Ecriture en Prolog d'un démonstrateur basé sur l'algorithme des tableaux pour la logique de description $\mathcal{ALC}$

Quelques rappels préliminaires

I - Logique de description $\mathcal{ALC}$

1) Alphabet

- Concepts atomiques : A, B, C, …
- Rôles atomiques : R, S, U, V , …
- Symboles : $\{\sqcup, \sqcap, \exists, \forall, \neg, \top, \bot, .\}$
- Instances de concepts : a, b, …

2) Base de connaissances

- TBox - Axiomes terminologiques :
  - Définitions : $C \equiv D$
  - Subsomptions : $C \sqsubseteq D$ (se lit C est subsumé par D)
- ABox - Assertions :

  Assertions de concepts : a : C

  Assertions de rôles : $\langle a, b \rangle$: R

3) Grammaire

concept ::= $\langle$concept atomique$\rangle$
| $\top$
| $\bot$
| $\neg$ $\langle$concept$\rangle$
| $\langle$concept$\rangle$ $\sqcup$ $\langle$concept$\rangle$
| $\langle$concept$\rangle$ $\sqcap$ $\langle$concept$\rangle$
| $\exists$ $\langle$rôle$\rangle$. $\langle$concept$\rangle$
| $\forall$ $\langle$rôle$\rangle$. $\langle$concept$\rangle$

4) Sémantique de $\exists$ et $\forall$

Etant donnée une interprétation $I = (\Delta^I, .^I)$, on a :

$(\exists R.C)^I = \{ x \in \Delta^I \mid \exists y, (x, y) \in R^I \wedge y \in C^I \}$

Exemple 1 - Avoir au moins un enfant qui est humain : ∃a_enfant.Humain.

Exemple 2 - Ne pas avoir d'enfants qui sont humains (on peut toutefois en avoir qui ne soient pas humains) : ¬∃a_enfant.Humain.

$(\forall R.C)^I = \{x \in \Delta^I \mid \forall y, (x, y) \in R^I \rightarrow y \in C^I\}$.

Exemple 1 - Avoir uniquement des enfants humains (mais ne pas nécessairement en avoir) :
∀a_enfant.Humain

Exemple 2 - Ne avoir pas uniquement des enfants humains (ie avoir au moins un enfant qui ne soit pas humain) : ¬∀a_enfant.Humain

II - Algorithme des tableaux pour la logique de description $\mathcal{ALC}$

1) Règles de réécriture

Etant donné un ensemble de formules F écrites en $\mathcal{ALC}$ sous forme normale négative,

	Si F contient	***Ajouter a F***
(Règle-⊓)	a : C ⊓ D	a : C et a : D
(Règle-⊔)	a : C ⊔ D	a : C ou a : D (deux branches)
(Règle-∀)	a : ∀R.C et ⟨a, b⟩ : R	b : C
(Règle-∃)	a : ∃R.C	⟨a, b⟩ : R et b : C, où b est un nouvel objet

2) Principe de l'algorithme

L'algorithme consiste à developper l'arbre de démonstration en appliquant récursivement les règles précédentes.

Une branche est dite *fermée*, si elle contient deux assertions a : A et a : ¬A.
Une branche est dite *complète*, si plus aucune règle ne s'applique.
L'algorithme s'arrête, si toutes les branches sont fermées ou si une branche est complète.
L'ensemble de formules F est *insatisfiable* si et seulement si toutes les branches de l'arbre sont fermées.

III - Le prédicat cut (noté '!') en Prolog

Le prédicat ! est le coupe-choix. Normalement, Prolog doit fournir toutes les instantiations possibles de toutes les variables apparaissant dans le but qu'on lui demande de vérifier. Illustrons ce processus :

Soit la base de faits et de règles suivante :
homme(nicolas).

homme(fabien).
femme(magali).
femme(caroline).
enfant(chloe).
enfant(justine).
On cherche à former une famille fictive pour jouer dans un film avec un homme pour le père, une femme pour la mère et un enfant tenant le rôle de l'enfant du couple.
On définit :
famille(X,Y,Z) :- homme(X), femme(Y), enfant(Z).
On soumet à Prolog le but : famille(X,Y,Z).

famille(X,Y,Z)

homme(X)	femme(Y)	enfant(Z)
X=nicolas.	Y=magali	Z=chloe

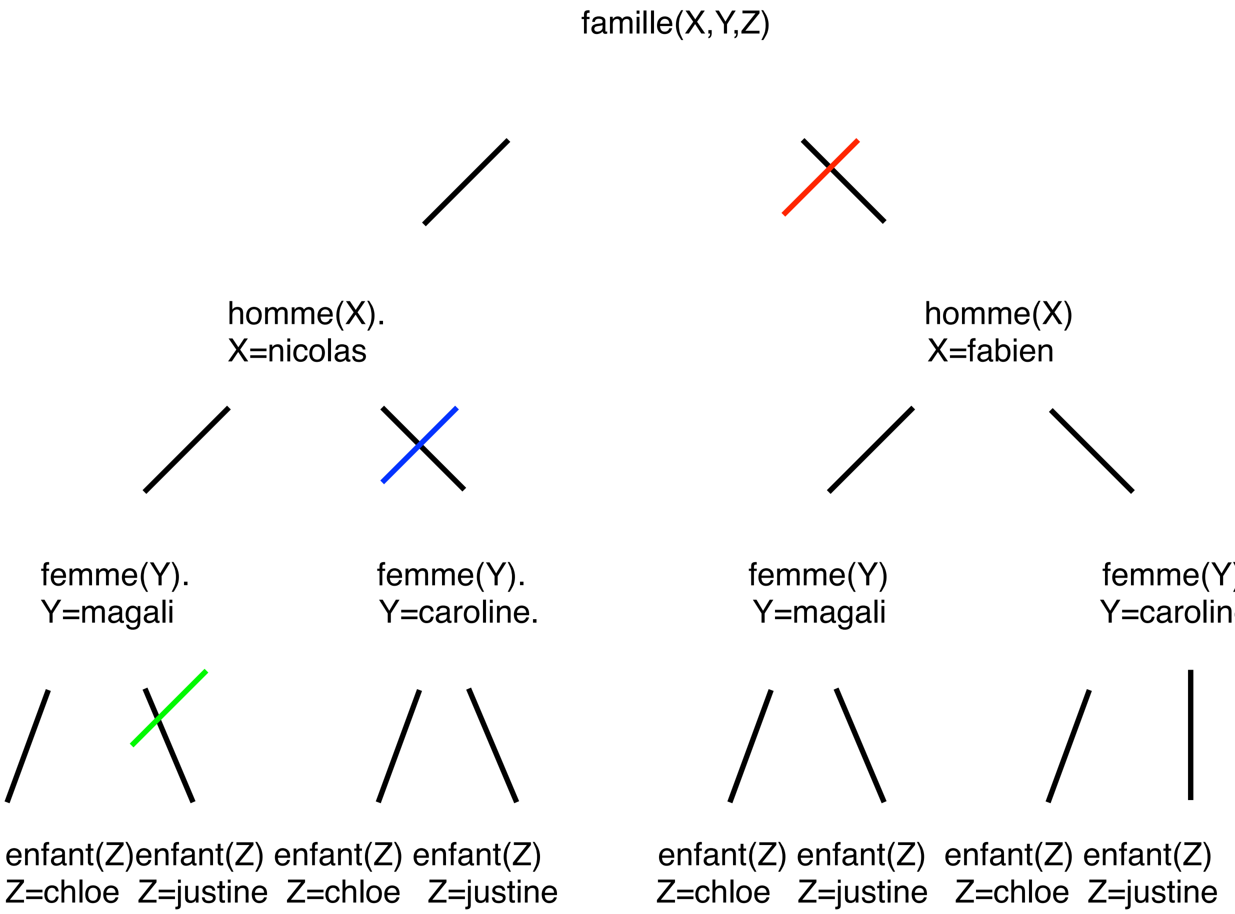
………..

<u>1ère solution</u> : X=nicolas,Y=magali,Z=chloe
Après avoir trouvé cette solution, Prolog va revenir sur le dernier choix réalisé, c'est-à-dire le choix de la clause enfant(chloe).
et il va faire le choix de la clause suivante dans l'ordre des clauses du prédicat enfant : enfant(justine).

<u>2ème solution</u> : X=nicolas,Y=magali,Z=justine
Ayant épuisé tous les choix possibles pour la variable Z du dernier prédicat, enfant, Prolog va revenir sur le choix antérieur, celui de la variable Y et il va instancier Y grâce à la seconde clause du prédicat femme : femme(caroline) et va essayer à nouveau d'instancier la variable Z à l'aide des 2 clauses du prédicat enfant pour générer 2 solutions supplémentaires et ainsi de suite. Il reviendra ensuite sur l'instantiation de la variable X. Ces retours sur les choix sont appelés des backtracks.

Par défaut, Prolog backtracke systématiquement sur le choix des valeurs affectées aux variables libres, c'est-à-dire à instancier.

Le prédicat prédéfini cut, qui s'écrit '!', permet de « couper les choix » de la façon suivante. Lorsque l'on met un cut dans la partie droite d'une clause Prolog parmi les autres prédicats de cette partie, on indique à Prolog de ne pas backtracker sur les instantiations possibles des prédicats qui se trouvent avant le cut dans la partie droite de la clause, ainsi que de ne pas backtracker sur les instantiations de la tête de la clause.
Si l'on schématise les instantiations dans l'exemple précédent par l'arbre suivant :



Le parcours de cet arbre en profondeur d'abord schématise l'enchaînement des backtracks effectués par Prolog sur les instantiations des variables X, Y et Z.

Si l'on écrit la clause :
famille(X,Y,Z) :- homme(X), !, femme(Y), enfant(Z).
on va supprimer le backtrack sur les instantiations possibles du prédicat homme, donc on ne va pas explorer la branche droite partant de famille(X,Y,Z), c'est-à-dire que l'on n'examine pas l'instantiation de la variable X avec fabien, ni a fortiori les instantiations possibles à partir de là pour Y et Z. On a donc coupé la branche où figure le trait rouge.
Les solutions trouvées sont, dans l'ordre :
(nicolas,magali,chloe), (nicolas,magali,justine),(nicolas,caroline,chloe),
(nicolas,caroline,justine)

Si l'on écrit la clause :
famille(X,Y,Z) :- homme(X), femme(Y), !, enfant(Z).
on coupe alors les choix possibles sur les variables X et Y des prédicats homme et femme qui se trouvent avant le cut.
Dans ce cas, non seulement la « branche rouge » est coupée (non explorée), mais aussi la branche bleue.
Les solutions trouvées sont, dans l'ordre :
(nicolas,magali,chloe), (nicolas,magali,justine)

Enfin, si l'on écrit la clause :
famille(X,Y,Z) :- homme(X), femme(Y), enfant(Z), !.

on coupe aussi les choix possibles pour la variable Z du prédicat enfant situé avant le cut. Dans ce cas, les branches rouge, bleue et verte sont coupées.
La seule solution trouvée est :
(nicolas,magali,chloe)

Mettre un cut en dernier prédicat de la partie droite d'une clause revient à demander à Prolog une seule solution pour l'instantiation du but exprimé par la tête de la clause en question.

IV - Mise sous forme normale négative d'une expression conceptuelle dans la logique de description $\mathcal{ALC}$ en Prolog

Il s'agit de transformer l'expression d'un concept de telle sorte que, s'il y a des négations dans cette expression, elles portent directement sur un concept atomique.

On va donc examiner la façon de concevoir la définition d'un prédicat qui met sous forme normale négative un concept ou la négation d'un concept, ce concept étant lui-même défini selon la grammaire des concepts en $\mathcal{ALC}$, vue dans les rappels.

Voici les règles de mise sous forme normale négative (nnf : negative normal form) d'un concept. On envisage tous les cas pour l'expression possible du concept (on constatera le caractère récursif de la définition de la fonction nnf) :

nnf(not(and(C1,C2))) = or(nnf(not(C1)),nnf(not(C2)))	(1)
nnf(not(or(C1,C2))) = and(nnf(not(C1)),nnf(not(C2)))	(2)
nnf(not(all(R,C)))= some(R,nnf(not(C)))	(3)
nnf(not(some(R,C)))=all(R,nnf(not(C)))	(4)
nnf(not(not(C))=C	(5)
nnf(not(C))=not(C), si C est un concept atomique	(6)
nnf(and(C1,C2))=and(nnf(C1),nnf(C2))	(7)
nnf(or(C1,C2))=or(nnf(C1),nnf(C2))	(8)
nnf(some(R,C))=some(R,nnf(C))	(9)
nnf(all(R,C))=all(R,nnf(C))	(10)
nnf(C)=C si C est un concept atomique	(11)

Exemple : Soit le concept C dont la définition est la suivante :
C = personne ⊓ (¬(∃a_ecrit.livre) ⊓ (∃a_edite.livre))

Avec une notation préfixée :
C = and(personne,and(not(some(a_ecrit,livre)),some(a_edite,livre))

Mettons sous forme normale négative la négation de C :
nnf(not(C)) =
nnf(not(and(personne,and(not(some(a_ecrit,livre)),some(a_edite,livre))))) =
(par la règle 1)
or(nnf(not(personne)),nnf(not(and(not(some(a_ecrit,livre)),some(a_edite,livre)))))=
(par les règles 6 et 1)

or(not(personne),or(nnf(not(not(some(a_ecrit,livre)))),nnf(not(some(a_edite,livre)))))=
(par la règle 5)
or(not(personne),or(nnf(some(a_ecrit,livre)),nnf(not(some(a_edite,livre)))))=
(par les règles 9 et 4)
or(not(personne),or(some(a_ecrit,nnf(livre)),all(a_edite,nnf(not(livre)))))=
(par les règles 11 et 6)
or(not(personne),or(some(a_ecrit,livre),all(a_edite,not(livre))))

et voici la définition du prédicat nnf en Prolog :

```
nnf(not(and(C1,C2)),or(NC1,NC2)):- nnf(not(C1),NC1),
                                    nnf(not(C2),NC2),!.
nnf(not(or(C1,C2)),and(NC1,NC2)):- nnf(not(C1),NC1),
                                    nnf(not(C2),NC2),!.
nnf(not(all(R,C)),some(R,NC)):- nnf(not(C),NC),!.
nnf(not(some(R,C)),all(R,NC)):- nnf(not(C),NC),!.
nnf(not(not(X)),X):-!.
nnf(not(X),not(X)):-!.
nnf(and(C1,C2),and(NC1,NC2)):- nnf(C1,NC1),nnf(C2,NC2),!.
nnf(or(C1,C2),or(NC1,NC2)):- nnf(C1,NC1), nnf(C2,NC2),!.
nnf(some(R,C),some(R,NC)):- nnf(C,NC),!.
nnf(all(R,C),all(R,NC)) :- nnf(C,NC),!.
nnf(X,X).
```

Il est calqué sur la définition de la fonction nnf précédente.

Pourquoi avoir mis systématiquement un prédicat de coupure de choix, !, en fin de clause dans toutes les clauses sauf la dernière ?

Chacune des clauses correspond à un cas pour l'expression du concept dont on cherche la forme normale négative. Etant donné donc un concept, pour utiliser une des formules de « calcul » exprimées par les clauses, il faut que l'expression du concept s'unifie avec le premier argument du prédicat nnf de cette clause. Si l'on a trouvé une telle clause, il n'est pas nécessaire d'aller trouver une autre solution, donc une autre clause, pour calculer encore la nnf de notre concept, puisque cette dernière est unique. On peut ainsi mettre un cut en dernière position dans les parties droites des clauses.

Il faut toutefois examiner ce processus plus finement. Si l'on regarde par exemple la première clause et les 4 suivantes, on s'aperçoit que si notre concept s'unifie avec l'expression `not(inter(C1,C2))`, il ne risquera pas de s'unifier avec le premier argument du prédicat nnf dans les clauses suivantes. L'intérêt de mettre un cut est simplement d'éviter à Prolog d'aller examiner ces clauses et de tenter une unification vouée à l'échec.

Par contre, considérons la clause :
`nnf(not(X),not(X)) :- !.`

Un concept de la forme `not(and(C1,C2))` ou `not(or(C1,C2))` ou `not(some(R,C))`, par exemple, peut s'unifier avec le premier argument du prédicat nnf de cette clause et si Prolog tentait d'utiliser cette clause, il déduirait que la nnf de ce concept est lui-même, ce qui est faux !

C’est pour éviter cela que l’on a mis un cut à la fin des clauses précédentes. De ce fait, on est sûr que si Prolog tente d’utiliser cette clause, c’est que le concept en question commence par une négation, mais qu’après on ne trouve pas dans son expression ni `and`, ni `or`, ni `all`, ni `some`, ni `non`, cas qui auraient été traités dans les clauses précédentes, c’est donc un concept atomique et alors la nnf de sa négation est sa négation elle-même.

Structure du programme Prolog implémentant le démonstrateur

Partie 1 - Etape préliminaire de vérification et de mise en forme de la Tbox et de la Abox

Dans cette partie, on va vérifier la correction syntaxique et sémantique d'une Tbox et d'une Abox et les mettre en forme, afin de nous en servir comme bases de tests pour le démonstrateur à réaliser, lorsque nous lui fournirons une proposition à démontrer.

La Tbox et la Abox données en entrées seront représentées en utilisant les prédicats Prolog suivants :

- `equiv(ConceptAtom,ConceptGen)`
qui précise les équivalences entre concepts, le premier paramètre étant un concept atomique et le second un concept quelconque (au sens de la grammaire des concepts, voir plus loin) auquel il est équivalent.

- `inst(Instance,ConceptGen)`
qui précise les instantiations de concepts.

- `instR(Instance1, Instance2, role)`
qui précise instantiations de rôles.

- `cnamea(ConceptAtom)`
qui précise les identificateurs des concepts atomiques.

- `cnamena(ConceptNonAtom)`
qui précise les identificateurs des concepts non atomiques

- `iname(Instance)`
qui précise les identificateurs des instances.

- `rname(Role)`
qui précise les identificateurs des rôles.

On notera que dans la Tbox, nous ne représenterons pas de relations de subsumption, par souci de simplification.
On constituera un fichier .pl représentant la Tbox et la Abox de l'exercice 3 du TD4.
Ce fichier sera systématiquement chargé avant de pouvoir utiliser le démonstrateur à développer. Bien entendu, si l'on souhaite tester le démonstrateur avec une autre Tbox et une autre Abox, il suffira de créer un nouveau fichier .pl les représentant et de charger ce fichier avant d'utiliser le démonstrateur.

ATTENTION !

Remarque 1 :
Vous devez vous assurer que les définitions de concepts données pour la Tbox ne sont pas circulaires, c'est-à-dire auto-référentes, sinon vous risquez le bouclage dans le traitement des expressions de concepts.

Exemples de définitions de concepts circulaires :
sculpture ≡ objet ⊓ ∀cree_par.sculpteur
sculpteur ≡ personne ⊓ ∃a_cree,sculpture

Si l'on remplace l'identificateur du concept complexe sculpteur par sa définition dans la définition du concept complexe sculpture, on obtient :

sculpture ≡ objet ⊓ ∀cree_par.(personne ⊓ ∃a_cree,sculpture)
C'est ce que l'on appelle une Tbox définitoire avec cycles, qu'il faut éviter.

Remarque 2 :
lComme nous l'avons déjà écrit, il est nécessaire de vérifier la correction syntaxique et la correction sémantique de la Tbox et de la Abox en entrées, ainsi que celles des expressions que l'utilisateur entrera au clavier lorsqu'il sera sollicité pour fournir une proposition à démontrer.

Remarque 3 :
Une fois vérifiée la correction syntaxique et sémantique d'une expression conceptuelle équivalente à un concept complexe défini dans la Tbox, cette expression est remplacée par une expression où ne figurent plus que des identificateurs de concepts atomiques et qui a été mise sous forme normale négative.

Remarque 4 :
Une fois vérifiée la correction syntaxique et sémantique de l'expression conceptuelle d'appartenance d'une instance de la Abox, il s'agit de remplacer tous les identificateurs de concepts complexes d'une telle expression conceptuelle par leurs définitions respectives ne comportant que des concepts atomiques (que l'on peut récupérer dans la Tbox simplifiée que l'on vient de générer). Cette expression doit enfin être mise sous forme normale négative.

Remarque 5 :
On n'oubliera pas de lister dans les clauses donnant les identificateurs de concepts atomiques (prédicat **`cnamea`**) les identificateurs **`anything`** et **`nothing`** qui correspondent respectivement à ⊤ et ⊥.

Implémentation de la partie 1

On va utiliser le fichier suivant représentant la Tbox et la Abox inspirées de l'exercice 3 du TD4. C'est ce fichier que l'on chargera tout de suite après avoir chargé le programme du démonstrateur dans Prolog.

```
equiv(sculpteur,and(personne,some(aCree,sculpture))).
equiv(auteur,and(personne,some(aEcrit,livre))).
```

```
equiv(editeur,and(personne,and(not(some(aEcrit,livre)),some(aEdite
,livre)))).
equiv(parent,and(personne,some(aEnfant,anything))).

cnamea(personne).
cnamea(livre).
cnamea(objet).
cnamea(sculpture).
cnamea(anything).
cnamea(nothing).

cnamena(auteur).
cnamena(editeur).
cnamena(sculpteur).
cnamena(parent).

iname(michelAnge).
iname(david).
iname(sonnets).
iname(vinci).
iname(joconde).

rname(aCree).
rname(aEcrit).
rname(aEdite).
rname(aEnfant).

inst(michelAnge,personne).
inst(david,sculpture).
inst(sonnets,livre).
inst(vinci,personne).
inst(joconde,objet).

instR(michelAnge, david, aCree).
instR(michelAnge, sonnets, aEcrit).
instR(vinci, joconde, aCree).
```

En interne, on codera la Tbox par une liste de doublets :

```
[(sculpteur,and(personne,some(aCree,sculpture))),
(auteur,and(personne,some(aEcrit,livre))),
(editeur,and(personne,and(not(some(aEcrit,livre)),some(aEdite,livr
e)))),
(parent,and(personne,some(aEnfant,anything)))]
```

Idem pour la Abox, que l'on partitionnera en deux sous-listes, une qui contient les assertions de concepts :

```
[(michelAnge,personne), (david,sculpture), (sonnets,livre),
(vinci,personne), (joconde,objet)]
```

et l'autre qui contient les assertions de rôles :

`[(michelAnge, david, aCree), (michelAnge, sonnet, aEcrit),(vinci, joconde, aCree)]`

Ces deux dernières listes seront bien sûr amenées à évoluer, lorsque l'utilisateur soumettra une proposition à la démonstration et que l'arbre de démonstration se développera progressivement.

A plusieurs étapes dans le programme, il sera nécessaire de vérifier la correction sémantique et syntaxique des expressions des concepts atomiques ou complexes.

• Pour ce qui est de la **correction sémantique**, on pourra la vérifier à l'aide des prédicats **`cnamea, cnamena, iname, rname,`** c'est-à-dire que l'on vérifiera que tout identificateur de concept, d'instance, de rôle en est bien un, car on peut obtenir facilement la liste des identificateurs des concepts atomiques, celle des concepts non atomiques, celle des instances et celles des relations.
A cet effet, on pourra utiliser le très pratique prédicat **`setof`** .
`setof(Motif, But, Liste)`
`setof/3` crée une liste des instantiations de Motif par retours arrières sur But et unifie le résultat dans Liste. Si le But n'a pas de solution, **`setof`** retournera la liste vide []. Il y a suppression des doublons.
Exemple : On veut récupérer la liste des instances de la Abox précédente.
On soumet à Prolog le but suivant :
setof(X, iname(X),L).
On va avoir pour réponse : L=[david, vinci, joconde, sonnets, michelAnge]

• Pour ce qui est de la **correction syntaxique**, on dispose de la grammaire des concepts donnée au paragraphe I des Rappels préliminaires et l'on a vu au paragraphe IV de ces mêmes rappels comme il est facile en Prolog de transposer une grammaire en un analyseur syntaxique.

On écrira ainsi le **prédicat `concept`** (correspond au traitement mentionné dans la remarque 2 précédente)**.**

On écrira également les prédicats suivants :

- **`autoref`** qui, à partir d'un nom de concept complexe et de son expression conceptuelle équivalente dans la Tbox, ainsi que d'autres paramètres, permet de tester si ce concept est auto-référent. S'il existe un seul concept complexe auto-référent, le programme se termine en signalant une erreur. (correspond au traitement mentionné dans la remarque 1 précédente). On peut, plutôt que de définir le prédicat **`autoref`**, définir le prédicat **`pas-autoref`** qui, à partir d'un nom de concept complexe et de son expression conceptuelle équivalente dans la Tbox, ainsi que d'autres paramètres, permet de tester si ce concept n'est pas auto-référent.

- **`traitement_Tbox`** en définissant des paramètres pertinents. (correspond au traitement mentionné dans la remarque 3 précédente).

- **`traitement_Abox`** en définissant des paramètres pertinents. (correspond au traitement mentionné dans la remarque 4 précédente).

Partie 2 - Saisie de la proposition à démontrer

I - Types de propositions à prouver

Cette partie a pour but d'acquérir la proposition à prouver à l'aide du démonstrateur.

Cette proposition peut revêtir deux formes :

- L'instance I appartient au concept C (I : C).

- Les concepts C1 et C2 ont une intersection vide (C1 ⊓ C2 ⊑ ⊥).

I est désignée par son identificateur et les concepts peuvent être désignés par leur identificateur ou leur définition sous forme d'expression selon la grammaire des concepts.

Le principe est le même dans les deux cas pour démontrer la proposition, à l'aide de la méthode des tableaux : on ajoute sa négation aux assertions de concepts et aux assertions de rôles de la Abox et l'on démontre que l'ensemble est insatisfiable.

La définition d'un concept dans une telle proposition est donnée en notation préfixée lorsque l'on utilise un opérateur binaire portant sur deux concepts comme intersection ou union.
Par exemple, pour exprimer : A ⊓ B, on utilise and(A,B).
De même, pour A ⊔ B, on utilise or(A,B), pour ¬A, not(A).

De façon analogue, pour des définitions de concepts comportant ∃ ou ∀ :
- pour ∃R:C, l'expression donnée sera : some(R,C)
- pour ∀R:C, l'expression donnée sera : all(R,C)

1er cas
Proposition de type : I : C
La négation de cette proposition que l'on doit ajouter aux assertions de la Abox est : I : ¬C
Avant d'ajouter cette proposition à la Abox, il est nécessaire d'effectuer le traitement suivant sur le concept ¬C :
a) remplacer de manière récursive les identificateurs de concepts complexes par leur définition,
b) mettre les expressions obtenues sous forme normale négative.

2ème cas
Proposition de type : C1 ⊓ C2 ⊑ ⊥
La négation de cette proposition est : ∃ inst, inst : C1 ⊓ C2
De même que dans le cas précédent, il faut traiter de façon similaire l'expression du concept C1 ⊓ C2, avant d'ajouter l'assertion de concept à la Abox.

Comme on vient de le voir, que l'on souhaite démontrer une proposition de l'un ou l'autre des deux types autorisés dans ce démonstrateur, il s'agit d'ajouter à la Abox une nouvelle assertion de concepts. Il s'agira donc de vérifier la syntaxe et la sémantique de l'expression conceptuelle concernée et de la mettre sous forme normale négative, en réutilisant des prédicats définis précédemment à la partie 1.

Implémentation de la partie 2

A ce stade, on peut écrire le prédicat qui démarre le programme global du démonstrateur :

```
programme :-
             premiere_etape(Tbox,Abi,Abr),
             deuxieme_etape(Abi,Abi1,Tbox),
             troisieme_etape(Abi1,Abr).
```

Les prédicats **premiere_etape**, **deuxieme_etape** et **troisieme_etape** correspondent évidemment au travail effectué dans chacune des parties 1, 2 et 3 détaillées dans cet énoncé.

Les paramètres du prédicat **premiere_etape** sont respectivement la liste représentant la Tbox, la liste représentant les assertions de concepts de la Abox et la liste représentant les assertions de rôles de la Abox. On peut écrire à ce stade le prédicat **premiere_etape** grâce aux prédicats définis dans la partie 1 précédente.

Les paramètres du prédicat **deuxieme_etape** sont respectivement la liste des assertions de concepts initiales de la Abox, la liste des assertions de concepts complétée après la soumission d'une proposition à démontrer et la liste représentant la Tbox.

Les paramètres du prédicat **troisieme_etape** sont respectivement la liste des assertions de concepts complétée et la liste des assertions de rôles qui peut également évoluer lors de la démonstration.

L'implémentation de la partie 2 est donnée partiellement :

```
deuxieme_etape(Abi,Abi1,Tbox) :-

               saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox).


saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox) :-
nl,write('Entrez le numero du type de proposition que vous voulez
demontrer :'),nl,
write('1 Une instance donnee appartient a un concept donne.'),nl,
write('2 Deux concepts n"ont pas d"elements en commun(ils ont une
intersection vide).'),nl, read(R), suite(R,Abi,Abi1,Tbox).

suite(1,Abi,Abi1,Tbox) :-
    acquisition_prop_type1(Abi,Abi1,Tbox),!.
suite(2,Abi,Abi1,Tbox) :-
    acquisition_prop_type2(Abi,Abi1,Tbox),!.
suite(R,Abi,Abi1,Tbox) :-
```

```
    nl,write('Cette reponse est incorrecte.'),nl,
    saisie_et_traitement_prop_a_demontrer(Abi,Abi1,Tbox).
```

Les paramètres ont la même signification que précédemment.

Les prédicats **`acquisition_prop_type1`** et **`acquisition_prop_type2`** réalisent respectivement l'acquisition d'une proposition de type 1 et l'acquisition d'une proposition de type 2, puis formatent la proposition de manière adéquate comme cela a été expliqué. La liste représentant les assertions de concepts est complétée.

Partie 3 - Démonstration de la proposition

Cette partie représente le coeur du démonstrateur, puisqu'elle implémente l'algorithme de résolution basé sur la méthode des tableaux.

On doit montrer que l'ensemble des assertions de la Abox étendue, que l'on a construite dans la partie 2 et que l'on va désigner désormais par Abe, est insatisfiable.
On va construire progressivement un arbre de résolution à partir de ces assertions.

Le noeud racine de l'arbre de résolution contient les assertions de Abe.

La boucle de résolution est la suivante :
Sur le noeud en cours, on va tenter d'appliquer l'une des règles suivantes, dans cet ordre (choisi arbitrairement) :

La règle ∃
Si, dans Abe, on trouve une assertion de la forme a : ∃R.C, alors on ajoute les assertions ⟨a, b⟩ : R et b : C, où b est un nouvel objet et l'on génère un nouveau noeud de l'arbre de résolution.

La règle ⊓
Si, dans Abe, on trouve une assertion de la forme a : C ⊓ D, alors on ajoute à Abe les assertions a : C et a : D et l'on génère un nouveau noeud de l'arbre de résolution.

La règle ∀
Si, dans Abe, on trouve des assertions de la forme a : ∀R.C et ⟨a, b⟩ : R, alors on ajoute à Abe l'assertion b : C et l'on génère un nouveau noeud de l'arbre de résolution.

La règle ⊔
Si, dans Ab, on trouve une assertion de la forme a : C ⊔ D, alors on génère deux nouveaux noeuds frères de l'arbre de résolution. Dans l'un, on ajoute à Abe l'assertion a : C et dans l'autre, on ajoute à Abe l'assertion a : D . Ces deux noeuds sont les racines respectives de deux nouvelles branches de l'arbre de résolution.

Si l'une des règles s'applique et qu'elle apporte au moins une nouvelle assertion, on teste s'il y a un clash dans Abe, c'est-à-dire deux assertions de la forme a : C et a : ¬C.

- Si c'est le cas, la branche est fermée et on abandonne le développement de cette branche.
- Si ce n'est pas le cas, on tente d'appliquer à nouveau une règle, puisqu'on sait que Abe a évolué et qu'il est possible qu'une nouvelle règle s'applique de ce fait. On se rebranche donc sur la boucle de résolution.

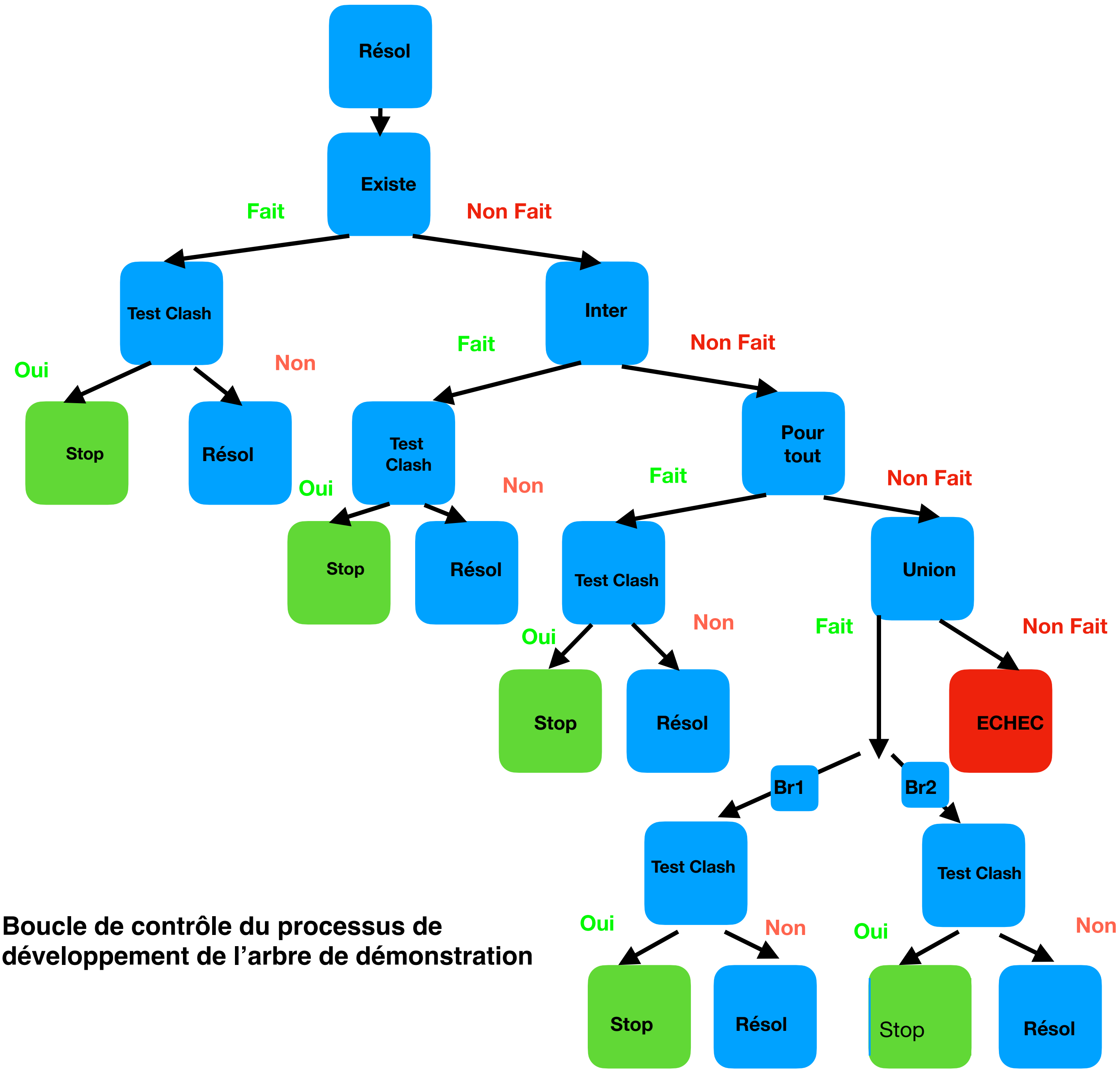
Si aucune des règles ne s'applique, la situation du noeud en cours ne peut plus évoluer et il n'est donc pas possible de trouver un clash. La branche en cours est complète, **il y a échec de la résolution**. On n'a pas pu démontrer la proposition initiale.

Si toutes les branches de l'arbre de résolution sont fermées, Abe est insatisfiable et l'on peut donc affirmer que la **proposition initiale est démontrée**.

Remarques importantes :
1) Il est clair que lorsque l'on applique une règle ⊓ ou une règle ⊔ ou une règle ∃, l'assertion qui a motivé l'application de cette règle peut être retirée de Abe. Elle ne servira plus.
On peut se poser la question au sujet d'une règle ∀ et l'on peut imaginer qu'elle puisse être utilisée plusieurs fois. On rappelle que lorsque l'on applique une règle ∀, c'est que l'on a pu trouver des assertions de la forme a : ∀R.C et 〈a, b〉 : R, on a alors ajouté à Abe l'assertion b : C.
Supposons que l'on ne supprime pas l'assertion a : ∀R.C, après l'avoir appliquée une fois, on pourra appliquer une autre fois cette règle, si l'on trouve une autre assertion de la forme 〈a, b〉 : R dans Abe.
Si, par contre, on applique en une seule fois la règle ∀ à tous les couples possibles d'assertions (∀R.C, 〈a, b〉 : R), si on pouvait l'appliquer une nouvelle fois, cela signifierait qu'une nouvelle assertion 〈a, d〉 : R a été générée par application d'une règle et la seule règle générant une telle assertion aurait pu être la règle a :∃R.C. Or cette règle n'a rien pu apporter comme nouvelle information dans Abe, car l'information qu'elle véhicule, il existe un certain y tel que 〈a, y〉 : R, était évidente puisqu'on avait déjà appliquée la règle ∀R.C au moins une fois et par conséquent, on avait déjà trouvé un tel y !
Ainsi, pour pouvoir supprimer une assertion de la forme ∀R.C de Abe, **il faut faire en une seule fois toutes les déductions possibles** avec toutes les assertions de rôle de la forme 〈a, y〉 : R que l'on peut trouver.

2) Pour ne pas trop compliquer la programmation, on considère que toute règle qui a pu être appliquée a « réussi », même si les assertions qu'elle a permis de déduire n'ont pas été ajoutées à la Abox étendue, car elles y étaient déjà. Dans ce cas, il est ainsi possible de faire des tests de clash inutiles, à la sortie de l'application de règles qui n'ont produit aucune assertion nouvelle, mais cela n'a pas de conséquence néfaste majeure sur le programme.

On peut schématiser ainsi la boucle de contrôle du processus de développement de l'arbre de démonstration.



Boucle de contrôle du processus de développement de l'arbre de démonstration

Implémentation de la partie 3

On n'oubliera pas d'ajouter dans le fichier .pl contenant le programme, la clause :

```
compteur(1).
```

qui initialise un compteur que l'on utilisera dans la génération de nouveaux identificateurs.

Voici la définition du prédicat `troisieme_etape` :

```
troisieme_etape(Abi,Abr) :-
                            tri_Abox(Abi,Lie,Lpt,Li,Lu,Ls),
                            resolution(Lie,Lpt,Li,Lu,Ls,Abr),
                            nl,write('Youpiiiiii, on a demontre la
                            proposition initiale !!!').
```

Le prédicat **`tri_Abox`**, à partir de la liste des assertions de concepts de la Abox étendue après soumission d'une proposition à démontrer, génère 5 listes :
- la liste Lie des assertions du type (I,some(R,C))
- la liste Lpt des assertions du type (I,all(R,C))
- la liste Li des assertions du type (I,and(C1,C2))
- la liste Lu des assertions du type (I,or(C1,C2))
- la liste Ls des assertions restantes, à savoir les assertions du type (I,C) ou (I,not(C)), C étant un concept atomique.

Ce tri est réalisé avant le processus de résolution pour accélérer la recherche d'une assertion susceptible de provoquer l'application de l'une des 4 règles permettant de développer l'arbre de démonstration.

Le prédicat **`resolution`** a pour paramètres les 5 listes Lie, Lpt, Li, Lu, Ls, des assertions de concepts de la Abox étendue et la liste Abr des assertions de rôles de celle-ci.
Ce prédicat va utiliser les prédicats suivants, qui possèdent tous les mêmes paramètres qui traduisent l'état de la Abox étendue à un instant donné :

- `complete_some(Lie,Lpt,Li,Lu,Ls,Abr)`
Ce prédicat cherche une assertion de concept de la forme (I,some(R,C)) dans la liste Lie. S'il en trouve une, il cherche à appliquer la règle ∃ (voir le schéma de la boucle de contrôle présenté plus haut, qui permet de comprendre la structure de ce prédicat).

- `transformation_and(Lie,Lpt,Li,Lu,Ls,Abr)`
Ce prédicat cherche une assertion de concept de la forme (I,and(C1,C2)) dans la liste Li. S'il en trouve une, il cherche à appliquer la règle ⊓ (voir le schéma de la boucle de contrôle présenté plus haut, qui permet de comprendre la structure de ce prédicat).

- `deduction_all(Lie,Lpt,Li,Lu,Ls,Abr)`
Ce prédicat cherche une assertion de concept de la forme (I,all(R,C)) dans la liste Lpt. S'il en trouve une, il cherche à appliquer la règle ∀ (voir le schéma de la boucle de contrôle présenté plus haut, qui permet de comprendre la structure de ce prédicat).

- `transformation_or(Lie,Lpt,Li,Lu,Ls,Abr)`
Ce prédicat cherche une assertion de concept de la forme (I,or(C1,C2)) dans la liste Lu. S'il en trouve une, il cherche à appliquer la règle ⊔ (voir le schéma de la boucle de contrôle présenté plus haut, qui permet de comprendre la structure de ce prédicat).

On écrira également les prédicats suivants (entre autres) :
`evolue(A, Lie, Lpt, Li, Lu, Ls, Lie1, Lpt1, Li1, Lu1, Ls1)`

A représente une nouvelle assertion de concepts à intégrer dans l'une des listes Lie, Lpt, Li, Lu ou Ls qui décrivent les assertions de concepts de la Abox étendue et Lie1, Lpt1, Li1,Lu1 et Ls1 représentent les nouvelles listes mises à jour.

`affiche_evolution_Abox(Ls1, Lie1, Lpt1, Li1, Lu1, Abr1, Ls2, Lie2, Lpt2, Li2, Lu2, Abr2)`

Ce prédicat affiche l'évolution d'un état de la Abox étendue (en ce qui concerne les listes des assertions de concepts et la liste des assertions de rôles) vers un état suivant, les 6 premiers paramètres décrivant l'état de départ, les 6 suivants l'état d'arrivée. Abr1 et Abr2 représentent respectivement les états de départ et d'arrivée de l'ensemble des assertions de rôles de la Abox étendue.

L'affichage doit être agréable pour permettre à l'utilisateur de suivre facilement le développement de l'arbre de démonstration. On affichera donc les différentes assertions en utilisant les symboles mathématiques $\exists$, $\sqcup$, $\neg$, $\forall$, $\sqcap$, et une notation infixée et non préfixée.

Les prédicats à écrire sont obligatoires et doivent respecter les définitions mentionnées dans l'énoncé, mais il est évident qu'ils peuvent utiliser d'autres prédicats plus simples non précisés qu'il vous incombera de définir, et ceci dans les 3 parties du programme Prolog.

Annexe
Quelques utilitaires précieux

`member(X,L)` : prédicat prédéfini, teste si l'élément X appartient à la liste L.

`concat(L1,L2,L3)` : concatène les deux listes L1 et L2 et renvoie la liste L3.

```
concat([],L1,L1).
concat([X|Y],L1,[X|L2]) :- concat(Y,L1,L2).
```

`enleve(X,L1,L2)` : supprime X de la liste L1 et renvoie la liste résultante dans L2.

```
enleve(X,[X|L],L) :-!.
enleve(X,[Y|L],[Y|L2]) :- enleve(X,L,L2).
```

`genere(Nom)` : génère un nouvel identificateur qui est fourni en sortie dans Nom.

```
genere(Nom) :- compteur(V),nombre(V,L1),
               concat([105,110,115,116],L1,L2),
               V1 is V+1,
               dynamic(compteur/1),
               retract(compteur(V)),
               dynamic(compteur/1),
               assert(compteur(V1)),nl,nl,nl,
               name(Nom,L2).
nombre(0,[]).
nombre(X,L1) :-
                R is (X mod 10),
                Q is ((X-R)//10),
                chiffre_car(R,R1),
                char_code(R1,R2),
                nombre(Q,L),
                concat(L,[R2],L1).

chiffre_car(0,'0').
chiffre_car(1,'1').
chiffre_car(2,'2').
chiffre_car(3,'3').
chiffre_car(4,'4').
chiffre_car(5,'5').
chiffre_car(6,'6').
chiffre_car(7,'7').
```

```
chiffre_car(8,'8').
chiffre_car(9,’9’).
```

`flatten(Liste1,Liste2) :` Aplanit Liste1 et met le résultat dans Liste2. Liste1 peut contenir de nombreuses listes imbriquées récursivement. `flatten/2` extrait tous les éléments contenus dans Liste1 et stocke le résultat dans une liste "plane" (à une seule dimension).

BAREME

Le projet est noté sur 26

Une bonne rédaction comprenant en particulier une explication claire des prédicats écrits pourra rapporter un bonus de 1 à 3 points.

Barème de la partie 1 : 8 points
`autoref` : 2 points
`concept` : 2 points
`traitement_Tbox` : 2 points
`traitement_Abox` : 2 points

Barème de la partie 2 : 4 points
`acquisition_prop_type1` : 2 points
`acquisition_prop_type2` : 2 points

Barème de la partie 3 : 14 points
`tri_Abox` : 2 points
`resolution` : 1 point
`complete_some(Lie,Lpt,Li,Lu,Ls,Abr)` : 2 points
`transformation_and` : 3 points
`deduction_all` : 2 points
`transformation_or` : 2 points
`evolue` : 1 point
`affiche_evolution_Abox` : 1 point