

Vers la programmation logique

SAMUEL GALLAY

17 mai 2020

1 Introduction au langage Prolog

Prolog, pour programmation en logique est un langage créé par A. Colmerauer, en 1972 à Marseille. La programmation logique est un paradigme différent de la programmation impérative ou de la programmation fonctionnelle. On définit des faits élémentaires et des règles de la manière suivante :

```
apprend(eve, mathematiques).
apprend(benjamin, informatique).
apprend(benjamin, physique).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).
```

```
etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
```

Ces informations décrivent l'ensemble des connaissances du programme. On lit ici “Benjamin apprend l’informatique”, ou “Alice enseigne la physique”. La dernière ligne est une règle : “L’élève E est étudiant du professeur P si E apprend la matière M et que P enseigne M”. Le symbole `:-` se lit *si*, et la virgule entre `apprend(E,M)` et `enseigne(P,M)` signifie *et*. Ensuite l'utilisateur peut effectuer des requêtes comme ceci :

```
?-etudiant_de(E, pierre)
```

L'interpréteur Prolog renvoie alors `E = benjamin` et `E = eve`.

Ce qui différencie la programmation logique des formes plus courantes d'informatique impérative, c'est qu'on n'explique pas au programme comment résoudre la requête, on se contente de décrire le problème.

De très nombreuses introductions au langage Prolog peuvent être trouvées sur internet, je n'en cite qu'une seule qui m'est apparue comme convenable pour une première approche : *Simply Logical : Intelligent Reasoning by Example*, P. Flach, 1994 [2].

2 La représentation des programmes Prolog

La représentation des programmes Prolog dans OCaml se fait de la manière suivante :

- `var` est une variable (commençant par une majuscule en Prolog), identifiée par une chaîne de caractères, et un numéro. Le numéro sera utilisé pour renommer facilement les variables, pour éviter des conflits de noms.
- `atom` est une chaîne de caractères représentant les symboles de prédicats : soit des constantes (arité 0 du prédicat), soit les foncteurs (arité strictement positive). L'arité de `etudiant_de` est 2.
- un `term` est soit une variable, soit sous la forme `prédicat(terme1, terme2, ...)`, qui est donc un atome, et une liste de termes.
- une `clause` est de la forme `etudiant_de(E,P):-apprend(E,M), enseigne(P,M).`, un terme à gauche et une liste de termes à droite. `eleve(samuel).` est une clause, il n'y rien à droite.

Ce n'est pas moi qui ai inventé cette structure particulière : je l'ai retrouvée dans un diapo de cours en Haskell sur le langage Prolog écrit par Alan Smaill en 2009 [6].

Une structure proche est utilisée en Lisp par Peter Norvig dans *Paradigms of Artificial Intelligence Programming*, 1992 [4]. Une citation qui permet de comprendre la structure d'un programme en Prolog : "We will build a single uniform data base of clauses, without distinguishing rules from facts. The simplest representation of clauses is as a cons cell holding the head and the body. For facts, the body will be empty."

J'explique plus bas le *pourquoi* de cette structure, qui permet de représenter les clauses de Horn.

2.1 L'analyse lexicale : le lexer

Il faut d'abord transformer le programme qui est une chaîne de caractères, en une liste de symboles. Les **token** sont les différents symboles présents dans un programme Prolog simple. Des variables et des atomes, et ces caractères spéciaux : (|) | , | . | :-.

Quelques remarques sur le lexer : j'ai bien repris les idées du livre *Le Langage Caml*, Pierre Weis et Xavier Leroy, 1992 [7]. Une grande différence est l'utilisation des List à la place des Stream. L'utilisation des Stream semble plus intelligente, mais ils ont pour des raisons qui m'échappent ils ont presque disparu des dernières versions d'OCaml (le pattern-matching est beaucoup plus difficile). Avec les listes il y a bien plus de copies, mais nous n'avons pas là de véritable problème de performance.

2.2 L'analyse syntaxique : le parser

```
<Programme>    -> <Clause>
<Programme>    -> <Clause> <Programme>
<Clause>       -> <Terme> .
<Clause>       -> <Terme> :- <ListeTermes> .
<ListeTermes>  -> <Terme>
<ListeTermes>  -> <Terme> , <ListeTermes>
<Terme>        -> Variable
<Terme>        -> Predicat
<Terme>        -> Predicat ( <ListeTermes> )

<Terme>        -> <Tableau>
<Tableau>      -> []
<Tableau>      -> [ <ListeTermes> ]
<Tableau>      -> [ <ListeTermes> | <Tableau> ]
```

Si on veut que chaque symbole terminal n'intervienne que dans des productions unitaires :

```
<.>            -> .
<,>            -> ,
<|>           -> |
<:->          -> :-
<Pred>        -> Predicat
<LPar>        -> (
<RPar>        -> )
<LBra>        -> [
<RBra>        -> ]

<Terme>        -> Variable

<Terme>        -> <Pred>
<Terme>        -> <Pred> <LPar> <ListeTermes> <RPar>
<Terme>        -> <Tableau>
```

```

<Programme>    ->  <Clause>
<Programme>    ->  <Clause> <Programme>

<Clause>       ->  <Terme> <.>
<Clause>       ->  <Terme> <:-> <ListeTermes> <.>

<ListeTermes>  ->  <Terme>
<ListeTermes>  ->  <Terme> <,> <ListeTermes>

<Tableau>      ->  <LBra> <RBra>
<Tableau>      ->  <LBra> <ListeTermes> <RBra>
<Tableau>      ->  <LBra> <ListeTermes> <|> <Tableau> <RBra>

```

Il faut maintenant transformer la liste de `tokens` en une structure formée des types `clause`, `term`, `atom` et `var`.

- Isoler les clauses séparées par des points
- Isoler les termes de droite et de gauche d'une clause séparés par `:-`
- Isoler les différents termes d'une suite de termes séparés par des virgules
- Transformer la liste de symboles en un terme
- Transformer les symboles en clauses, le programme en liste de clauses

Je l'ai écrit moi-même, sans reprendre une structure existante... C'est très inefficace : le programme est parcouru au moins 3 fois par les 3 fonctions d'isolement, avant même d'entrer dans la fonctions récursive `tokenlist_to_term` qui sert réellement à créer l'arbre... De plus, le programme est copié plein de fois dans les fonctions de parcours. Néanmoins le coût total me paraît de loin proportionnel à la longueur du code, et probablement proportionnel au niveau d'imbrication `((()))` des parenthèses (ce qui pourrait certainement être évité). Je pense que ce n'est pas le plus important : les coûts d'exécution seront bien plus problématiques...

Je pense qu'il pourrait exister encore des codes Prolog dont la syntaxe est fautive et qui passent le parser. (En tous cas je ne suis pas assuré du contraire). Il faut se demander si l'on peut représenter des programmes faux dans les structures formées des types `clause`, `term`, `atom` et `var`.

Remarque : vous êtes priés de ne rentrer que des codes justes dans le parser, les messages d'erreur sont plus que lacunaires...

3 L'algorithme derrière Prolog

Ma référence en ce qui concerne l'algorithme utilisé par Prolog est *Logic Programming and Prolog*, 1990, Nilsson et Małuszyński [3].

Ce qui semble être le premier article traitant du type de résolution utilisé par Prolog est *A Machine-Oriented Logic Based on the Resolution Principle*, Robinson, 1965 [5]. Cet article introduit l'unification, et le *principe de résolution* (Resolution Principle) dont dérive la SLD-resolution (Linear resolution for Definite clauses with Selection function) utilisée par Prolog.

Une *clause définie*, aussi nommée clause de Horn est de la forme $(P_1 \text{ et } P_2 \text{ et } \dots \text{ et } P_n) \Rightarrow Q$. En particulier, $(\text{non } P) \Rightarrow Q$ n'est pas une clause de Horn. On peut par contre exprimer $(P_1 \text{ ou } P_2) \Rightarrow Q$ avec des clauses de Horn, il suffit d'écrire les deux clauses $P_1 \Rightarrow Q$ et $P_2 \Rightarrow Q$. Les versions premières de Prolog se limitent aux clauses de Horn, pour une principale raison : la *correction* et la *complétude* (*soundness* and *completeness*) ont été montrée pour la SLD-resolution, qui n'est valide que sur des clauses de Horn. Pour les citations (un peu compliquées) : la SLD-resolution est introduite par Kowalski en 1974, la correction est montrée par Clark en 1979, la complétude a été montrée premièrement par Hill en 1974, mais quelque chose de plus fort a été montré par Clark en 1979.

3.1 L'unification

L'unification se fait entre deux termes A et B. Deux termes s'unifient s'il existe une substitution θ des variables de A telle que $B = \theta(A)$.

Soit la clause du programme Prolog `etudiant_de(E,P):-apprend(E,M), enseigne(P,M)`.. Si l'on cherche à réaliser la requête `?-etudiant_de(E, pierre)`, on unifie `etudiant_de(E, pierre)` avec `etudiant_de(E,P)`, le membre de gauche de la clause. La substitution θ remplace P par `pierre` et ne modifie pas les autres variables. Pour continuer la recherche, on applique la substitution au termes de droite de la clause.

Voici l'algorithme décrit dans [3] :

E est l'ensemble des équations. Au départ il n'y en a qu'une seule :
Par exemple $E = \{\text{etudiant_de}(E,P) = \text{etudiant_de}(E, \text{pierre})\}$.

Répéter tant que E change

(jusqu'à ce que l'on ne puisse plus rien appliquer aux équations)

Sélectionner une équation $s = t$ dans E;

Si $s = t$ est de la forme :

$f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$ avec $n \geq 0$

Alors remplacer l'équation par $s_1=t_1 \dots s_n=t_n$

$f(s_1, \dots, s_m) = g(t_1, \dots, t_n)$ avec $f \neq g$

Alors ÉCHEC

$X = X$

Alors supprimer l'équation

$t = X$ où t n'est pas une variable

Alors remplacer l'équation par $X = t$

$X = t$ où $X \neq t$ et X apparaît plus d'une fois dans E

Si X est un sous-terme de t Alors ÉCHEC

Sinon on remplace toutes les autres occurrences de X par t

Il est prouvé que cet algorithme termine et renvoie soit échec, soit un ensemble équivalent des équations sous *forme résolue*. Un ensemble d'équations $\{X_1 = t_1, \dots, X_n = t_n\}$ est dit sous forme résolue si les (X_n) sont des variables, les (t_n) sont des termes et aucune des variables X_n n'apparaît dans les t_n . Cela permet ensuite de déterminer un unifieur.

Remarques :

- C'est dommage que l'algorithme se prête si bien à la programmation impérative quand on programme dans un langage fonctionnel.
- On trouve des algorithmes pour déterminer un unifieur qui ne manipulent pas exactement comme ça des systèmes d'équations, (et qui sont dans des styles plus fonctionnels), mais cet algorithme est le seul que j'arrive à comprendre convenablement, et en plus [3] prouve sa terminaison et sa correction, ce qui est très bon point.
- Le test **Si X est un sous-terme de t Alors ÉCHEC** est en pratique pas réalisé dans la plupart des implémentations Prolog (pas comme ça en tout cas), il est très lent et le cas n'arrive que très peu en pratique. On peut donc avoir des boucles infinies... Les versions "modernes" gèrent les unifications de structures infinies... Une citation de [4] "*This represents a circular, infinite unification. Some versions of Prolog, notably Prolog II (Giannesini et al. 1986), provide an interpretation for such structures, but it is tricky to define the semantics of infinite structures.*"

À la place de manipuler des ensembles (Set existe en OCaml), j'ai juste utilisé des liste triées. La fonction `compare` est magique, elle définit une relation d'ordre pour n'importe quel type. J'ai ici utilisé la version complète de l'algorithme, qui n'est certainement pas la plus efficace. Comme l'a dit Donald Knuth : "*Premature optimization is the root of all evil (or at least most of it) in programming.*"

On définit une substitution comme une liste de couples (var,term), le terme est inséré à la place de la variable.

Si l'on sait que `etudiant_de(E1,P):-apprend(E1,M), enseigne(P,M)`. (c) et que l'on veut montrer `etudiant_de(E0, pierre)`. (r), alors il faut trouver une substitution qui unifie la tête de la clause (c) avec la requête (r).

Pour l'exemple plus haut, il faut la substitution $\theta = \{ P/pierre, E1/E0 \}$. Il faut comprendre "La variable P est remplacée par 'pierre', la variable E1 est remplacée par la variable E0. C'est bien un unifieur, si on l'applique sur la requête et sur la tête de la clause, les deux deviennent égales.

Il peut exister plusieurs unifieurs, par exemple $\theta_2 = \{ P/pierre, E0/samuel, E1/samuel \}$ en est aussi un. On voit que θ_2 , c'est θ composé avec $\{E0/samuel\}$, on dit alors que θ est plus général que θ_2 . Nous, nous cherchons le plus général de ces unifieurs, le MGU pour *Most General Unifier*.

Un détail : on peut avoir θ plus général que ω , et ω plus général que θ , la relation n'est donc pas antisymétrique. En fait, le MGU est unique au renommage des variables près.

On peut déduire facilement un MGU du système d'équations sous forme résolue de la fonction `solve` (c'est à ça qu'elle sert). Si $\{X_1 = t_1, \dots, X_n = t_n\}$ est sous forme résolue, alors $\{X_1/t_1, \dots, X_n/t_n\}$ est un MGU.

Lorsque l'on veut unifier `etudiant_de(E,P)` et `etudiant_de(E,pierre)`, les deux variables E ne doivent pas être nommées de la même manière.

Quelques explications sur les derniers deux exemples :

- Je sais que pour tout Z, $f(g(Z), Z)$ est vraie, et j'aimerais savoir s'il existe des couples (X,Y) tels que $f(X, g(Y))$ soit vraie. Le programme me répond oui, il suffit de choisir $X = g(g(Y))$
- Je sais que pour tout (X,Y), $f(X, g(Y))$ est vraie, et j'aimerais savoir s'il existe des Z tels que $f(g(Z), Z)$ soit vraie. Le programme me répond oui, il suffit de choisir $Z = g(Y)$

3.2 Le backtracking

Il me semble que nous nous rapprochons du but ! L'unification est une partie très importante de la SLD-resolution. L'autre point important est le backtracking. Encore quelques fonctions pour appliquer des substitutions sur des termes, listes de termes, pour composer des substitutions, pour vérifier que un terme et une clause n'ont pas de variables en commun, pour effectuer les renommages si nécessaire, et après je pense qu'il sera possible d'implémenter le backtracking.

Voilà la manière dont j'ai compris l'algorithme :

- On cherche à satisfaire une requête $\leftarrow A_1, \dots, A_n$.
- On a A_1 qui unifie avec H_i , où $H_i \leftarrow C_{i_1}, \dots, C_{i_m}$ est une la i-ème clause du programme. On a auparavant renommé toutes les variables de $H_i \leftarrow C_{i_1}, \dots, C_{i_m}$ qui étaient présentes dans A_1 , sinon on ne peut pas appeler mgu. On a donc un MGU θ_1 .
- La nouvelle requête à satisfaire est $\leftarrow \theta_1(C_{i_1}, \dots, C_{i_m}, A_2, \dots, A_n)$.
- On récurse, si on a à montrer $\theta_k(\dots \theta_{k+1}(\theta_{k+2}(\dots \theta_{k+m}(_vide_)) \dots)$ alors c'est gagné ($_vide_$ signifie toujours vrai), la composée θ_{tot} des θ_k est une substitution des variables de la requête de départ. Ce qu'on veut renvoyer à l'utilisateur c'est l'image des variables de A_1, \dots, A_n par θ_{tot} .
- Si A_1 ne s'unifie avec aucun H_i , ça ne sert à rien d'essayer d'unifier A_2 , puisque de toute façon on garde A_1 dans la nouvelle requête. Alors il faut remonter. C'est à dire qu'il faut essayer les autres unifications à l'étape d'avant.

Comment renommer les clauses pour avoir des noms libres à chaque unification ? On peut utiliser le niveau de récursion comme identifiant, que l'on place dans l'entier transporté avec la variable.

- On veut des variables numérotées à 0 dans la requête.
- On passe 1 lors du premier appel de sld : la première clause utilisée est renommée à 1, les variables de la requête sont toutes à 0.

- La deuxième clause utilisée est renommée à 2, dans la requête il y a des variables à 1 et à 0 : pas de conflit ! etc...

3.3 Que faire maintenant ?

- Beaucoup de tests pour vérifier si tout fonctionne.
- Peut-être nettoyer un peu le code...
- Écrire quelques fonctions comme une ligne de commande interactive, pour une utilisation plus facile. Rendre les résultats plus clairs.
- Proposer une deuxième version de la recherche, qui renvoie tous les résultats possibles, ou qui propose de chercher le résultat suivant.
- Je pensais essayer d'écrire un programme pour “résoudre” le Cluedo. J'avais déjà essayé sans succès en C++. Le problème semble plus adapté au langage Prolog : “*tel joueur possède telle carte*”, “*si un joueur montre une carte à un autre pour réfuter une supposition (triplet de cartes), c'est qu'il possède une des trois cartes*”, sont des règles que l'on peut écrire en Prolog. Malheureusement j'ai trouvé un super mémoire de Master [1] qui traite bien le sujet.
- Je regarde un peu comment est-ce que l'on peut sortir des clauses de Horn, mais c'est un sujet compliqué.
- Je voulais voir s'il est possible d'étendre à des langages fonctionnels des éléments de programmation logique. C'est aussi un thème intéressant, qui se nomme *functional logic programming*. On trouve plusieurs petits langages sur internet qui utilisent des approches différentes, mais aucun d'eux ne semble bien abouti. Ce serait presque un idéal d'associer des mécanismes efficaces des langages fonctionnels comme l'évaluation paresseuse, et les capacités de la programmation logique pour décrire les problèmes...
- Pour compiler Prolog, il faut se tourner vers les *Machines abstraites de Warren*.

Références

- [1] V. I. Aartun. Reasoning about knowledge and action in cluedo using prolog. 2016.
- [2] P. Flach. *Simply Logical : Intelligent Reasoning by Example*. 1994.
- [3] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. 1990.
- [4] P. Norvig. *Paradigms of Artificial Intelligence Programming*. 1992.
- [5] J. A. Robinson. A machine-oriented logic based on the resolution principle. 1965.
- [6] A. Smaill. Logic programming. 2009.
- [7] P. Weis and X. Leroy. *Le langage Caml*. 1992.

A Déchiffrement de la syntaxe Prolog

```
[ ]: type var = Id of string * int

type table = Empty | NonEmpty of term * table
and term = Var of var | Predicate of string * (term list) | Table of table

type clause = Clause of term * (term list);;
```

A.1 Le nouveau Parser :

```
[ ]: type structure = S of string | V of var | L of table | T of term | C of clause
                        | W of clause list | P of string | TL of term list
```

```
[ ]: let to_list str = List.init (String.length str) (String.get str);;
let rec remove_spaces = function
| [] -> []
| (' ' | '\t' | '\n')::l -> remove_spaces l
| a::l -> a::(remove_spaces l);;
let (|>) a b = b a;;
let of_char c = String.make 1 c;;
```

```
[ ]: let ( <+> ) ra rb = fun l -> (ra l) @ (rb l)

let chain y l = List.map (fun (a, b) -> (a, y@b)) l

let ( ** ) ra rb =
let chain y l = List.map (fun (a, b) -> (a, y@b)) l in
    fun l -> List.concat (
        List.map (fun (x, y) -> chain y (rb x)) (ra l)
    )

let ( >~> ) ra f = fun l -> List.map (fun (a, b) -> (a, f b)) (ra l)

let skip = fun l -> []

let fix aux = let rec g x = aux g x in g
```

```
[ ]: let rlow = function
| ('a'..'z' as c)::l -> [l, [S (of_char c)]]
| _ -> []

let rup = function
| ('A'..'Z' as c)::l -> [l, [S (of_char c)]]
| _ -> []

let rsym s = function
| h::t when s = h -> [t, [S (of_char s)]]
| _ -> []

let rchar = function
| ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' as c)::t -> [t, [S (of_char c)]]
```

```

| _ -> []

let rmot =
let aux self = rchar
    <+> rchar *** self >~> (function | [S s; S s'] -> [S (s^s')]) | 1 -> 1)
in fix aux

let variable =
    rup >~> (function | [S s] -> [V (Id (s,0))]) | 1 -> 1)
<+> rup *** rmot >~> (function | [S s; S s'] -> [V (Id ((s^s'),0))]) | 1 -> 1)

let predicate = rlow >~> (function | [S s] -> [P s]) | 1 -> 1)
    <+> rlow *** rmot >~> (function | [S s; S s'] -> [P (s^s')]) | 1 -> 1)

let rec term l = (
    variable >~> (function | [V s] -> [T (Var s)]) | 1 -> 1)
<+> predicate >~> (function | [P s] -> [T (Predicate (s,[]))]) | 1 -> 1)
<+> predicate *** (rsym '(' >~> skip) *** termlist *** (rsym ')' >~> skip)
    >~> (function | [P s; TL tl] -> [T (Predicate (s, tl))]) | 1 -> 1)
) 1
and termlist l = (
    term >~> (function | [T t] -> [TL [t]]) | 1 -> 1)
<+> term *** (rsym ',' >~> skip) *** termlist >~> (function | [T t; TL tl] -> [TL (t::tl)]) | 1 -> 1)
) 1

let clause = term *** (rsym '.' >~> skip) >~> (function | [T t] -> [C (Clause (t,[]))]) | 1 -> 1)
    <+> term *** (rsym ':' *** rsym '-' >~> skip) *** termlist *** (rsym '.' >~> skip)
    >~> (function | [T t; TL tl] -> [C (Clause (t,tl))]) | 1 -> 1)

let programme =
let aux self = clause >~> (function | [C c] -> [W [c]]) | 1 -> 1)
    <+> clause *** self >~> (function | [C c; W cl] -> [W (c::cl)]) | 1 -> 1)
in fix aux

let check r l =
let aux = function
| [] , result -> Some result
| _ -> None
in match List.filter_map aux (r l) with
| [] -> failwith "Parsing failed"
| [a] -> a
| _ -> failwith "Ambiguous grammar";;

let parse_program s = match (s |> to_list |> remove_spaces |> (check programme)) with
| [W w] -> w
| _ -> failwith "input is not a program"

```


B Algorithme d'unification

```
[ ]: (* Teste si une variable est dans un terme *)
let rec var_in_term v = function
| Var w -> v = w
| Term (_, l) -> List.mem true (List.map (var_in_term v) l)
| PList Empty -> false
| PList (PLCons (l, t)) -> List.mem true (List.map (var_in_term v) l) || var_in_term v t;;

var_in_term (VarT ("P", 0)) (parser_term "[A,[B,Z],C | [E | P] ]");;
```

```
[ ]: (* Remplace une variable par un nouveau terme, récursivement à l'intérieur d'un terme *)
let rec replace_var_in_term var new_t = function
| Var v when v = var -> new_t
| Var v -> Var v
| Term (a, l) -> Term (a, List.map (replace_var_in_term var new_t) l)
| PList Empty -> PList Empty
| PList (PLCons (l, t)) -> PList
(PLCons ((List.map (replace_var_in_term var new_t) l), (replace_var_in_term var new_t t)));;

replace_var_in_term (VarL ("E",0)) (parser_term "pierre") (parser_term "etudiant_de([E,B |
↪E],P)");;
```

```
[ ]: (* Teste si une variable est dans une liste de couples de termes *)
let var_in_eql v e = List.mem true (List.map (function t -> var_in_term v (fst t) ||
↪var_in_term v (snd t)) e);;

(* Remplace une variable par un nouveau terme, à l'intérieur d'une liste de couples de termes ↪
↪*)
let replace_var_in_eql var new_t eql =
List.map (function (a, b) -> (replace_var_in_term var new_t a), (replace_var_in_term var
↪new_t b)) eql;;
```

```
[ ]: (* E est une liste de couples de termes *)
(* transforme E en un équivalent sous forme résolue (type option, None si impossible) *)
let rec solve e =
  (* Parcourt une fois l'ensemble des équations *)
  let rec pass tmp = function
  | [] -> Some tmp

  | (Term (f, lf), Term (g, lg))::l when f = g && List.length lf = List.length lg ->
    pass ((List.map2 (fun a b -> a,b) lf lg)@tmp) l

  | (Term (_, _), Term (_, _))::l -> None (* failwith "Unification failed" *)

  | (Var x, Var y)::l when x = y -> pass tmp l

  | (Term (a, la), Var x)::l -> pass ((Var x, Term (a, la))::tmp) l

  | (Var x, t)::l when var_in_eql x tmp || var_in_eql x l ->
    if var_in_term x t then None (* failwith "Unification failed, loop" *)
    else pass ((Var x, t)::(replace_var_in_eql x t tmp)) (replace_var_in_eql x t l)
```

```

| (Var x, t)::l -> pass ((Var x, t)::tmp) l

in match (pass [] e) with
| None -> None
| Some eqlpp -> let b = List.sort_uniq compare eqlpp in if e = b then Some e else solve b;;

```

```

[ ]: solve [parser_term "etudiant_de(E, P)", parser_term "etudiant_de(S, P)"];; (* E = S *)

```

```

[ ]: (* Prend 2 terme et renvoie l'unifieur le plus général s'il existe, None sinon.
Attention, aucune variable ne doit être présente dans (r) la requête et dans (c) la tête de l
↪ la clause *)

```

```

let rec mgu r c = let f = function
  | Var x, Term (a, l) -> x, Term (a, l)
  | Var x, Var y -> x, Var y
  | Term (_, _), _ -> failwith "Should not happen."
in match solve [r, c] with
| None -> None
| Some e -> Some (List.map f e);;

```

```

[ ]: (* La fonction test_unification a pour but de tester mgu *)
let test_unification str_r str_c = mgu (parser_term str_r) (parser_term str_c);;

test_unification "etudiant_de(E, pierre)" "etudiant_de(F,P)";; (* F/E et P/pierre *)

test_unification "etudiant_de(F,P)" "etudiant_de(E, pierre)";; (* E/F et P/pierre *)

test_unification "f(X,g(Y))" "f(g(Z),Z)";; (* X/g(g(Y)) et Z/g(Y) *)

test_unification "f(g(Z),Z)" "f(X,g(Y))";; (* X/g(g(Y)) et Z/g(Y) *)

```

C Algorithme de backtracking

```

[ ]: (* Applique une substitution sur un terme *)
let rec apply_subst_on_term uni term = match uni with
| [] -> term
| (v,t)::l -> apply_subst_on_term l (replace_var_in_term v t term);;

(* Applique une substitution sur une liste de termes *)
let rec apply_subst_on_termlist uni lst = List.map (apply_subst_on_term uni) lst;;

```

```

[ ]: (* Prend un entier et une clause, renvoie la clause avec les variables renommées à l'entier *)
let rename n (Clause (t1, tl)) =
let rec f = function
| Var (VarT (str, _)) -> Var (VarT (str, n))
| Term (atm, l) -> Term (atm, List.map f l)
in Clause (f t1, List.map f tl);;

rename 42 (parser_clause "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)");;

```

```
[ ]: type 'a tree = Leaf of 'a | Node of (('a tree) Lazy.t) list;;

(* La fonction de recherche, renvoie l'arbre des solutions *)
let rec sld_tree world req subs n = match req with
| [] -> Leaf subs
| head_request_term::other_request_terms ->
    Node (List.filter_map (fun c -> let Clause (left_member, right_member) = rename n c in
        (match mgu head_request_term left_member with
         | None -> None
         | Some unifier -> Some (lazy (sld_tree world
            (apply_subst_on_termlist unifier (right_member@other_request_terms))
            (unifier::subs) (n+1)))
        )) world);;

[ ]: (* Compose correctement la liste de substitutions *)
let compose l =
let compose2 s1 s2 = s1@(List.map (function (v2, lt2) -> (v2, apply_subst_on_term s1 lt2)) s2)
in List.fold_left compose2 [] l;;

(* Recherche les termes qui sont des variables récursivement dans une liste de termes *)
let rec find_vars_in_termlist tl = List.sort_uniq compare (List.concat (List.map
(function
| Var v -> [Var v]
| Term (_,l) -> find_vars_in_termlist l)
tl));;

[ ]: let list_to_seq l = List.fold_right (fun x s -> (fun ()->Seq.Cons(x, s)) ) l Seq.empty;;

let rec to_seq = function
| Leaf str -> Seq.return str
| Node tl -> Seq.flat_map (fun par -> to_seq (Lazy.force par)) (list_to_seq tl);;

let solutions world req = let tree = sld_tree world (parser_term_list req) [] 1 in
let vars = (find_vars_in_termlist (parser_term_list req)) in
Seq.map (fun l -> vars, apply_subst_on_termlist (compose l) vars) (to_seq tree);;

let request world req = let sol = solutions world req in
if sol () = Seq.Nil then Format.printf "This is false.\n%!"
else Seq.iter (fun (vars, tl) ->
if vars = [] then Format.printf "This is True.\n%!"
else Format.printf "There is : %s\n%!"
(String.concat ", " (List.map2 (fun v t -> (term_to_string v) ^ " = " ^ (term_to_string t))
vars tl))
) sol;;
```

D Essais de l'implémentation

D.1 Tests basiques... des élèves, des cours et des professeurs

```
[ ]: let program = "apprend(eve, mathematiques).
apprend(benjamin, informatique).
apprend(benjamin, physique).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).
etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
";

lexer program;;
```

```
[ ]: extract_clauses (lexer program);;

extract_rightleft (lexer "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)");;

extract_term_list_old (lexer "apprend(E,M), enseigne(P,M)");;

extract_term_list (lexer "apprend([A,B,C]), enseigne([B,C,A]), [a(A),b(B),c(C) | [] ]");;

inside_parenthesis (lexer "(E,M)");;

inside_parenthesis (lexer "[ (E), [M], A ]");;

tokenlist_to_term (lexer "apprend(benjamin, informatique)");;

tokenlist_to_term (lexer "[benjamin, informatique, [A,B,C | T ] ]");;

let world1 = parser program;;

parser_term "apprend(benjamin, informatique)";;

parser_term_list "apprend(E,M), enseigne(P,M)";;

parser_clause "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)";;
```

```
[ ]: request world1 "etudiant_de(E, pierre)";;
```

```
[ ]: request world1 "etudiant_de(E, pierre), etudiant_de(E, alice)";;
```

```
[ ]: request world1 "etudiant_de(A, B)";;
```

```
[ ]: request world1 "etudiant_de(A, A)";;
```

```
[ ]: request world1 "apprend(A, A)";;
```

```
[ ]: request world1 "enseigne(A, A)";;
```

```
[ ]: request world1 "enseigne(alice, physique)";;
```

```
[ ]: request world1 "enseigne(alice, mathematiques)";;
```

D.2 Le métro londonien

```
[ ]: let program = "  
connected(bond_street,oxford_circus,central).  
connected(oxford_circus,tottenham_court_road,central).  
connected(bond_street,green_park,jubilee).  
connected(green_park,charing_cross,jubilee).  
connected(green_park,piccadilly_circus,piccadilly).  
connected(piccadilly_circus,leicester_square,piccadilly).  
connected(green_park,oxford_circus,victoria).  
connected(oxford_circus,piccadilly_circus,bakerloo).  
connected(piccadilly_circus,charing_cross,bakerloo).  
connected(tottenham_court_road,leicester_square,northern).  
connected(leicester_square,charing_cross,northern).  
  
nearby(X,Y):-connected(X,Y,L).  
nearby(X,Y):-connected(X,Z,L),connected(Z,Y,L).  
  
reachable(X,Y):-connected(X,Y,L).  
reachable(X,Y):-connected(X,Z,L),reachable(Z,Y).  
  
path(X,Y,noroute):-connected(X,Y,L).  
path(X,Y,route(Z,R)):-connected(X,Z,L),path(Z,Y,R).";; (*connected(X,Y,L):-connected(Y,X,L).*)
```

```
[ ]: let world = parser program;;
```

```
[ ]: request world "connected(piccadilly_circus,leicester_square,piccadilly)";;
```

```
[ ]: request world "nearby(oxford_circus, charing_cross)";;
```

```
[ ]: request world "nearby(tottenham_court_road,W)";;
```

```
[ ]: request world "reachable(bond_street, leicester_square)";;
```

```
[ ]: request world "connected(oxford_circus, bond_street, L)";;
```

```
[ ]: request world "path(oxford_circus,charing_cross,R)";;
```

```
[ ]: program;;
```