Vers la programmation logique

Samuel Gallay

25 mai 2020

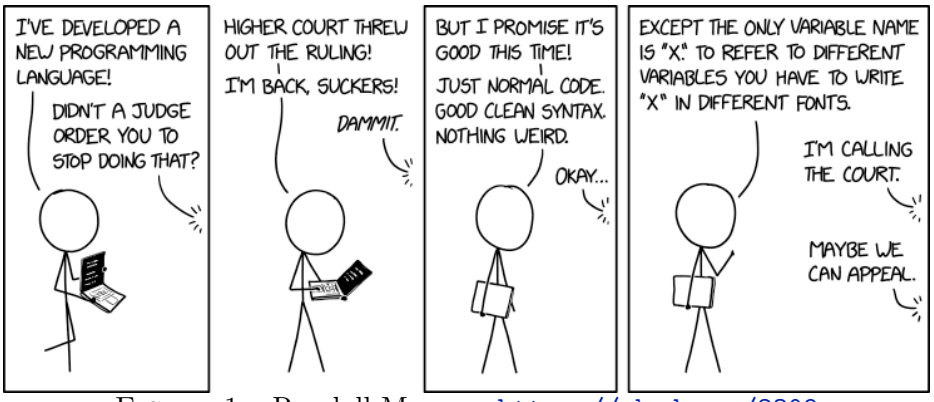

Figure 1 – Randall Munroe, https://xkcd.com/2309

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 P enseigne M". Le symbole `:-` se lit *si*, et la virgule entre `apprend(E,M)` et `enseigne(P,M)` sinifie *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 : Flach (1994).

Un autre ouvrage plutôt plus complet est Sterling et Shapiro (1994).

Dans la suite j'essaie d'implémenter moi même un interpréteur d'un sous-ensemble du langage Prolog, en utilisant le langage OCaml qui a la particularité d'être principalement un langage fonctionnel.

2 La grammaire du langage Prolog

La première chose à faire est d'étudier la grammaire du langage, c'est à dire comment sont formés les mots du langage (en linguistique on parle de morphologie), mais aussi comment peuvent s'agencer les mots pour former un programme valide (c'est la syntaxe).

La grammaire du langage Prolog est non-contextuelle, ce qui signifie que toutes les règles ne contiennent à gauche qu'un seul symbole. Plus intuitivement, pour décrire un symbole non-terminal (entre chevrons) on a pas besoin de connaitre ce qu'il y a avant ou après, d'où le "non-contextuel". J'en suis arrivé à cette définition de la grammaire de mon sous-ensemble du langage Prolog, ici présentée sous la forme de Backus-Naur :

```
<Caractère>    ::=  'a'..'z' | 'A'..'Z' | '_' | '0'..'9'
<Mot>          ::=  <Caractère> | <Caractère> <Mot>
<Prédicat>     ::=  'a'..'z' | 'a'..'z' <Mot>
<Variable>     ::=  'A'..'Z' | 'A'..'Z' <Mot>
<Programme>    ::=  <Clause> | <Clause> <Programme>
<Clause>       ::=  <Terme> '.' | <Terme> ':-' <ListeTermes> '.'
<ListeTermes>  ::=  <Terme> | <Terme> ',' <ListeTermes>
<Terme>        ::=  <Variable> | <Predicat> | <Predicat> '(' <ListeTermes> ')' | <Tableau>
<Tableau>      ::=  '[]' | '[' <ListeTermes> ']'
            | '[' <ListeTermes> '|' <Tableau> ']' | '[' <ListeTermes> '|' <Variable> ']'
```

Les grammaires non-contextuelles sont équivalentes aux automates à pile : ces derniers sont plus puissants que les automates finis (équivalents aux expressions régulières), mais moins puissants que les machines de Turing.

Une sous classe des grammaires non contextuelles est celle des grammaires non-contextuelles déterministes. Une grammaire non contextuelle est dit déterministe si elle est équivalente à un automate à pile déterministe. Cette sous catégorie est intéressante puisqu'il existe alors des algorithmes en temps linéaire pour effectuer l'analyse syntaxique… Je suppose que cette grammaire de Prolog est bien déterministe, mais je ne sais pas le montrer.

De toute façon la structure de l'analyseur syntaxique que je vais utiliser me permet en théorie d'utiliser n'importe quelle grammaire non-contextuelle.

3 La représentation des programmes Prolog

C'est probablement la partie la plus importante du design de l'interpréteur Prolog, puisque la manière de stocker les programmes Prolog en Caml impactera toute la suite.

```
type var = Id of string * int
type term = Var of var | Predicate of string * (term list)
type clause = Clause of term * (term list);;
```

Cette première structure permet de représenter le plus petit Prolog possible. Chaque variable est représentée par une chaîne de caractères, son nom, et un numéro. Le numéro permet un renommage facile des variables, qui est nécessaire pour éviter des conflits noms dans l'algorithme d'unification. Le type term est le plus important, un terme peut être une variable, où une structure composée d'une chaîne, le prédicat et une liste de termes. Ainsi `apprend(eve, mathématiques)` est un terme. Finalement une clause est un terme, qui est à gauche du symbole `:-`, et la liste des termes qui sont à droite. Un programme sera alors une liste de clauses, séparées par des points.

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, Smaill (2009). Une structure proche est utilisée en Lisp par Peter Norvig dans Norvig (1992).

```
type var = Id of string * int
type table = Empty | NonEmpty of term * table | TVar of var
and term = Var of var | Predicate of string * (term list) | Table of table
type clause = Clause of term * (term list);;
```

L'ajout des listes Prolog (je les ai appelées tables ici pour les différencier des listes du langage Caml, mais ce n'est probablement pas le meilleur nom), rend la structure sensiblement plus complexe. J'ai choisi ce design pour pouvoir modéliser facilement une table comme `[a, b, c | X]`. Ici `X` est une variable qui ne peut que représenter une table, on ne veut pas que cela puisse être un terme quelconque (comme `eve` exemple). C'est une très bonne chose de rendre impossible la représentation de programmes invalides dans le type des objets, mais cela rend certaines choses plus compliquées. Les variables sont présentes à différents endroits, et certaines ne peuvent représenter que des tableaux alors que d'autres peuvent représenter n'importe quel terme.

J'ai mis un moment (plus ou moins les deux derniers mois) à me décider sur cette structure, et c'est parce que j'ai dû la changer un certain nombre de fois que j'ai réécrit beaucoup de choses pour qu'elles soient un peu plus modulables.

4 L'analyse syntaxique

Si j'ai bien appris une chose en écrivant un premier parser moi même sans reprendre de grandes idées existantes, c'est qu'il mieux vaut bien réfléchir avant si on veux être capable de modifier, ou même de relire son code après. Maintenant j'ai compris que le parsing est un art en soi, et j'ai décidé d'écrire un analyseur récursif descendant utilisant des combinateurs. J'ai appris les idées dans Ljunglöf (2002).

Je vois ici un parser comme une fonction qui à une chaîne de caractères à analyser renvoie un couple composé de la chaîne restante à analyser et une structure, un arbre produit par l'analyseur qui représente les caractères analysés. L'idée des combinateurs est de créer de nouveaux parsers plus complexes à partir de ceux existants. Voici la structure que produit le 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
```

Les parsers élémentaires que j'ai écrit directement permettent de reconnaître un caractère passé en paramètre et renvoient une structure `S chaîne` avec chaîne étant le caractère reconnu. J'en ai écrit d'autres permettant de reconnaître n'importe quelle lettre majuscule, ou minuscule, ou encore n'importe quel caractère valide.

Ensuite j'utilise deux combinateurs : `<+>` et `<*>`. Pour des raisons de priorité des opérateurs en OCaml, je les ai renommés `+~+` et `*~*` respectivement. Le combinateur `<+>` exprime le *ou* `|` des règles de grammaire. Le combinateur `<*>` quant à lui premet de chaîner les parsers, il n'est pas représenté dans les règles de grammaire, les symboles sont juste accolés. Ainsi la règle `<Mot> ::= <Caractère> | <Caractère> <Mot>` se traduirait par :

```
let rec mot = caractère +~+ caractère *~* mot
```

Malheureusement, on ne peut pas exprimer directement la récursion ainsi… Même si cela fonctionne très bien pour les règles non récursives, une limitation de OCaml —pour éviter de déclarer des structures comme `let rec a = a`— impose d'écrire quelques lignes en plus, rendant le code moins lisible.

Le dernier opérateur que je définis est `>~>`, d'une manière très similaire au *bind* `>>=` monadique. Cet opérateur prend à gauche un parser, et à droite une fonction d'une liste de structures vers une structure (en simplifiant). La fonction peut par exemple décrire comment construire un nouveau mot à partir d'un caractère et d'un mot.

5 Les fonctions de base sur les programmes

- `string_of_term : term -> string` Transforme un terme en une chaîne de caractères.
- `var_in_term : var -> term -> bool` Teste si une variable est dans un terme.
- `var_in_eql : var -> (term * term) list -> bool` Teste si une variable est dans une liste de couples de termes.

— `replace_var_in_term : var -> term -> term -> term` Remplace une variable par un terme dans un terme. Attention, la fonction échoue si on essaie de remplacer une variable qui est censée représenter un tableau par un terme qui n'est pas un tableau.

— `replace_var_in_eql : var -> term -> (term * term) list -> (term * term) list` Remplace une variable par un terme dans une liste de couples de termes. Mêmes limitations.

— `find_vars_in_termlist : term list -> term list` Liste toutes les variables d'une liste de termes.

— `find_tvars_in_term : term -> var list` Liste toutes les variables représentant un tableau dans un terme.

— `find_tvars_in_termlist : term list -> var list` Liste toutes les variables représentant un tableau dans une liste de termes.

— `find_tvars_in_clause : clause -> var list` Liste toutes les variables représentant un tableau dans une clause.

6 La correction des types

Le problème est qu'une même variable ne peut pas représenter n'importe quel terme si l'on sait que la variable ne peut représenter que des tableaux. La solution est de rechercher toutes les variables qui ne peuvent représenter que des tableaux, et forcer toutes leurs occurences par des variables qui ne peuvent représenter que des tableaux.

Ensuite j'ai juste écrit des fonctions composant le parser et le vérificateur de types.

Dans toute la suite du code il ne faut jamais qu'une variable appraisse à un endroit comme pouvant représenter n'importe quel terme et dans un autre comme pouvant ne représenter que des tableaux.

7 L'algorithme derrière Prolog

Ma référence en ce qui concerne l'algorithme utilisé par Prolog est Nilsson et Małuszyński (1990).

Ce qui semble être le premier article traitant du type de résolution utilisé par Prolog est Robinson (1965). 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 `(P1 et P2 et ... et Pn) => Q`. En particulier, `(non P) => Q` n'est pas une clause de Horn. On peut par contre exprimer`(P1 ou P2) => Q` avec des clauses de Horn, il suffit d'écrire les deux clauses `P1 => Q` et `P2 => 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.

8 L'unification

L'unification se fait entre deux termes A et B. Deux termes s'unifient s'il existe une substitution `thêta` des variables de A telle que `B = thêta(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 `?-étudiant_de(E, pierre)`, on unifie `étudiant_de(E, pierre)` avec `etudiant_de(E,P)`, le membre de gauche de la clause. La substitution thêta 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 Nilsson et Małuszyński (1990) :

```
E est l'ensemble des équations. Au départ il n'y en a qu'une seule :
Par exemple E = {etudiant_de(E,P) = étudiant_de(E, 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(s1, ..., sn) = f(t1, ..., tn) avec n >= 0
            Alors remplacer l'équation par s1=t1 ... sn=tn
        f(s1, ..., sm) = g(t1, ..., tn) avec f != 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 != t et X apparait plus d'une fois dans E
            Si X est un sous-terme de t Alors ÉCHEC
            Sinon on remplace toutes les autres occurences 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 `{X1 = t1, ..., Xn = tn}` est dit sous forme résolue si les `(Xn)` sont des variables, les `(tn)` sont des termes et aucune des variables `Xn` n'apparait dans les `tn`. 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 Nilsson et Małuszyński (1990) 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 Norvig (1992) "*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 `étudiant_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 `thêta = { 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 `thêta2 = { P/pierre, E0/samuel, E1/samuel}` en est aussi un. On voit que thêta2, c'est thêta composé avec `{E0/samuel}`, on dit alors que thêta est plus général que thêta2. Nous, nous cherchons le plus général de ces unifieurs, le MGU pour *Most General Unifier*.

Un détail : on peut avoir thêta plus général que oméga, et oméga plus général que théta, 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 `{X1 = t1, ..., Xn = tn}` est sous forme résolue, alors `{X1/t1, ..., Xn/tn}` 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)`

9 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 `<- A1,...An`.

— On a `A1` qui unifie avec `Hi`, où `Hi <- Ci_1,..Ci_m` est une la i-ème clause du programme. On a auparavant renommé toutes les variables de `Hi <- Ci_1,..Ci_m` qui étaient présentes dans `A1`, sinon on ne peut pas appeler `mgu`. On a donc un MGU `thêta1`.

— La nouvelle requête à satisfaire est `<- thêta1(Ci_1,...Ci_m, A2,...An)`.

— On récure, si on a à montrer `thêta_k( ... thêta2( thêta1( _vide_) ) ...)` alors c'est gagné (`_vide_` signifie toujours vrai), la composée `thêta_tot` des `thêta_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 `A1,...An` par `thêta_tot`.

— Si `A1` ne s'unifie avec aucun `Hi`, ça ne sert à rien d'essayer d'unifier `A2`, puisque de toute façon on garde `A1` 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…

10 Le Prolog aujourd'hui

J'ai beaucoup aimé la lecture de la préface de Sterling et Shapiro (1994). Robert Kowalski raconte son premier contact avec Prolog, le langage ayant été initié vers 1972 par Alain Colmerauer et Philippe Roussel à Marseille. Kowalski y est invité, mais les serveurs de calcul ne sont pas là, ils se connectent par telnet sur une machine IBM

à Grenoble ! Pour la petite histoire, le premier message que Kowalski reçoit à l'exécution de son programme est "DEBORDEMENT DE PILE" !

Kowalski a introduit le SLD-résolution, entre autre nombreux travaux sur les algorithmes de Prolog. Il a eu deux étudiants en thèse. Keith Clark, qui a montré la correction et la complétude de la SLD-résolution et qui a introduit le principe de *négation as failure*. David H. D. Warren lui a écrit le premier compilateur pour Prolog, utilisant les *machines abstraites de Warren*.

11 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 Aartun (2016) 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 fonctionels 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

Vemund Innvær Aartun : Reasoning about knowledge and action in cluedo using prolog. 2016.

Peter Flach : *Simply Logical : Intelligent Reasoning by Example*. 1994.

Peter Ljunglöf : Pure functional parsing, an advanced tutorial. 2002.

Ulf Nilsson et Jan Małuszyński : *Logic, Programming and Prolog*. 1990.

Peter Norvig : *Paradigms of Artificial Intelligence Programming*. 1992.

John Alan Robinson : A machine-oriented logic based on the resolution principle. 1965.

Alan Smaill : Logic programming. 2009.

Leon Sterling et Ehud Shapiro : *The Art of Prolog*. 2nde édition, 1994.

A Déchiffrement de la syntaxe Prolog

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

type table = Empty | NonEmpty of term * table | TVar of var
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 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')] | l -> l)
in fix aux
```

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

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

let rec table_from_list tl t = match tl with
| h::q -> NonEmpty (h, table_from_list q t)
| [] -> t

let rec term l = (
    variable >~> (function | [V s] -> [T (Var s)] | l -> l)
<+> predicate >~> (function | [P s] -> [T (Predicate (s,[]))] | l -> l)
<+> predicate *~* (rsym '(' >~> skip) *~* termlist *~* (rsym ')' >~> skip)
        >~> (function | [P s; TL tl] -> [T (Predicate (s, tl))] | l -> l)
<+> table >~> (function | [L t] -> [T (Table t)] | l -> l)
) l
and termlist l = (
term >~> (function | [T t] -> [TL [t]] | l -> l)
<+> term *~* (rsym ',' >~> skip) *~* termlist >~> (function | [T t; TL tl] -> [TL (t::tl)] | l -> l)
) l
and table l = (
rsym '[' *~* rsym ']' >~> (function | [S "["; S "]"] -> [L Empty] | l -> l)
<+> (rsym '[' >~> skip) *~* termlist *~* (rsym ']' >~> skip)
    >~> (function | [TL tl] -> [L (table_from_list tl Empty)] | l -> l)
<+> (rsym '[' >~> skip) *~* termlist *~* (rsym '|' >~> skip) *~* table *~* (rsym ']' >~> skip)
    >~> (function | [TL tl; L t] -> [L (table_from_list tl t)] | l -> l)
<+> (rsym '[' >~> skip) *~* termlist *~* (rsym '|' >~> skip) *~* variable *~* (rsym ']' >~> skip)
    >~> (function | [TL tl; V v] -> [L (table_from_list tl (TVar v))] | l -> l)
) l


let clause = term *~* (rsym '.' >~> skip) >~> (function | [T t] -> [C (Clause (t,[]))] | l -> l)
            <+> term *~* (rsym ':' *~* rsym '-' >~> skip) *~* termlist *~* (rsym '.' >~> skip)
                    >~> (function | [T t; TL tl] -> [C (Clause (t,tl))] | l -> l)
let programme =
let aux self = clause >~> (function | [C c] -> [W [c]] | l -> l)
            <+> clause *~* self >~> (function | [C c; W cl] -> [W (c::cl)]| l -> l)
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"

let parse_term s = match (s |> to_list |> remove_spaces |> (check term)) with
| [T t] -> t
| _ -> failwith "input is not a term"
```

```
let parse_clause c = match (c |> to_list |> remove_spaces |> (check clause)) with
| [C a] -> a
| _ -> failwith "input is not a clause"

let parse_termlist str = match (str |> to_list |> remove_spaces |> (check termlist)) with
| [TL tl] -> tl
| _ -> failwith "input is not a termlist"
```

B Fonctions de base

[ ]:
```
module TermSet = Set.Make(struct
   type t = term * term
   let compare = compare
end)

module VarMap = Map.Make(struct
   type t = var
   let compare = compare
end)
```

[ ]:
```
(* Transforme un terme en une chaîne de caractères *)
let rec string_of_term = function
| Predicate (p, []) -> p
| Predicate (p, l) -> p ^ "(" ^ (String.concat ", " (List.map string_of_term l)) ^ ")"
| Var (Id (s, _)) -> s
| Table (Empty) -> "[]"
| Table (TVar (Id (s,_))) -> s
| Table (NonEmpty (t, l)) ->
let rec aux = function
| Empty -> ""
| TVar (Id (s, _)) -> "|"^s
| NonEmpty (t, l) -> ", "^(string_of_term t)^(aux l)
in "["^(string_of_term t)^(aux l)^"]"

(* DEBUG : affiche un liste de couple de termes *)
let print_eqs eqs =
TermSet.iter (fun (t1,t2) -> Format.printf "%s <-> %s\n%!" (string_of_term t1) (string_of_term t2)) eqs
```

[ ]:
```
(* Applique une substitution sur un terme *)
(* Attention, on ne peut remplacer une variable qui représente un tableau que par un tableau *)
let rec replace_var_in_term var new_term term = match term with
| Var v -> if v = var then new_term else Var v
| Predicate (p, l) -> Predicate (p, List.map (replace_var_in_term var new_term) l)
| Table tbl ->
let rec aux = function
| Empty -> Empty
| TVar v -> if v=var then
(match new_term with | Table nt -> nt
| _ -> failwith "Try to replace a table by a term that is not a table")
else TVar v
| NonEmpty (head, tail) -> NonEmpty (replace_var_in_term var new_term head, aux tail)
in Table (aux tbl)

let replace_var_in_eqs var new_term eqs =
TermSet.map (fun (a, b) -> replace_var_in_term var new_term a, replace_var_in_term var new_term b) eqs

let rec var_in_term var = function
```

```
| Var v -> if v=var then true else false
| Predicate (p, l) -> List.mem true (List.map (var_in_term var) l)
| Table t -> let rec aux = function
| Empty -> false
| NonEmpty (head, tail) -> var_in_term var head || aux tail
| TVar v -> if v=var then true else false
in aux t

let var_in_eqs var eqs =
TermSet.exists (function (a, b) -> var_in_term var a || var_in_term var b) eqs;;


(* Recherche les termes qui sont des variables récursivement dans une liste de termes *)
let rec find_vars_in_termlist tl =
let rec find_vars_in_term = function
| Var v -> [Var v]
| Predicate (_, l) -> find_vars_in_termlist l
| Table t ->
let rec aux = function
| Empty -> []
| TVar v -> [Table (TVar v)]
| NonEmpty (head, tail) -> (find_vars_in_term head)@(aux tail)
in aux t
in List.sort_uniq compare (List.concat (List.map find_vars_in_term tl));;

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'un terme *)
let rec find_tvars_in_term = function
| Var _ -> []
| Predicate (_, l) -> List.concat (List.map find_tvars_in_term l)
| Table t ->
let rec aux = function
| Empty -> []
| NonEmpty (head, tail) -> (find_tvars_in_term head) @ (aux tail)
| TVar v -> [v]
in aux t

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'une liste de termes *)
let rec find_tvars_in_termlist tl = List.concat (List.map find_tvars_in_term tl)

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'une clause *)
let find_tvars_in_clause (Clause (t, tl)) = (find_tvars_in_term t) @ (find_tvars_in_termlist tl)
```

C Vérification des types

[ ]:
```
let type_check_term term =
List.fold_left (fun t v -> replace_var_in_term (v) (Table (TVar v)) t) term (find_tvars_in_term term)

let type_check_cl cl =
let l = find_tvars_in_clause cl in
List.fold_left (fun cl v ->
let aux = replace_var_in_term (v) (Table (TVar v))
and Clause (left, right) = cl in
Clause (aux left, List.map aux right)
) cl l

let read_term str = type_check_term (parse_term str)
let read_program str = List.map type_check_cl (parse_program str)
```

```
let read_clause str = type_check_cl (parse_clause str)
let read_termlist str = List.map type_check_term (parse_termlist str)
```

D Algorithme d'unification

[ ]:
```
(* E est un ensemble couples de termes *)
(* transforme E en un équivalent sous forme résolue (type option, None si impossible) *)
type equation_set = {mutable data : TermSet.t; mutable correct : bool}

let rec solve e =
let eqs = {data = e; correct = true} in

let aux = function
| Predicate (f, lf), Predicate (g, lg) when f = g && List.length lf = List.length lg ->
    List.iter2 (fun a b -> eqs.data <- TermSet.add (a,b) eqs.data) lf lg;
    eqs.data <- TermSet.remove (Predicate (f, lf), Predicate (g, lg)) eqs.data;

| Predicate (_, _), Predicate (_, _) ->
    eqs.correct <- false;

| Var x, Var y when x = y ->
    eqs.data <- TermSet.remove (Var x, Var y) eqs.data

| Predicate (a, la), Var x ->
    eqs.data <- TermSet.remove (Predicate (a, la), Var x) eqs.data;
    eqs.data <- TermSet.add    (Var x, Predicate (a, la)) eqs.data;

| Table t, Var x ->
    eqs.data <- TermSet.remove (Table t, Var x) eqs.data;
    eqs.data <- TermSet.add    (Var x, Table t) eqs.data;

| Var w, t when var_in_eqs w (TermSet.remove (Var w, t) eqs.data) ->
    if var_in_term w t then eqs.correct <- false
    else (
    eqs.data <- TermSet.remove (Var w, t) eqs.data;
    eqs.data <- replace_var_in_eqs w t eqs.data;
    eqs.data <- TermSet.add (Var w, t) eqs.data;
    )

| Var v, t ->
    ()

| Predicate (_, _), Table _ ->
    eqs.correct <- false

| Table _, Predicate (_, _) ->
    eqs.correct <- false

| Table t1, Table t2 -> (
    match t1, t2 with
    | Empty, Empty ->
        eqs.data <- TermSet.remove (Table Empty, Table Empty) eqs.data;

    | Empty, NonEmpty _ ->
        eqs.correct <- false;

    | NonEmpty _, Empty ->
```

```
        eqs.correct <- false;

    | (NonEmpty (head1, tail1) as t1), (NonEmpty (head2, tail2) as t2) ->
        eqs.data <- TermSet.remove (Table t1, Table t2) eqs.data;
        eqs.data <- TermSet.add (head1, head2) eqs.data;
        eqs.data <- TermSet.add (Table tail1, Table tail2) eqs.data;

    | (NonEmpty (_, _) | Empty as t), TVar a ->
        eqs.data <- TermSet.remove (Table t, Table (TVar a)) eqs.data;
        eqs.data <- TermSet.add (Table (TVar a), Table t) eqs.data;

    | TVar a, TVar b when a=b ->
        eqs.data <- TermSet.remove (Table (TVar a), Table (TVar b)) eqs.data;

    | TVar w, table when var_in_eqs w (TermSet.remove (Table (TVar w), Table table) eqs.data) ->
        if var_in_term w (Table table) then eqs.correct <- false
        else eqs.data <- replace_var_in_eqs w (Table table) eqs.data;

    | TVar _ , _ ->
        ()
    )
in
let old = ref TermSet.empty in
while eqs.data <> !old && eqs.correct do
old := eqs.data;
TermSet.iter (fun c -> if eqs.data <> !old then () else aux c) eqs.data;
done;

if eqs.correct then Some eqs.data else None
```

[ ]:
```
(* 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 la clause *)

let rec mgu r c = let f = function
    | Var x, t -> x, t
    | Predicate (_, _), _ -> failwith "Should not happen."
    | Table (TVar r), t -> r, t
    | Table (Empty | NonEmpty _), _ -> failwith "Should not happen."
in match solve (TermSet.singleton (r, c)) with
| None -> None
| Some e -> Some (e |> TermSet.to_seq |> Seq.map f |> VarMap.of_seq);;
```

[ ]:
```
(* La fonction test_unification a pour but de tester mgu *)
let test_unification str_r str_c = Format.printf "Unification de %s et de %s :\n%!" str_r str_c;
match mgu (str_r |> read_term) (str_c |> read_term) with
| Some t -> VarMap.iter
    (fun (Id (s, a)) term -> Format.printf "%s <- %s\n%!" s (string_of_term term)) t
| None -> Format.printf "No unification\n%!"
;;

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) *)
```

```
test_unification "[a, a, b, c]" "[A | B]";

test_unification "[a, a, b, c]" "[A | A]";

test_unification "[[a, b, c], a, b, c]" "[A | A]"
```

E Algorithme de backtracking

[ ]:
```
(* Applique une substitution *)
let apply_subst_on_term = VarMap.fold replace_var_in_term;;

(* Applique une substitution sur une liste de termes *)
let apply_subst_on_termlist uni = apply_subst_on_term uni |> List.map;;
```

[ ]:
```
(* 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 (Id (str, _)) -> Var (Id (str, n))
| Predicate (atm, l) -> Predicate (atm, List.map f l)
| Table t ->
let rec aux = function
| Empty -> Empty
| TVar (Id (str, _)) -> TVar (Id (str, n))
| NonEmpty (head, tail) -> NonEmpty (f head, aux tail)
in Table (aux t)
in Clause (f t1, List.map f tl);;
```

[ ]:
```
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);;
```

[ ]:
```
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 (read_termlist req) [] 1 in
let vars = (find_vars_in_termlist (read_termlist req)) in
Seq.map  (fun l -> vars,
List.fold_right apply_subst_on_termlist 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 -> (string_of_term v) ^ " = " ^ (string_of_term t)) vars tl))
) sol;
Format.printf "\n%!";;
```

[ ]:
```
type 'a ftree = FLeaf of 'a | FNode of ('a ftree) list;;

let rec force_tree (tree : 'a tree) = match tree with
| Leaf a -> FLeaf a
| Node atl -> FNode (List.map (fun t -> force_tree (Lazy.force t)) atl)
```

F Mémoïsation

[ ]:
```
(* Renvoie un set de (Var Model -> Var Request) *)

let equivalent r m =
let rec equal map_opt request model = match map_opt with
| None -> None
| Some map ->
match request, model with
| Var vr, Var vm -> (match VarMap.find_opt vm map with
    | None -> Some (VarMap.add vm vr map)
    | Some vs -> if vs = vr then Some map else None)
| Predicate (pr, lr), Predicate (pm, lm) ->
    if pr <> pm || List.length lr <> List.length lm then None
    else List.fold_left2 equal (Some map) lr lm
| Table tr, Table tm ->
    let rec aux map_opt tbl_r tbl_m = match map_opt with
    | None -> None
    | Some map -> (
    match tbl_r, tbl_m with
    | Empty, Empty -> Some map
    | TVar vr, TVar vm -> (match VarMap.find_opt vm map with
        | None -> Some (VarMap.add vm vr map)
        | Some vs -> if vs = vr then Some map else None)
    | NonEmpty (head_r, tail_r), NonEmpty (head_m, tail_m) ->
        aux (equal (Some map) head_r head_m) tail_r tail_m
    | _, _ -> None
    ) in aux (Some map) tr tm
| _, _ -> None
in equal (Some (VarMap.empty)) r m;;

let test_eq s1 s2 = Option.iter
(VarMap.iter (fun key v -> Format.printf "%s -> %s\n%!" (string_of_term (Var key)) (string_of_term␣
↪(Var v))))
(equivalent (read_term s1) (read_term s2));;

test_eq "table(A, B)" "table(C, C)";;

test_eq "table(A, A)" "table(B, C)"
```

G Essais de l'implémentation

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

[ ]:
```
let world1 = read_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).
" in

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)";;
```

G.2 Le métro londonien

[ ]:
```
let world = read_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)."
in

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)";;
```

G.3 Les listes

[ ]:
```
let world4 = read_program "
member(X, [X|Xs]).
member(X, [Y|Ys]) :- member(X, Ys).

prefix([], Ys).
prefix([X|Xs],[X|Ys]) :- prefix(Xs, Ys).

sublist(Xs, Ys) :- prefix(Xs, Ys).
sublist(Xs, [Y|Ys]) :- sublist (Xs, Ys).

append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).

reverse([],[]).
reverse([X|Xs], R) :- reverse(Xs, Z), append(Z, [X], R).

adjacent(X,Y,Zs) :- append(As, [X,Y|Ys] ,Zs).

equal(X, X).


" in

request world4 "member(a, [c,d,a,b])";
request world4 "prefix(P, [c,d,a,b])";
request world4 "sublist(S, [a,b,c,d])";
request world4 "append([a,b,c], [d,e,f], X)";
request world4 "reverse([a,b,c,d,e,f], R)";
request world4 "adjacent(X, Y, [a,b,c,d])";
request world4 "equal([a, b], X)";;
```

[ ]:
```
let world5 = read_program "
satisfiable(true).
satisfiable(and(X, Y)) :- satisfiable(X), satisfiable(Y).
satisfiable(or(X, Y)) :- satisfiable(X).
satisfiable(or(X, Y)) :- satisfiable(Y).
satisfiable(not(X)) :- invalid(X).

invalid(false).
invalid(or(X, Y)) :- invalid(X), invalid(Y).
invalid(and(X, Y)) :- invalid(X).
invalid(and(X, Y)) :- invalid(Y).
invalid(not(X)) :- satisfiable(X).
" in

request world5 "satisfiable(false)"
```