Insa de Rennes

Département informatique

Les textes de Travaux Pratiques

Prolog

E. Monnier

TP 1 : Interrogation style base de données

Le monde de la famille.

L'objectif est ici de se familiariser avec l'interprêteur ECLIPSE.
Pour appeler l'interprêteur taper la commande `eclipsep`.
Vous êtes alors prêt(e) à adresser des requêtes.

Entrer les faits correspondant à la famille vue en cours. (est_hom, est_fem, $parent$)

Définir les prédicats $pere(X, Y)$, $mere(X, Y)$, $frere(X, Y)$, $oncle(X, Y)$.

Tester ces prédicats.

Le monde de l'épicerie.

Deux entités sont retenues dans la modélisation de ce monde : l'entité $fournisseur$ et l'entité $produit$.
Pour un fournisseur, seuls nous intéressent son *code*, son *nom*, la *remise* qu'il consent, sa *ville*. D'où la relation f(<codef>,<nomf>,<remisef>,<villef>).
Pour un produit les informations retenues sont son *code*, son *nom*, sa *couleur*, son *origine*.D'où la relation p(<codep>,<nomp>,<coulp>,<originep>).
Ces deux entités sont mises en "relation" grâce au prédicat $fournit$.
Abréviations : f pour $fournisseur$, p pour $produit$, mf pour $fournit$.
Soit l'état du monde suivant :

f(f1,bornibus,5,paris).	*p(p1,cassis,rouge,dijon).*	*mf(f3,p2,5).*
f(f2,mercier,7,paris).	*p(p2,champagne,blanc,reims).*	*mf(f2,p2,1).*
f(f3,colbert,3,reims).	*p(p3,huitre,vert,riec).*	*mf(f1,p6,2).*
f(f4,bossuet,6,dijon).	*p(p4,moutarde,jaune,dijon).*	*mf(f1,p4,1).*
f(f5,tanguy,10,riec).	*p(p5,salade,vert,nice).*	*mf(f1,p1,1).*
f(f6,dupont,0,paris).	*p(p6,cornichon,vert,dijon).*	*mf(f4,p6,3).*
	p(p7,muscadet,blanc,nantes).	*mf(f4,p5,7).*
		mf(f1,p5,8).
		mf(f4,p4,2).
		mf(f3,p4,1).
		mf(f2,p4,1).
		mf(f5,p3,10).

Tapez ces faits et complétez le programme de façon à y inclure les clauses correspondant aux questions suivantes.

- nom des fournisseurs.
- nom des fournisseurs qui fournissent quelque chose.
- nom des fournisseurs qui fournissent au moins un produit vert
- origine des produits qui sont fournis par Mercier.
- nom des produits qui sont fournis par deux fournisseurs au moins.

- nom des fournisseurs qui fournissent au moins un produit originaire de leur ville.
- nom des fournisseurs qui habitent Paris ou fournissent au moins 2 produits verts différents.
- nom des produits pour lesquels il existe une remise égale à 7
- nom des fournisseurs qui me fournissent au moins un produit en quantité inférieure à 5

La requête :
nomf(N) :- f(F,N,R,V), mf(F,P,Q), p(P,NP,C,O), \=(C,vert).
ne correspond pas à la question *nom des fournisseurs qui ne fournissent aucun produit vert*
Pourquoi ?

```
/* pour éviter les réponses en double */
/* on annonce que le predicat dejavu d'arité 1 peut disparaitre-apparaitre */
:-dynamic dejavu/1.

affich(X) :- dejavu(X), !, fail.
affich(X) :- assert(dejavu(X)).

sd :- retract(dejavu(X)),sd.
sd.
```

TP 2 : Manipulation de termes construits

Le monde du Poker

Chaque carte d'un jeu de 52 cartes a
- une hauteur ($deux, trois, quatre, cinq, six, sept, huit, neuf, dix, valet, dame, roi, as$) et
- une couleur ($trefle, carreau, coeur, pique$).

Les hauteurs comme les couleurs sont ici données en ***ordre croissant***.

Au poker une main est constituée de cinq cartes.

1. Ecrire le prédicat *est_carte*, à **un seul argument**, définissant une carte du jeu. La requête :
   est_carte(C); doit donc réussir 52 fois.
2. Ecrire le prédicat *est_main*, à **un seul argument**, définissant une main. La requête :
   est_main(M); doit énumérer toutes les mains possibles d'un jeu de 32 cartes. Il faut bien sûr imposer que toutes les cartes d'une main soient différentes !
   Pour chaque réponse, Eclipse indique le temps machine nécessaire à son élaboration.
   Comment faire en sorte que la première main soit trouvée "très rapidement" ? (temps $< 1\mu$s)

3. Pour évaluer plus facilement une main, il est intéressant d'avoir les cinq cartes en ***ordre croissant***. Pour cela il faut définir la relation $inferieure$ entre toutes cartes $C1$ et $C2$.

$$\begin{array}{lll} C1 < C2 & \text{si} & \text{hauteur(C1)} < \text{hauteur(C2)} \\ & & \text{ou} \\ & \text{si} & \text{hauteur(C1)} = \text{hauteur(C2) et} \\ & & \text{couleur(C1)} < \text{couleur(C2)} \end{array}$$

   Ecrire le prédicat $inf\_carte(C1, C2)$ qui réussit quand la carte $C1$ est $inferieure$ à la carte $C2$. Il y a deux façons de procéder :
   (a) définir la relation d'ordre ci-dessus sans se préoccuper de savoir si $C1$ et $C2$ sont des cartes,
   (b) définir la relation d'ordre ci-dessus en imposant que $C1$ et $C2$ soient des cartes

   Utiliser les 2 versions du prédicat inf_carte pour connaître toutes les cartes inférieures au valet de coeur.
   Comparer les réponses.
   Pourquoi la version (a) est-elle meilleure ?

   A cette occasion essayer de définir le schéma général de la fermeture transitive d'une relation R.

4. Ecrire à nouveau le prédicat *est_main*. (il ne s'agit pas de faire un tri, mais de construire la main triée)
   Pour tester ce prédicat, prendre les mains définies dans *poker.pro* et utiliser des requêtes composées.

5. Ecrire les prédicats évaluant une main dont les cartes sont supposées fournies en ordre croissant.
   ATTENTION : faire en sorte que ces caractérisations soient mutuellement exclusives. C'est-à-dire qu'une main contenant un full ne sera pas retenue comme contenant une paire.
   - *une_paire*(M); M étant instancié, réussit si M contient 2 cartes de même hauteur
   - *deux_paires*(M); M étant instancié, réussit si M contient 2 fois deux cartes de même hauteur
   - *brelan*(M); M étant instancié, réussit si M contient 3 cartes de même hauteur
   - *suite*(M); M étant instancié, réussit si M contient 5 cartes dont les hauteurs se suivent
   - *full*(M); M étant instancié, réussit si M contient une paire et un brelan
   - *carre*(M); M étant instancié, réussit si M contient 4 cartes de même hauteur

TP 3 : Les listes

Cette liste comporte certains exercices vus en TD. Il est néanmoins intéressant d'analyser leur comportement.

0.1 Comprendre le prédicat $is/2$

Programmer et tester les prédicats suivants :

- $lg1(L, N)$: N est le nombre d'éléments de L.
  Constater que ce prédicat n'est pas *réversible* :
  - Pour une **liste donnée** ce prédicat sait calculer sa longueur, tester $lg1([2, 3, 1], N)$.
  - mais il sait ne sait pas construire une liste de **longueur donnée**. tester $lg1(L, 3)$.

  Essayer une aure version :

- $lg2(L, N)$: N est le nombre d'éléments de L.
  - Ce prédicat sait construire une liste de **longueur donnée**. tester $lg1(L, 3)$.
  - mais pour une **liste donnée** ce prédicat ne sait pas calculer sa longueur, tester $lg1([2, 3, 1], N)$.

Le fait que le prédicat qui établit la relation entre une liste et sa longueur ne soit pas réversible est dû au prédicat $is/2$.

0.2 Quelques classiques

- $faitdblliste(L, N)$ N étant donné, construit une liste L de longueur $2 * N$, qui contient 2 fois la même séquence d'éléments.
  Tester la requête :
  [eclipse 29] : L = [1,A,3,1|R], faitdblliste(L,5).

- $conc3(X, Y, Z, T)$ T est la concaténation des listes X, Y et Z.
  $conc3$ sait-il découper la liste T de toutes les façons possibles ?

- $membre(A, X)$ A est élément de la liste X.

- $renverser(X, Y)$ Y est la liste X à l'envers.

- $palind(X)$ X est une liste "palindrome".

- $horsde(A, X)$ A n'est pas élément de la liste X.

- $tousdiff(X)$ les éléments de X sont tous différents.

- $debpar(X, Y)$ la liste X commence par la liste Y.

- $sousliste(X, Y)$ la liste Y est sous-liste de la liste X.

0.3 Un peu plus difficile (quoique !)

- $elim(X, Y)$ X étant donné, on construit la liste Y qui contient tous les éléments de X, une fois.

- $suitmotif(M, L)$ M est une liste d'éléments. La liste L est constituée de la répétition des éléments de M. *[eclipse 30] : suitmotif(M,[2,3,2,3,2,X]).*

- $tri(X, Y)$ la liste Y est le résultat du tri par ordre croissant de la liste d'entiers X.

0.4 Modélisation des ensembles

Dans les exercices suivants les listes PROLOG représentent des ensembles. Nous admettrons donc qu'il n'y a jamais deux mêmes éléments dans une liste. **N'utiliser que** *membre* **et** *horsde* **dans les définitions qui suivent** (et bien sûr la récursivité !)

- $inclus(X, Y)$ tous les éléments de la liste X sont présents dans la liste Y.

- $noninclus(X, Y)$ au moins un élément de la liste X est hors de la liste Y.

- $union(X, Y, Z)$ Z est l'union ensembliste des listes X et Y considérées comme des ensembles.

- $inter(X, Y, Z)$ Z est l'intersection ensembliste des listes X et Y considérées comme des ensembles.

- $diffe(X, Y, Z)$ Z est la différence ensembliste des listes X et Y considérées comme des ensembles.
  [eclipse 31] : diffe([1,5,3],[5,4,7],Z) ;
  Z = [1,3]

TP 4 : Arbres

1. Ecrire un prédicat :
   $faitabin(L, A)$
   "A est un arbre binaire dont les feuilles sont les éléments de la liste L"

   note :
   – on utilisera le symbole fonctionnel *abin* d'arité 2 pour construire l'arbre.
   – on n'attachera pas d'importance à l'ordre des feuilles dans l'arbre.
   – on produira un arbre de profondeur minimale (i.e. pour les listes L dont la taille est n la profondeur de l'arbre sera $\lceil log_2(n) \rceil$.

2. Nous voulons travailler sur des arbres d'arité quelconque et variable selon les noeuds. Pour cela nous choisissons de représenter de tels arbres par des termes.
   Soit $Avar$ le nom de ce domaine.
   – les arbres Prolog réduits à un noeud sont des $Avar$.
   – si L est une liste d'$Avar$, alors $arb(L)$ est un $Avar$.
   Ecrire le prédicat
   $sym(A)$
   "A représente un $Avar$ dont la structure est symétrique".
   (on ne s'intéresse pas aux étiquettes.)

   exemple :
   [eclipse 25] : sym(arb([arb([arb([4]),8]),12,arb([toto,arb([72])])])).
   yes.
   [eclipse 26] : sym(arb([arb([arb([4]),8]),arb([toto,arb([72])])])).
   yes.
   [eclipse 27] : sym(arb([arb([arb([4]),8]),arb([12,0,17]),arb([toto,arb([72])])])).
   yes.
   [eclipse 28] : sym(arb([arb([arb([4 | Y]),8]),arb([12,0,17]),arb([toto,arb([72,11])])])).
   Y = [A].

3. Plus fort : plutôt que de se définir le domaine $Avar$, qui permettait de travailler bien confortablement sur des arbres "sur mesure", nous souhaitons traiter les arbres Prolog.
   Ecrire le prédicat ;
   $symgen(A)$
   "A représente un arbre Prolog dont la structure est symétrique".
   (on ne s'intéresse pas aux étiquettes.)

   exemple :
   [eclipse 35] : symgen([]).
   yes
   [eclipse 36] : symgen([1]).
   yes
   [eclipse 37] : symgen([1,2]).
   no
   [eclipse 38] : symgen(d(5,[1])).
   no

[eclipse 39] : symgen([[1],2]).
yes

4. Ecrire le prédicat :
   $estordo(A)$
   "A représente un $Avar$ (cf. question 2) dont toutes les feuilles sont numériques et tel qu'un parcours descendant gauche droite trouve les nombres en ordre croissant".

   exemple :
   *[eclipse 45] : estordo(*arb*([*arb*([*arb*([4,6]),8]),*arb*([12,13,17]),*arb*([25,*arb*([72])])])).*
   yes
   *[eclipse 46] : estordo(*arb*([*arb*([*arb*([4,6]),5]),*arb*([12,13,17]),*arb*([25,*arb*([72])])])).*
   no

TP 5 : Application partielle

Exercice

Nous souhaitons travailler sur des arbres binaires *Abinexp* correspondant aux expressions arithmétiques qui utilisent les opérateurs + , * et -. Seule la forme de ces arbres nous intéresse et les seules feuilles permises sont étiquetées par la valeur entière 1. Ce domaine *Abinexp* est spécifié par :

$$1 \in Abinexp$$

si $G \in Abinexp$ et $D \in Abinexp$ et $O \in \{mul, add, sub\}$ alors $O(G, D) \in binexp$

Exemple :
$sub(1, add(1, 1))$ est un *Abinexp*

Ecrire un prédicat $constabexp(n, a)$ qui reçoit le nombre entier n et qui produit tous les arbres a à n feuilles du domaine *Abinexp*

Problème

Nous choisissons de représenter des expressions arithmétiques par des termes Prolog. Pour simplifier, nos expressions arithmétiques ne vont comporter que des produits (foncteur $mul/_2$) et des sommes (foncteur $add/_2$).

Un entier est une expression arithmétique,
Une variable est une expression arithmétique,
Si G et D sont des expressions arithmétiques alors $mul(G, D)$ *et* $add(G, D)$ *sont des expressions arithmétiques.*

Une expression arithmétique est **calculable** si elle ne comporte aucune variable.
Simplifier une expression arithmétique E consiste à réécrire E en remplaçant les expressions calculables de E par leur valeur.
Exemple :

$E = mul(X, add(Y, 7))$	E n'est pas calculable	$simpl(E) = mul(X, add(Y, 7))$
$E = mul(X, add(2, 7))$	E n'est pas calculable	$simpl(E) = mul(X, 9)$
$E = mul(3, add(2, 7))$	E est calculable	$simpl(E) = 27$

Définir le prédicat $simpl(E, Es)$ qui à toute expression E associe son expression simplifiée Es.
[eclipse 20] : simpl(mul(X,add(Y,7)),Es).
Es= mul(X,add(Y,7))
[eclipse 21] : Y=2, simpl(mul(X,add(Y,7)),Es).
Es= mul(X,9)
[eclipse 22] : X=3, simpl(mul(X,add(Y,7)),Es).
Es= mul(3,add(Y,7))
[eclipse 23] : X=3,Y=2, simpl(mul(X,add(Y,7)),Es).
Es= 27
Nous choisissons de représenter une fonction par un terme Prolog (foncteur

$fonc/_2$), de la façon suivante : $fonc(Lvar, E)$ où E est une expression arithmétique et $Lvar$ la liste des variables contenues dans cette expression arihmétique.
Ainsi $(x, y, z) -> (x + z) * (y + 7)$
sera représentée par $fonc([X, Y, Z], mul(add(X, Z), add(Y, 7)))$

Appliquer une fonction $fonc(Lvar, E)$ à $Lval$ consiste à calculer la valeur de l'expression E sachant que les variables de $Lvar$ sont remplacées par les valeurs de même rang dans $Lval$. Dans le cas où la liste $Lval$ est plus courte que $Lvar$, seules les premières variables de $Lvar$ sont remplacées et la valeur rendue est une fonction résultat de l'application partielle.
Exemples :
$F = fonc([X, Y, Z], mul(add(X, Z), add(Y, 7)))$
$Lval = [3, 2]$
$appli(F, L) = fonc([Z], mul(add(3, Z), 9))$

Définir le prédicat $appli(F, Lval, Fr)$ qui calcule Fr résultat de l'application de F à $Lval$.
[eclipse 24] : appli(fonc([X,Y,Z],mul(add(X,Z),add(Y,7))),[2,5],R).
R = fonc([Z],mul(add(2,Z),12))
[eclipse 25] : appli(fonc([X,Y,Z],mul(add(X,Z),add(Y,7))),[2,5],R),appli(R,[7],V).
R = fonc([7],mul(add(2,7),12)), V = 108
[eclipse 26] : appli(fonc([X,Y,Z],mul(add(X,Z),add(Y,7))),[2,5,7],R).
R = 108

Notre ambition est maintenant d'appliquer une fonction F à chaque liste d'une liste de listes, et récupérer les résultats dans une liste.
Définir le prédicat $map\_appli(F, L\_Lval, L\_Fr)$ qui calcule la liste L_Fr contenant les résultats de l'application de F à chacune des listes de L_Lval.

[eclipse 27] : map_appl(fonc([X,Y,Z],mul(add(X,Z),add(Y,7))),[[1,2,3],[3,4,5],[5,7,6]],L).
L = [36,88,154]
[eclipse 28] : map_appl(fonc([X,Y,Z],mul(add(X,Z),add(Y,7))),[[1,2,3],[4,5],[5,7,6]],L).
L = [36,fonc([A],mul(add(4,A),12)),154]

Annexe

Les prédicats prédéfinis suivants peuvent vous être utiles :

$var(T)$ réussit si le terme T est réduit à une variable, échoue dans le cas inverse.

$integer(T)$ réussit si le terme T représente un arbre réduit à une feuille dont l'étiquette est un entier connu.

$copy\_term(T1, T2)$ réussit si le terme $T2$ est unifiable avec un terme T qui est une copie à variables renommées du terme $T1$.

[eclipse 29] : copy_term(add(X,mul(7,Y)),T2), X = 3, Y = 5.
T2 = add(X,mul(7,Y))

Y = 5
X = 3

[eclipse 30] : add(*X*,mul(*7*,*Y*)) = *T2*, *X* = *3*, *Y* = *5*.
T2 = add(*3*,mul(*7*,*5*))
Y = 5
X = 3

TP 6 : Prototype de système expert

1. Le système expert qui sature le monde

   Définir les prédicats :
   applicable/$_2$
   productive/$_2$
   appliquer/$_3$
   pour obtenir un moteur d'inférences en chaînage avant qui "sature" le monde.

   Tester.

2. Le système expert qui "chasse" un fait donné

   Aménager le moteur d'inférences de façon à arrêter les inférences dès que qu'un fait donné est démontré.

3. Le système expert qui "mémorise" le fil du raisonnement.

   Aménager le moteur d'inférences de façon à associer à chaque fait la raison pour laquelle il a été produit (numéro de règle, hypothèses instanciées de la règle).

4. Le système expert qui "explique" son raisonnement.

   Quand le moteur arrête les inférences, le monde contient toute la matière première nécessaire à l'affichage de la démonstration.
   Définir le prédicat $affichdemo/_2$ qui pour un fait F donné et un monde M donné provoque l'affichage de la démonstration de F en exploitant les informations contenues dans M. Au moment de l'arrêt du moteur d'inférences appeler ce prédicat.
   Exemple de démonstration :
   [eclipse 25] : finit(M), deduire(M,domest(felix)).
   je sais ronronne(tim) par hypothèse
   je sais pere(tim, felix) par hypothèse
   d apres la regle numero r4
   sachant ronronne(tim) et pere(tim, felix)
   je deduis ronronne(felix)
   je sais familier(tim) par hypothèse
   je sais pere(tim, felix) par hypothèse
   d apres la regle numero r4
   sachant familier(tim) et pere(tim, felix)
   je deduis familier(felix)
   je sais maison(tim) par hypothèse
   je sais pere(tim, felix) par hypothèse
   d apres la regle numero r4
   sachant maison(tim) et pere(tim, felix)
   je deduis maison(felix)

d apres la regle numero r2
sachant ronronne(felix) et familier(felix) et maison(felix)
je deduis domest(felix)

TP 7 : Normalisation d'arbre (mise sous forme clausale

Une proposition est définie comme suit :

- **une proposition atomique (identificateur) est une proposition,**
- **si** f **est une proposition,** $non\ f$ **est une proposition,**
- **si** f **et** g **sont deux propositions,** $f\ et\ g$, $f\ ou\ g$ **sont des propositions.**

Nous allons travailler sur les arbres abstraits de ces propositions.
Exemple :

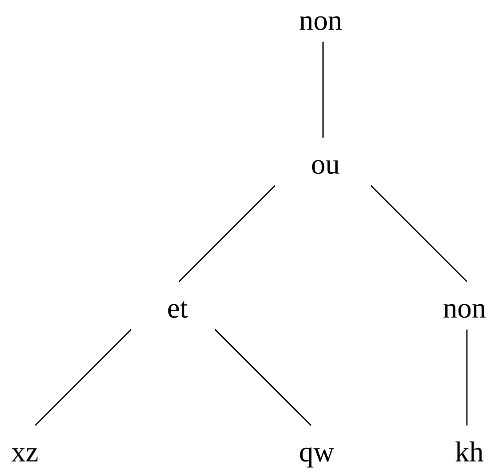

FIG. 1 –

sera noté par le terme : $non(ou(et(xz, qw), non(kh)))$

Votre travail consiste en l'écriture d'un prédicat :
$fnc(F, G)$ spécifié par :
G est une proposition sous forme normale conjonctive, sémantiquement équivalente à F.

Pour faire cela, il faut,

1. partant de F, construire une proposition H, sémantiquement équivalente à F, et telle que les négations portent sur les propositions atomiques. (Se souvenir de ses équivalences remarquables !)

   puis (un peu plus délicat) :

2. partant de H, construire une proposition G, sémantiquement équivalente à H, et telle que les disjonctions sont “en dessous des conjonctions” dans l'arbre abstrait de H.

Vous avez donc deux transformations à programmer et ce n'est pas si facile !

TP 8 : Analyse syntaxique (mise sous forme clausale)

Nous considérons le langage des propositions construit sur les connecteurs $\Rightarrow$ *et ou non*. Les propositions atomiques que nous autorisons sont représentées par des identificateurs (autres que *et*, *ou*, *non*).

Exemple de proposition :

non(mtv => (svp ou bbc) et (kfor => fmi)) et nbc.

L'idée de ce TP est de compléter en amont et en aval le TP de la semaine passée.

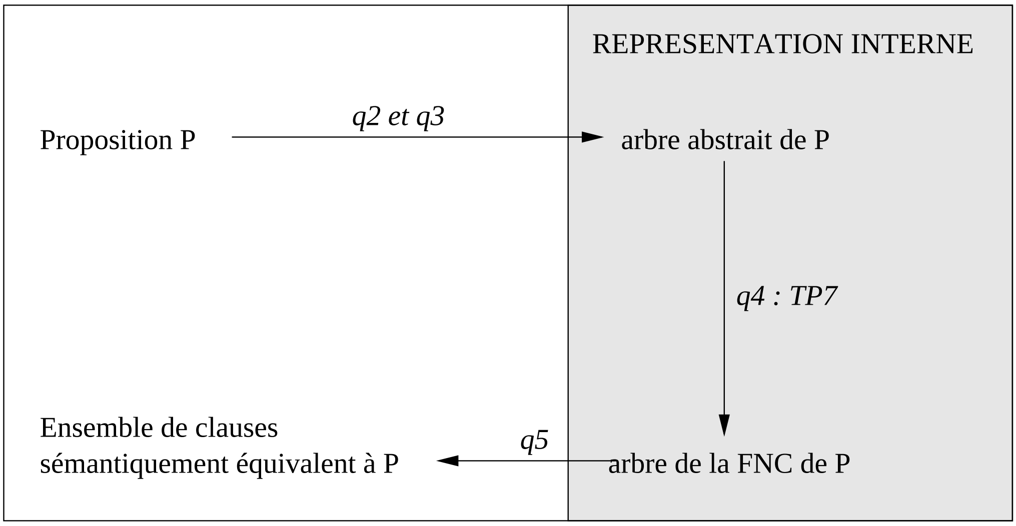

FIG. 2 –

Avant toute chose, construire un analyseur lexical très rudimentaire $analex/_2$. La phrase à analyser doit se terminer par le caractère *point*. Pour écrire cet analyseur lexical utiliser le prédicat read_token$/_2$, qui lit le token suivant dans le flot d'entrée. L'analyseur rend la liste des tokens et la liste des classes de token.

Exemple :
[eclipse 47] : analex(Y,T).
xc => ac ou sb.
T = [atom, atom, atom, atom, atom]
Y = [xc, =>, ac, ou, sb]

1. Concevoir une grammaire des propositions qui prend en compte la priorité des connecteurs

(ordre des connecteurs par priorité décroissante : *non* , *et* , *ou* , =>).

2. Ecrire par la méthode 2 un analyseur du langage des propositions.
   Le tester.

3. Ajouter le calcul d'attributs pour construire un arbre conforme à celui que vous avez normalisé au TP précédent.
   Tester.

4. Insérer votre TP précédent. Enchainer les étapes analyse-génération normalisation

5. Ecrire enfin un "décompilateur" d'arbre normalisé qui affiche l'ensemble de clauses .

$write(S)$: sortie du terme S à l'écran.
nl : passage à la ligne.

TP 9 : DCG et inférence de type

Soit la grammaire d'un sous-ensemble des expressions CAML.

expr : :=		if expr then expr else expr	conditionnelle
	|	function v − > expr	abstraction
	|	(expr expr)	application
	|	[expr { ; expr }* 0/1]	liste
	|	(expr { , expr }*)	nuplet
	|	v	variable
	|	csteentiere	const. entier
	|	true	const. booleen
	|	false	const. booleen

les terminaux sont :
$if\ then\ else\ function\ -> (\ )\ [\ ]\ ;\ ,\ v\ csteentiere\ true\ false$

une variable v est lexicalement réduite à une lettre.

Le but de ce TP est l'écriture d'un analyseur syntaxique de telles expressions, puis la mise en oeuvre d'un calcul d'attributs permettant d'inférer le type de toute expression, c'est-à-dire d'engendrer l'"expression de type" de toute expression. Si t est une expression de type, t appartient au langage engendré par la grammaire :

t : :=		ent
	|	bool
	|	Vt
	|	fonc(t , t)
	|	liste(t)
	|	nuplet(t {, t}*)

où Vt est une variable de type.

Le calcul d'attribut est basé sur des règles d'inférences propres à chaque catégorie d'expression :

$ENV \vdash expr : t$ se prononce :
dans l'environnement de types ENV, l'expression expr a pour expression de type t

$$\frac{}{ENV \vdash csteentiere : ent}$$ cont. entier

$$\frac{}{ENV \vdash true : bool}$$ const. booleen

$$\frac{}{ENV \vdash false : bool}$$ const. booleen

$$\frac{}{ENV \quad tq\{v : \tau\} \in ENV \vdash v : \tau}$$ variable

$$\frac{ENV \vdash expr1 : boolexpr2 : \tau expr3 : \tau}{ENV \vdash if \quad expr1 \quad then \quad expr2 \quad else \quad expr3 : \tau}$$ conditionnelle

$$\frac{ENV \vdash expr1 : fonc(\tau, \tau')expr2 : \tau}{ENV \quad \vdash \quad (expr1 \quad expr2) : \tau'}$$ application

$$\frac{v : \tau \cup ENV \vdash expr : \tau'}{ENV \vdash functionv- > expr : fonc(\tau, \tau')}$$ abstraction

$$\frac{ENV \vdash expr_1 : ....expr_n : \tau}{ENV \vdash [expr_1; ....; expr_n] : liste(\tau)}$$ liste

$$\frac{ENV \vdash expr_1 : \tau_1.........expr_n : \tau_n}{ENV \vdash (expr_1, ..., expr_n) : nuplet(\tau_1, ..., \tau_n)}$$ nuplet

Tester les expressions :

function x − > (5,true,function y − > x)
`fonc(i_4,nuplet(ent,bool,fonc(i_10,i_4)))`

if x then function y − > [x ;y] else function z − > [ ]
`non typable`

(function x − > x 5)
`ent`

(function x − > (x,x) [ ])
`nuplet(liste(i_11),liste(i_11))`

function x − > function y − > if x then [y] else [(x)]
`fonc(bool,fonc(nuplet(bool),liste(nuplet(bool))))`