie que la règle ne peut s'appliquer que si le triplet qui suit n'existe pas dans la base de faits.

**=** signifie que la règle ne peut s'appliquer que si un tel fait existe dans la base de faits.

Nous rappelons que toutes les conditions doivent être vérifiées pour que la règle puisse s'appliquer.

Chaque action est basée sur une fonction et se présente de la manière suivante : (nom_fonction paramètre) où le paramètre peut être une variable.

Nous avons implémenté deux types de règles.

Le premier type de règle permet de comparer les attributs de l'étudiant avec ceux des associations. Dès que l'étudiant partage des caractéristiques communes avec une association, l'association voit son poids augmenté au moyen d'une fonction de service ajout_asso qui sera explicitée dans la partie suivante.

Exemple : (R0 ((= etudiant gout (? . gout)) (= objet (? . asso) asso) (= (? . asso) gout (? . gout))) ((ajout_asso (? . asso))))

Cette règle se traduit littéralement par : si l'étudiant a un certain centre d'intérêt et qu'il existe une association partageant le même centre d'intérêt alors l'asso prend plus de poids.

Les variables grâce au moteur d'inférences sont remplacées par les faits existants dans la base de faits.

Il existe actuellement 16 règles de ce type dont le nom commence par RXX.

Le deuxième type de règle, quant à lui, permet de générer des nouvelles questions en fonction des patterns étudiant ajoutés au fur et à mesure dans la base de faits.

Il existe actuellement 12 règles de ce type-ci dont le nom commence par QXX.

Voici un exemple :

(Q0 ((= etudiant façon_pratique (? . façon_pratique)) (= etudiant gout musique)) ((demand style)) )

Cette règle se traduit littéralement par : si l'étudiant a pour centre d'intérêt la musique et il veut jouer dans un groupe ou orchestre ou un autre type de pratique, on lui demande quel style de musique il apprécie.

Ce type de règle permet donc d'élaborer un arbre de questions.

La première règle qui sera automatiquement utilisée est la Q4 permettant de demander le centre d'intérêt de l'étudiant. Elle représente notre point d'entrée du système.

Remarque sur les règles :

Certaines règles du type 1 peuvent paraître identiques comme R1 et R2 :

(R1 ((= etudiant style (? . style)) (= etudiant gout musique) (= objet (? . asso) asso) (= (? . asso) gout musique) (= (? . asso) style (? . style))) ((ajout_asso (? . asso))))

(R2 ((= etudiant style (? . style)) (= etudiant gout danse) (= objet (? . asso) asso) (= (? . asso) style (? . style) (= (? . asso) gout danse))) ((ajout_asso (? . asso))))

Mais en fait, on s'intéresse à deux types d'assos différents : celles de musique et de danse. Ces assos possèdent des attributs communs : style mais ne sont pas associées au même goût.

On impose aux règles des critères (= etudiant gout musique) pour R1 et (= etudiant goût danse) pour R2 afin que par la suite lors de la recherche des règles candidates, il n'y a pas de confusion : si on pose goût musique pour l'étudiant, la règle R2 ne sera pas sélectionnée.

# Le Moteur d'inférence

## Description des principales fonctions

### Les fonctions au cœur du moteur

- (evalVar (list l_vars))

  La fonction **evalVar** évalue une liste donnée en remplaçant les variables qu'il contient par leur valeur. Elle prend en paramètre la liste à évaluer ainsi que la liste des variables et leur valeur. La liste des variables est de la forme: ((nomVar . valeur) ...) par exemple ((asso . Larsen) (gout .musique))

  Tous les niveaux de la liste sont explorés. Une variable dont la valeur n'est pas spécifiée dans la liste des variables est laissée tel quel.

  Ainsi (R1 ((= (? . to) taad) (? . test)) (? . bateau)) '((test . a) (to . b)))

  retourne '(R1 ((= b taad) a) (? . bateau))

- ( fMatching (pattern BF))

La fonction **fMatching** cherche les faits qui correspondent à un pattern donné. Par exemple (Larsen gout musique) correspond au pattern ((? . asso) gout (? . gout)) pour les valeur ((asso . Larsen) (gout . musique)).

fMatching renvoie pour chaque fait correspondant au pattern la combinaison des valeurs correspondantes en vue de l'utilisation d'evalVar. fMatching renvoie donc une liste de listes de variables et leur valeur.

Par exemple (fMatching '((? . asso) gout (? . gout)) *BF*) retourne

(
((gout . musique) (asso . Larsen))
((gout . musique) (asso . PianoUT))
((gout . danse) (asso . PicNRock))
)

On remarque que si aucun fait n'est trouvé, fMatching retourne () (une liste vide, équivalente à nil)

On remarque que si fMatching est appelée sur un pattern sans variable (ie un fait, exemple (fMatching '(etudiant gout musique) *BF*)) il retourne () si ce fait n'est pas dans *BF* et retourne ((nil)) si ce fait est trouvé une fois, ((nil) (nil)) s'il est trouvé 2 fois…

- `(rMatching (l_premisses BF))`

La fonction **rMatching** retourne les combinaisons de variables tel que les prémisses d'une règle soient vraies, de la même façon que fMatching retourne les combinaisons de variables tel que le pattern donné corresponde à un fait dans BF.

- `(regles_candidates (BR BF))`

La fonction **regles_candidates** retourne les règles applicables (dont les prémisses sont vérifiées) . Par exemple :

(
(R0 ((= etudiant gout musique) (= objet Larsen asso) (= Larsen gout musique)) ((ajout_asso Larsen)))
(R0 ((= etudiant gout danse) (= objet PicNRock asso) (= PicNRock gout danse)) ((ajout_asso PicNRock)))
(R1 ((= etudiant style rock) (= etudiant gout musique) (= objet Larsen asso) (= Larsen gout musique) (= Larsen style rock)) ((ajout_asso Larsen)))
)

- `(ajoutFait (fait BR BF))`

La fonction **ajouFait** permet :

- d'une part, d'ajouter un fait à la base de faits

- d’autre part de sélectionner et d’appliquer de nouvelles règles, non applicables auparavant sans l’ajout de ce nouveau fait à la BF.
Cette fonction fait appel à regles_candidates afin de récupérer des règles applicables.
Une fois ces règles récupérées, au moyen d’un loop, elles vont être appliquées grâce à la fonction apply_regle. Toutefois, la totalité de ces règles peut ne pas être appliquée si on a réussi à trouver l’association qui correspondait à l’étudiant.

Ensuite, les règles du premier type seront toujours appliquées avant celles du deuxième type.

Les règles qui seront sélectionnées correspondent d’abord aux règles du premier type : d’une règle, en particulier, en relation avec le pattern étudiant récemment ajouté (elle sera appliquée autant de fois qu’il y a d’associations partageant le même couple (attribut-valeur)).

Si dans la liste des règles candidates, seulement des règles du type 1 sont appliquées, alors cela signifie que nous avons parcouru tout un chemin et sommes arrivés à une feuille de notre arbre de questions. Ces règles sont appliquées et on fait appel ensuite à une fonction qui va évaluer quelle est l’asso qui a le meilleure score (fonction recom).

Finalement, on se trouve dans une telle situation si la dernière règle à appliquer est de type 1. (c’est cette condition qui est évaluée dans le loop).

Si maintenant dans la liste des règles candidates est présentées à la fois des règles du type 1 et 2, ajout_fait va être appelée récursivement suite à l’application de la règle du type 2.

Étant donnée la manière dont les règles sont implémentées, une seule règle de type 2 ne peut être appliquée dans le loop.

Si l’étudiant n’est pas satisfait de l’asso proposée par le recom, trouve reste à nil. On remonte dans les appels imbriquées et on explore les règles non appliquées ou les réponses multiples aux questions.

- `(apply-regle (regle BR BF ))`

  La fonction **apply_regle** permet :
  - d’une part, l’application de la règle envoyée en paramètre.
  - d’autre part, la mise à jour de la base de règles.

  Voici le fonctionnement :

  La conclusion est analysée et on procède à un funcall du car de cette conclusion.

  Les fonctions à appliquer via le funcall sont construites de la même manière avec trois paramètres : le premier paramètre est précisé dans la conclusion de la règle correspondant au cadr et les deux autres sont la BR mise à jour et la base de faits.

  Comme indiqué ci-dessus, cette fonction met à jour la base de règles en retirant la règle utilisée : le résultat sera stockée dans une copie de la base de règles. On ne touche jamais à la base de règles générale. On ne travaille que sur des copies.

En supprimant la règle utilisée, on permet au moteur d'inférences de réduire l'espace de recherche des règles candidates.

De même suite à l'application de la règle, la base de faits se trouve modifiée et est renvoyée suite au funcall.

La fonction apply_regle renvoie à la fonction appelante les modifications apportées sur la base de règles et la base de faits.

Suite à plusieurs appels récursifs de cette fonction, plusieurs copies de la base de règles et de la base de faits seront réalisées mais n'existeront que temporairement. Le fait de travailler sur des copies permet à la fois de remonter facilement dans l'arbre des questions et de redémarrer le système seulement avec la fonction start.

## Les fonctions de service

- `(demand (theme BR BF))`

Cette fonction de service possède une position stratégique dans notre programme car c'est elle qui pose les questions à l'utilisateur et déclenche AjoutFait (la fonction chargée de l'ajout des patterns correspondants à l'étudiant) lorsque la réponse de l'utilisateur est récupérée.

Selon le thème avec lequel la fonction "demand" est appelée, les conséquences sont différentes. Voici l'algorithme pour deux thèmes (les autres thèmes sont globalement construits de la même manière).

```
Fonction Demand (theme BR BF)
    Si theme==gout alors
        Faire //loop global qui ne prend fin que si l'on trouve ou
    force l'arrêt
            Si trouve != TRUE alors
                Break;
            Afficher ("Quels sont tes centres d'intérêt ?")
            Rep := lire clavier
            Si rep=='fin' alors
                Break;
            AjoutFait ((étudiant gout rep) BR BF)
        Tant que rep!='fin'
    Si theme == humanitaire alors
        Afficher ("Dans quel lieu ?")
        Rep := lire clavier
        AjoutFait ((étudiant lieu rep) BR BF)
    Si theme == …...
Fin Demand
```

La fonction est appelée dans les conclusions des règles commençant par "Qx", autrement dit les règles qui posent des questions à l'utilisateur.

- `(ajout_asso (asso BR BF))`

L'objectif de cette fonction est d"incrémenter" le score d'une association dans notre base de faits. Dans la pratique, il serait plus judicieux d'employer les termes "donner plus de poids" car ajout_asso fonctionne de la façon suivante:

En cas de réponse de l'utilisateur qui coïnciderait avec la caractéristique d'une association présente dans notre base de faits on fait appel à cette fonction.

En voici l'algorithme:

```
Fonction ajout_asso ( asso BR BF)
    Si (recom asso) n'appartient pas déjà à BF
        => on push dans BF l'élément (recom asso)
    Puis on push dans BF l'élément (score asso)
Fin ajout_asso
```

L'élément "(recom asso)" est un pattern que l'on récupérera dans la base de faits pour savoir si une association a un score supérieur à zéro. Le score est symbolisé par le nombre d'occurrence de "(score asso)" dans la base de faits.

- `(maximum (BF))`

La fonction **maximum** a pour objectif de renvoyer un couple (Asso . Score) avec l'association qui possède le meilleur score (donc celle pour laquelle on observe le plus d'occurrence de (score asso) dans la base de faits. Pour cela, on fait appel à la fonction fMatching deux fois. Cette fonction nous permet de reconnaître les patterns "(recom asso)" et "(score asso)" et renvoi une liste avec les patterns si l'on a appelé la fonction avec une variable inconnue; sinon elle renvoie une liste contenant des (NIL) autant de fois que le pattern est rencontré.

Voici l'algorithme:

```
Fonction maximum (BF)
    Lst_asso := fMatching ((recom ?) BF) // appel avec variable
    Maxi := (x . 0)
    Pour chaque asso récupérée dans lst_asso faire
        Lst_comptage := fMatching ((score asso) BF) //appel sans
    var
        Score := (asso . Length (lst_comptage))
        Si (cdr score) > (cdr maxi) alors
            Maxi := score
```

```
    Fin pour
Fin Maximum
```

- (recom (BF))

Cette fonction est appelée lorsqu'il n'y a plus de règles candidates qui permettent de poser une autre question à l'utilisateur. Le principe est simple, elle récupère l'association de score maximum au moyen de la fonction Maximum détaillée précédemment puis demande à l'utilisateur si le choix lui convient. Si besoin elle appelle la fonction descriptif qui fournit un descriptif de l'association en question. Si l'utilisateur répond "oui" le programme s'arrête. Sinon, il remonte dans les appels récursifs et dans les loops. Voici l'algorithme:

```
Fonction recom (BF)
    Asso := Maximum (BF)
    Afficher (" On te recommande cette association : $asso Cela te va?")
    Rep := lire clavier
        Si Rep=='?' alors
            Descriptif (asso)
            Afficher ("Veux tu t'y investir?")
            Rep := lire clavier
        Si Rep == 'oui'
            Trouve := TRUE
Fin Recom
```

- (start)

La seule vocation de cette fonction est de lancer l'application de la première règle candidate afin de démarrer le programme. Sa structure est du type:

```
Fonction Start ()
    Apply_regle ( ( Q4 ) *BR* *BF*)
Fin Start
```

où Q4 est la première question qui demande les centres d'intérêt de l'utilisateur lorsque la base de fait ne contient aucun pattern étudiant.

- `(descriptif (asso))`

La fonction **descriptif_asso** donne un descriptif de l'association envoyée en paramètre. L'association ainsi envoyée est comparée aux différentes associations dans un cond. Les descriptions sont tirées de la plaquette des assos. Seules trois descriptions existent. Il est très facile d'en implémenter des nouvelles.

- `(reset)`

La fonction **reset** permet de redémarrer le système et de mettre trouve à nil, si une asso avait été accepté.

# Exemple détaillé d'une application asso goût musique

Nous allons maintenant vous présenter un exemple dont la procédure sera expliquée pas à pas.

Lançons d'abord la fonction start : (start)

- BR_temp vaut *BR* exclus de la règle Q4
- La règle Q4 est appliquée au moyen du funcall de la manière suivante (demand goût BR_temp BF)

La deuxième condition de la fonction demand est vérifée : (eq theme 'gout) : on rentre alors dans un loop

- trouve valant nil, la question « Quels sont tes centres d'intérêt ? » est affichée :
- admettons que l'utilisateur rentre musique
- ajoutFait est alors appelée avec comme paramètre$_1$ = (etudiant goût musique)

(ajoutFait (etudiant goût musique) BF BR)

- trouve vaut toujours nil et ce fait n'existe pas encore dans BF, nous rentrons dans le deuxième if imbriqué :
- la fonction regles_candidates est appelée en ajoutant par un append ce fait à BF

---

(regles_candidates BR_{Q4}((etudiant goût musique) (objet asso Comet) … ) )

- 1ère itération du dolist :
  - regle = (R0 ((= etudiant gout (? . gout)) (= objet (? . asso) asso) (= (? . asso) gout (? . gout))) ((ajout_asso (? . asso))))
  - 2ème dolist, à la1ère itération,
    rMatching est appelée de la manière suivante :

---

```
(rMatching '((= etudiant gout (? . gout)) (= objet (? . asso)
asso) (= (? . asso) gout (? . goût))) BF)
```

- appel de fMatching sur (etudiant goût (? . goût)) BF) :
  ➔ 1ère itération du dolist : (il n'y en aura qu'une seule ici)
  fait = (etudiant goût musique)

1ère itération du do :
`l_fAtoms = (etudiant goût musique) & l_pAtoms = (etudiant goût (? . goût))`
=> il ne se passe rien
2ère itération du do :
`l_fAtoms = (goût musique) & l_pAtoms = (goût (? . goût))`
=> il ne se passe rien
3ère itération du do :
`l_fAtoms = (musique) & l_pAtoms = ((? . goût))`
`(if (and (listp (car l_pAtoms)) (equal (caar l_pAtoms) '?) (car l_fAtoms))` est vérifié
`l_vars = ((goût . Musique))`
On sort du do
`l_matchings = (((goût . musique)))`

Au final, un seul goût sera trouvé : fMatching renvoie (((goût . Musique)))

- Retour à rMatching : le codeur de l_premisses étant non nul
  On rentre dans les deux autres dolist imbriqués :
  La variable fMatching vaut ((goût . Musique))
  rMatching sera rappelée avec (= objet (? . asso) asso) (= (? . asso) gout musique)) où ( ? . goût) a été substitué grâce à evalVar par musique

→ On récupère ainsi avec fMatching la liste de toutes le assos : `(((asso . Comet)) (asso . SDF) ….)`

→ Chaque asso va être testée pour savoir si elle partage ce centre d'intérêt au moyen du dolist

→ à la première itération du `(dolist (fMatching l_fMatchings l_matchings)` , `fMatching = ((asso . Comet))`

→ rMatching est rappelée avec cette fois ( = comet gout musique)) où (? . asso) a été substitué par comet. Un tel fait n'existant pas dans la base de faits, fMatching renvoie nil. On procède ainsi pour chaque association.
Par contre, pour Stravaganza par exemple : le fait existe l_fMatchings à la suite de l'appel de rMatching aura renvoyé (nil)
l_matchings = (((asso . Stravaganza)))

→ A la fin, `l_matchings = (((asso . Stravaganza)) ((asso . Décibels)) ((asso . Larsen)) ((asso . PianoUT)) ((asso . Choruts)) ((asso . Ocata)) ((asso . Acoustic)) ((asso . AlaRUtc)))`

- On sort ainsi de l'appel récursif, on retourne au premier appel de rMatching par regles_candidates, on se situe alors au niveau du deuxième dolist :
  `(dolist (matching '(((asso . Stravaganza)) ((asso . Décibels)) ((asso . Larsen)) ((asso . PianoUT)) ((asso . Choruts)) ((asso . Ocata)) ((asso . Acoustic)) ((asso . AlaRUtc)))))`

→ 1ère itération :
matching = ((asso . Stravaganza))

Création du couple `((goût . Musique) (asso . Stavaganza))` pushé dans la variable l_matchings

→ n$^{ième}$ itération :

l_matchings en sortie vaut `(((goût . Musique) (asso . AlaRUtc)) … ((goût . Musique) (asso . Stavaganza)))`

---

Retour à regles_candidates :

Création des règles en substituant les variables par les valeurs trouvées par les fonctions précédentes :

→ 1$^{ère}$ itération

```
matching = ((goût . Musique) (asso . AlaRUtc))
(push (evalVar '(R0 ((= etudiant gout (? . gout))
(= objet (? . asso) asso) (= (? . asso) gout (? . gout)))
((ajout_asso (? . asso)))) matching)) l_regles_c)
l_regles_c = ((R0
 ((= ETUDIANT GOUT MUSIQUE) (= OBJET AlaRUtc ASSO)
  (=  AlaRUtc  GOUT MUSIQUE))
 ((AJOUT_ASSO AlaRUtc))))
```

→ n$^{ème}$ itération

```
l_regles_c = ((R0
 ((= ETUDIANT GOUT MUSIQUE) (= OBJET STRAVAGANZA ASSO)
  (= STRAVAGANZA GOUT MUSIQUE))
 ((AJOUT_ASSO STRAVAGANZA))) … (R0
 ((= ETUDIANT GOUT MUSIQUE) (= OBJET AlaRUtc ASSO)
  (=  AlaRUtc  GOUT MUSIQUE))
 ((AJOUT_ASSO AlaRUtc)))))
```

Les autres règles sont explorées de la même manière : on retiendra finalement en plus des règles précédentes la règle Q6 :
(Q6 ((= etudiant gout musique)) ((demand musique)))

La liste ainsi renvoyée par la fonction regles_candidates à ajoutFait est la suivante:

```
((R0
 ((= ETUDIANT GOUT MUSIQUE) (= OBJET AlaRUtc ASSO)
  (=  AlaRUtc  GOUT MUSIQUE))
 ((AJOUT_ASSO AlaRUtc))) …
 (R0
 ((= ETUDIANT GOUT MUSIQUE) (= OBJET STRAVAGANZA ASSO)
  (= STRAVAGANZA GOUT MUSIQUE))
 ((AJOUT_ASSO STRAVAGANZA))
(Q6 ((= etudiant gout musique)) ((demand musique))))
```

Retour à (ajoutFait (etudiant goût musique) BF BR)

🕐 Ces règles vont être tour à tour appliquées au moyen de la fonction apply_regles : l'application de la première règle va permettre d'ajouter deux nouveaux patterns à la base de faits :

(recom AlaRUtc) & (score AlaRUtc) qui vont nous permettre par la suite de comptabiliser les scores des assos et de déterminer la meilleure asso faite pour l'étudiant.

- Finalement pour l'ensemble des règles de type R0, deux patterns ayant pour attributs recom et score vont être ajoutés pour chaque association sélectionnée par la règle.
- Q6 sera appelée en dernier lieu : provoquant la génération d'une nouvelle question, en l'occurence ici : « Joues-tu d'instrument ? »
- L'utilisateur répond non à cette question : ajoutFait est de nouveau appelé avec cette fois (étudiant instrument non)

---

Une règle est trouvée dans ce cas : (R3 ((= etudiant instrument non) (= objet Décibels asso) Décibels instrument non)) ((ajout_asso Décibels))).
Etant donné, qu'il existe pas de règles de type 2, cela signifie que nous sommes arrivés à une feuille de notre arbre à questions.
La condition suivante est vérifiée : (and regle (null lst_regles) (eq (car (caaddr regle)) 'ajout_asso)) alors la fonction recom est appelée.

Recherche du maximun :

Nous n'allons pas décrire ici le déroulement de fMatching sur (recom ( ?. d)) mais lst_asso prendra la liste de toutes les assos qui ont un pattern recom.
On obtient ainsi lst_asso `(((d . AlaRUtc)) … ((d . Stravaganza)))` maxi est initialisé à (x . 0)

→ 1ère itération du dolist `asso = ((d . AlaRUtc))`

```
(setq lst_comptage (fMatching `(score ,(cdar asso)) BF))
(setq lst_comptage (fMatching `(score AlaRUtc) BF))
      lst_comptage = (nil)
(setq score (cons (cdar asso) (length lst_comptage)))
     score = (AlaRUtc . 1)
```

1>0 donc maxi = (AlaRUtc . 1)

→ n-1ième itération du dolist `asso = ((d . Décibels)`

```
(setq lst_comptage (fMatching `(score Décibels) BF))
      lst_comptage = (nil nil)
(setq score (cons (cdar asso) (length lst_comptage)))
     score = ( Décibels . 2)
```

2>1 donc maxi = (Décibels . 2)

Finalement , l'association Décibels lui est recommandée puisqu'elle possède le meilleure score. De plus, c'est la seule asso dans notre BF qui lie le centre d'intérêt musique et le fait defaire des activités en rapport avec la musique autre que jouer un instrument.

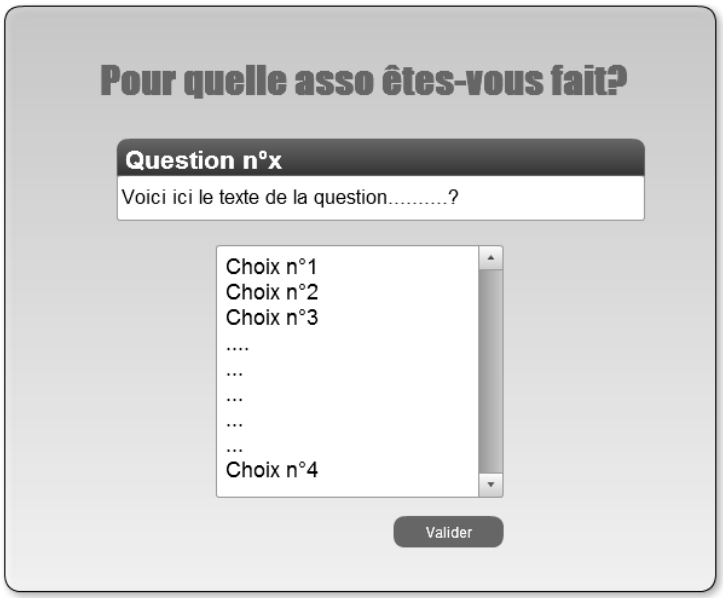
Admettons que l'utilisateur ne souhaite pas s'investir dans cet asso, on va remonter dans l'arbre des questions et retourner sur la question des centres d'intérêt car présent dans un loop. On repartira ainsi avec la base de règles et la base de faits initiales.

# Ouverture

## Interface utilisateur imaginée

Pour aller avec notre programme, nous avons envisagé une interface utilisateur où l'utilisateur pourrait plutôt choisir parmi une liste prédéfinie de réponses afin de garder une cohérence et d'éviter les erreurs que pourraient entraîner les fautes de frappes dans la suite de l'exécution de notre programme.

Cette interface prendrait la forme d'un questionnaire à choix multiple dynamique, dans lequel les questions apparaissent au fur et à mesure des réponses sélectionnées par l'utilisateur.




Selon les questions et le type de réponse attendu, la boîte de sélection de choix peut changer de forme (simple listbox, checkbox ou bien radio button). La réponse est interprétée par le programme qui change les box questions et réponses en accord avec la règle appliquée.

## Améliorations envisagées

Il aurait été intéressant d'ajouter les postes disponibles dans les asso et d'ainsi suggérer un poste à un étudiant tout en prenant compte de ses motivations à faire une asso et du degré d'implication souhaité. En lui suggérant un poste, cela peut être un moyen d'allier plusieurs centres d'intérêt.

On a aussi fait un autre système pour palier aux problèmes rencontrés lors de la création du premier. Celui-ci utilise des faits plus longs pour éviter des confusions entre les faits (par exemple : (etudiant niveau moyen) niveau en quoi ? musique ? danse ?...)

Ce système utilise la flexibilité du moteur d'ordre 1 pour utiliser un système de score et de pondération des goûts de l'étudiant et des caractéristiques des associations pour proposer des résultats plus précis.
Malheureusement, cela rend la création de la base de connaissance très longue et complexe et nous avons pas eu le temps de la compléter assez.

# Conclusion

La réalisation de ce système expert fut une expérience très enrichissante non seulement en ce qui concerne l'amélioration de nos compétences en LISP et en représentation des connaissances; mais cela nous a aussi permis de redécouvrir la vie associative de notre école et mieux la connaître et comprendre les motivations des uns et des autres. Notre système pourrait être exportable dans le cadre d'autres écoles. Il suffirait de changer la base de faits en la remplissant avec d'autres associations.

Nous avons pensé à proposer notre système au BDE pour qu'il puisse l'utiliser lors de la JDA (journée des associations) qui a lieu une fois à chaque début de semestre pour inciter les étudiants à s'engager. Ce système expert pourrait être une bonne animation qui permettrait peut-être à certains étudiants de se découvrir une vocation dans le monde associatif.