UNIVERSITÉ DE LORRAINE
FACULTÉ DE SCIENCE ET TECHNOLIGIE

MASTER 1 INFORMATIQUE

2025/2026

Projet de LMC

Algorithme d'unification **Martelli-Montanari**

Enseignant : **Didier Galmiche**

Auteurs :

CISSÉ Papa El Hadji Gormack

KEITA Mahamadou

24 novembre 2025

Table des matières

Introduction

Ce rapport présente l'implémentation en Prolog d'une variante de l'algorithme d'unification de Martelli-Montanari, telle que décrite dans le sujet du TP LMC. L'objectif principal est de résoudre un système d'équations d'unification P (représenté comme une liste $[S_1 ? = T_1, \ldots, S_n ? = T_n]$, avec l'opérateur ?= défini via op(20, xfy, ?=)) pour produire l'unificateur le plus général (mgu) ou signaler l'échec (*False*).

L'algorithme opère sur des paires (P ; S), où :

- P est l'ensemble des équations à résoudre.
- S est l'ensemble des substitutions résolues (bindings s = t).
- Les règles de transformation (Rename, Simplify, Expand, Orient, Decompose, Clash, Check) sont appliquées itérativement jusqu'à ce que P soit vide (succès avec mgu dans S) ou qu'un échec survienne (⊥).

Nous couvrons ici les Questions 1 et 2 du sujet :

- **Q1** : Implémentation de unifie(P) via les prédicats auxiliaires regle/2, occur_check/2, reduit/4.
- **Q2** : Paramétrage de unifie(P, S) avec des stratégies de choix d'équations (choix_premier, choix_pondere_1, choix_pondere_2, et une stratégie random supplémentaire).

Les choix d'implémentation privilégient la simplicité, l'efficacité et la fidélité à l'algorithme, en utilisant des prédicats Prolog standards comme functor/3, arg/3, select/3, findall/3 et sub_term/2 (avec adaptations pour éviter les bindings parasites). Les tests ont été réalisés sous SWI-Prolog 9.0.4.

Le code est modulaire, commenté, et testable interactivement. Des traces illustrent le fonctionnement sur les exemples du sujet.

Réponse à la question 1 : Implémentation du Prédicat Unifie(P)

Les Premiers Pas vers l'Unification

L'approche adoptée pour la Question 1 a consisté à développer itérativement les prédicats de détection et d'application des règles, en commençant par occur_check pour la vérification des occurrences, puis regle et reduit. Cela a permis de valider chaque composant isolément avant d'intégrer l'algorithme principal unifie(P), en s'appuyant sur les prédicats standards Prolog comme functor/3 et select/3 pour une implémentation efficace et lisible.

Exemple d'un prédicat *occur_check*

```
occur_check(V, T) :-
  V == T,
  !.
occur_check(V, T) :-
  var(V),
  nonvar(T),
  sub_term(Z, T),
  Z == V,
  !.
```

Listing 1 - Vérification d'occurrence dans un terme

Ce prédicat teste si V (variable) apparaît dans T (terme non variable), en utilisant sub_term/2 pour l'exploration récursive et == pour une égalité stricte sans unification parasite. Le cut (!) optimise en arrêtant dès la première occurrence détectée, évitant un backtracking inutile dans les cas positifs ce qui nous permet d'optimiser la recherche d'occurrence.

Exemple d'un Prédicat *regle*

```
regle(S ?= T, simplify) :-
  var(S),
  atomic(T),
  S \== T.

```

Listing 2 - Détection de la règle simplify

Le prédicat ***regle(E, R)*** analyse l'équation S ?= T pour identifier R. Les clauses sont séquencées pour prioriser les règles (ex. rename avant decompose), reflétant la logique de l'algorithme. Pour simplify, il vérifie que le côté gauche est une variable et le droit atomique.

Exemple d'un prédicat *reduit*

```
reduit(expand, S ?= T, P, Q) :-
  S=T,
  subst(S, T, P, Q ).

```

Listing 3 – Application de la réduction expand

reduit(R, E, P, Q) applique R à E pour transformer P en Q. Pour expand, il supprime E, substitue T pour X dans le reste de P (via substitute/4), et ajoute le binding X = T. Les cas d'échec (clash, check) retournent fail pour propager l'arrêt.

Unification

Le prédicat ***unifie(P)*** orchestre l'application itérative des règles sur (P ; []), en accumulant les substitutions dans un accumulateur S et en appliquant les bindings finaux.

```
unifie(P) :-
  unifie(P, choix_premier).

unifie(P, Strategie) :-
  unifie(P, [], Strategie).

unifie([],_, _) :-
  true.

unifie(P, Q, Strategie) :-
  choix(P, Q_rest, E, R, Strategie),
  echo('system : '), echo(P), ligne,
  echo(R), echo(': '), echo(E), ligne,
  reduit(R, E, Q_rest, S),
  unifie(S, Q, Strategie).

```

Listing 4 – Application d'unification d'un système d'équations

Cette version basique utilise un choix fixe (première équation), avec traces via echo. L'échec est propagé par fail en cas de clash/check.

Réponse à la question 2 : Implantation des stratégies de choix d'équation pour l'unification

L'optimisation de l'algorithme passe par une sélection intelligente des équations, impactant le nombre d'étapes et la robustesse face aux échecs. Nous avons généralisé ***unifie(P)*** en ***unifie(P, Strategie***), en implémentant ***choix(P, Q_rest, E, R, Strategie)*** pour quatre variantes, permettant une évaluation comparative sur des exemples variés.

Les stratégies couvrent :

- **Choix du premier** : Sélection systématique de la tête de P.
- **Choix pondéré 1** : Priorité aux règles critiques (clash/check=6, decompose=3).
- **Choix pondéré 2** : Variante favorisant simplify/rename=5.
- **Choix aléatoire** : Sélection non déterministe pour analyser la variance.

Exemple d'Exécution avec *Choix du Premier*

```
?- unifie([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_premier).
system : [f(_14052,_14054)?=f(g(_14058),h(a)),_14058?=f(_14054)]
decompose: f(_14052,_14054)?=f(g(_14058),h(a))
system : [_14052?=g(_14058),_14054?=h(a),_14058?=f(_14054)]
expand: _14052?=g(_14058)
system : [_14054?=h(a),_14058?=f(_14054)]
expand: _14054?=h(a)
system : [_14058?=f(h(a))]
expand: _14058?=f(h(a))
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

Listing 5 – Exemple d'exécution avec le choix de la première équation

Exemple d'Exécution avec *Choix Pondéré 1*

```
?- unifie([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_pondere_1).
system : [f(_2280,_2282)?=f(g(_2286),h(a)),_2286?=f(_2282)]
decompose: f(_2280,_2282)?=f(g(_2286),h(a))
system : [_2280?=g(_2286),_2282?=h(a),_2286?=f(_2282)]
expand: _2280?=g(_2286)
system : [_2282?=h(a),_2286?=f(_2282)]
expand: _2282?=h(a)
system : [_2286?=f(h(a))]
expand: _2286?=f(h(a))
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

Listing 6 : Exemple d'exécution avec le choix pondéré 1

Exemple d'Exécution avec *Choix Aléatoire*

```
?- unifie([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_random).
system : [f(_6678,_6680)?=f(g(_6684),h(a)),_6684?=f(_6680)]
decompose: f(_6678,_6680)?=f(g(_6684),h(a))
system : [_6678?=g(_6684),_6680?=h(a),_6684?=f(_6680)]
expand: _6678?=g(_6684)
system : [_6680?=h(a),_6684?=f(_6680)]
expand: _6680?=h(a)
system : [_22?=f(h(a))]
expand: _22?=f(h(a))
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

Listing 7 – Exemple d'exécution avec le choix aléatoire

Réponse à la question 3 : Implémentation de unif et de trace_unif

La Question 3 vise à contrôler les traces d'exécution via ***unif(P, S)*** (silencieux) et ***trace_unif(P, S)*** (avec affichage). Nous avons refactorisé les echo dans unifie pour utiliser le flag ***echo_on***, activé/désactivé par set_echo/clr_echo.

Exemple d'Exécution avec *Trace Activée*

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_premier).
system : [f(_3022,_3024)?=f(g(_3028),h(a)),_3028?=f(_3024)]
decompose: f(_3022,_3024)?=f(g(_3028),h(a))
system : [_3022?=g(_3028),_3024?=h(a),_3028?=f(_3024)]
expand: _3022?=g(_3028)
system : [_3024?=h(a),_3028?=f(_3024)]
expand: _3024?=h(a)
system : [_3028?=f(h(a))]
expand: _3028?=f(h(a))
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

Listing 8 – Exemple d'exécution avec trace

Exemple d'Exécution *Sans Trace*

```
?- unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_premier).
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

Listing 9 – Exemple d'exécution silencieuse

Conclusion

Ce projet nous a permis d'implémenter et d'analyser en profondeur l'algorithme d'unification de **Martelli–Montanari**, un élément central de la logique du premier ordre, de la programmation logique et des systèmes de preuve automatique. L'implémentation en Prolog a constitué un cadre particulièrement adapté pour explorer les mécanismes internes de l'unification : substitutions, structures de termes, règles de transformation, occur-check et propagation récursive des contraintes.

Sur le plan technique, nous avons construit progressivement un système d'unification robuste, fondé sur une architecture modulaire autour des prédicats occur_check/2, regle/2 et reduit/4. Cette structure nous a permis de gérer correctement les substitutions profondes, d'éviter les unifications parasites et de respecter fidèlement les règles de l'algorithme. La mise en œuvre d'un occur-check fiable, notamment via sub_term/2 et l'égalité stricte ==, s'est révélée essentielle pour garantir la conformité du comportement attendu.

Nous avons également étudié l'impact de différentes stratégies de choix d'équations sur l'exécution de l'algorithme. Les stratégies pondérées ont montré qu'un traitement prioritaire de certaines règles peut accélérer la détection des échecs, tandis que la stratégie aléatoire offre une variante utile pour explorer des chemins de résolution alternatifs. L'extension vers unif et trace_unif nous a en outre permis de disposer d'un outil flexible, capable d'exécuter l'unification de manière silencieuse ou détaillée.

Les tests réalisés sur un ensemble de 28 scénarios variés ont confirmé la robustesse et la fidélité de notre implémentation, tant pour les cas simples que pour les termes complexes, ainsi que pour la gestion correcte des échecs (clash, occur-check, conflits structurels). Ce travail a également mis en lumière plusieurs subtilités de Prolog, notamment la gestion des substitutions globales, la maîtrise de l'unification implicite et l'importance de l'ordre d'évaluation.

De manière générale, ce projet nous a permis de mieux comprendre le fonctionnement de l'unification et l'importance du choix des règles pour améliorer les performances. Il ouvre aussi plusieurs pistes intéressantes pour la suite, comme tester nos stratégies sur des exemples plus variés, essayer de nouvelles méthodes de sélection ou encore étendre l'unification à des cas plus avancés et l'intégrer dans un petit moteur de démonstration automatique.

Sources

Projet

```
:- op(20,xfy,?=).

% Prédicats d'affichage fournis

% set_echo: ce prédicat active l'affichage par le prédicat echo
set_echo :- assert(echo_on).

% clr_echo: ce prédicat inhibe l'affichage par le prédicat echo
clr_echo :- retractall(echo_on).

% echo(T): si le flag echo_on est positionné, echo(T) affiche le terme T
%         sinon, echo(T) réussit simplement en ne faisant rien.

echo(T) :- clause(echo_on, _), !, write(T) .
echo(_).

ligne :-
```

```
   clause(echo_on,_), !,
   nl.   % appel au nl standard
ligne.

% Question 1 : Implementation unifie(P)
   % Teste si la variable V apparaît dans le terme T
      occur_check(V, T) :- V == T,!.                % cas trivial
      occur_check(V, T) :- var(V), nonvar(T), sub_term(Z, T), Z == V,!.

%% --------------------------
% Détermine la règle de transformation R qui s'applique à l'équation E
      % Si S et T variables : rename si différente
         regle(S ?= T, rename) :- var(S), var(T), S \== T.

      % Si S et S variables  :  delete (x=x)
         regle(S ?= S, delete) :- var(S). % Pour x=x ou t=t
         regle(S ?= S, delete) :- atomic(S). % Pour x=x ou t=t

      % Si T constante et S variable : simplify
         regle( S ?= T, simplify) :- var(S),atomic(T),          S \== T.
      % Si T composé et S variable, sans occurrence :expand
         regle(S ?= T, expand) :- var(S), compound(T), \+ occur_check(S, T).

      % Si S non variables et T variable : orient
         %regle(S ?= T, orient) :- nonvar(S), var(T).
         regle(T ?= X, orient) :- var(X), \+ var(T).

      % Si même foncteur et même arité : decompose
         regle(S ?= T, decompose) :- compound(S), compound(T), functor(S, F, N),
functor(T, F, N).

      % Si les foncteur sont différents : clash
         regle(S ?= T, clash) :- compound(S), compound(T),
functor(S, F1, N1), functor(T, F2, N2), ( F1 \== F2 ; N1 \== N2 ).
      % Si variable = composé avec occurrence : check
         regle(S ?= T, check) :- var(S),compound(T), occur_check(S, T).

   % Transforme le système d'équations P en le système d'équations Q par application
de la règle de transformation R à l'équation E.

      % delete : on enlève l'équation
         reduit(delete, _ ?= _, P, P).       % select(X,Y,Z) supprime x dans y et mets le résultat
```

```
dans z

    % rename : on peut remplacer la variable S par T partout
       reduit(rename, S ?= T, P, Q) :- S=T, subst(S,T,P,Q).
    % simplify : idem que rename
       reduit(simplify, S ?= T, P, Q) :-  S=T, subst(S,T,P,Q).

       reduit(expand, S ?= T, P, Q) :- S=T, subst(S, T, P, Q ).

    % orient : échange S et T
       reduit(orient, T ?= S, P,[S ?= T | P]).

    % decompose : remplace S?=T par leurs sous-termes
       reduit(decompose, S ?= T, P, Q) :-
          S =.. [_|ArgsS],
          T =.. [_|ArgsT],
          pairs_to_eqs(ArgsS, ArgsT, EqList),
          append(EqList, P, Q).
```

```
% clash : on ne peut pas continuer
        reduit(clash, _, _, _) :-
            echo('Erreur : clash entre fonctions'), ligne,
            fail.

% check : variable occurrence détectée
      reduit(check, _, _, _) :-
        echo('Erreur : occur check échoué'), ligne,
        fail.
      % remplace profondément la variable X par T dans le terme InTerm -> OutTerm
        term_subst(X, T, InTerm, OutTerm) :-
            InTerm == X, !,              % même identificateur => remplacer
            OutTerm = T.
        term_subst(_, _, InTerm, InTerm) :-
            atomic(InTerm), !.           % atomes / constantes inchangés
        term_subst(_, _, InTerm, InTerm) :-
            var(InTerm), !.              % variable différente non remplacée
        term_subst(X, T, InTerm, OutTerm) :- % terme composé : on descend
            InTerm =.. [F|Args],
            maplist(term_subst(X, T), Args, Args2),
            OutTerm =.. [F|Args2].
      % remplace toutes les occurrences de X par T dans une liste de
        subst(_, _, [], []).
        subst(X, T, [H|Tail], [H2|Tail2]) :-
            H =.. [Op, A, B],
            term_subst(X, T, A, A2),
            term_subst(X, T, B, B2),
            H2 =.. [Op, A2, B2],
            subst(X, T, Tail, Tail2).
```

```
%Transforme deux listes en équations
    pairs_to_eqs([], [], []).
    pairs_to_eqs([H1|T1], [H2|T2], [H1 ?= H2|EqT]) :-
      pairs_to_eqs(T1, T2, EqT).

%unifie(P) :- set_echo,
    %           unifie(P, []).

    %unifie([],Q) :-
    %   echo(Q), ligne,
    %   echo('Yes').

    %unifie(P,Q) :-
    %   echo('system: '), echo(P), ligne,
    %   P = [E|Rest],  % Choisir la première équation E dans P
    %   (regle(E, R) ->
    %   echo(R), echo(': '), echo(E), ligne,
    %   reduit(R, E, Rest, S),
    %   unifie(S,Q);
    %   echo('check: '), echo(E), ligne, fail).

% Question 2 : Implantation des stratégies de choix d'équation pour l'unification
  unifie(P) :- unifie(P, choix_premier).

  %unifie(P, Strategie) :- set_echo, unifie(P, [], Strategie).
  unifie(P, Strategie) :- unifie(P, [], Strategie).

  unifie([],_, _) :-true.

  unifie(P, Q, Strategie) :-
    choix(P, Q_rest, E, R, Strategie),
    echo('system : '), echo(P), ligne,
    echo(R), echo(': '), echo(E), ligne,
    reduit(R, E, Q_rest, S),
    unifie(S, Q, Strategie).
```

```
% Choix systématique de la première équation
   choix([E|Q_rest], Q_rest, E, R, choix_premier) :-
      regle(E, R)->true;echo('check: '), echo(E), ligne,fail.                    % R = règle
applicable à E

%implémentation des 2 choix pondérée    choix(P, Q_rest, E, R, choix_pondere_1) :-
choix_pondere(P, Q_rest, E, R, 1).
   choix(P, Q_rest, E, R, choix_pondere_2) :- choix_pondere(P, Q_rest, E, R, 2).
   choix(P, Q_rest, E, R, choix_random) :- choix_random(P, Q_rest, E, R).

   % Implémentation d'une stratégie pondérée
      % Plus le nombre est grand, plus la règle est prioritaire
         poids(1,clash, 6).
         poids(1,check, 6).
         poids(1,rename, 5).
         poids(1,simplify, 5).
         poids(1,orient, 4).
         poids(1,decompose, 3).
         poids(1,expand, 2).
         poids(1,delete, 1).

      % Deuxieme stratégie de choix pondérer Plus le nombre est grand, plus la règle est
prioritaire
         poids(2,simplify, 5).
         poids(2,rename, 5).
         poids(2,decompose, 3).
         poids(2,orient, 4).
         poids(2,expand, 2).
         poids(2,clash, 6).
         poids(2,check, 6).
         poids(2,delete, 1).

      % Prédicat de choix pondéré
         choix_pondere(P, Q_rest, E, R,I) :-
            % Pour chaque équation, on calcul la règle et son poids, la fonction génère
toutes les combinaisons
            findall([Eq, Regle, Pds], (member(Eq, P),
            regle(Eq, Regle)->true;echo('check: '), echo(Eq), ligne, fail,
            poids(I, Règle, Pds)), L),L \= [],
            % On prend l'équation de poids maximal
            poids_max(L, [E, R, _]),
            select(E, P, Q_rest).    % Q_rest quotient le reste du systeme sans l'équation
```

```
choisie

        % poids_max(Liste, EleMax), renvoie dans EleMax l'entrée de L avec le plus grand
poids
        poids_max([H], H) :- !.  % cas de base, un seul élément alors c'est le max
        poids_max([[Eq,R,Pd]|Reste], Max) :-
poids_max(Reste,nouveauMax),nouveauMax=[_,_,nouveauPd],( Pd >= nouveauPd -> Max
= [Eq,R,Pd] ; Max = nouveauMax ).


   %Autres stratégies possibles( on a implémenter une stratégie random qui choisis les
équation aléatoirement)



      %implémentation d'un choix ramdom
      %la fonction random_member nous permet de choisir une équation aléatoirement
dans P
      choix_random(P, Q_rest, E, R) :- random_member(E, P),
      regle(E, R)
      ,select(E, P, Q_rest).

% Question 3 : Implémentation de unif et de trace_unif

   % unif( P , S ) permet de faire l'unification sans afficher les traces des différents règles
utiliser
   unif(P, S) :-
      clr_echo,            % désactiver l'affichage
      unifie(P, S).         % appeler ta version existante
```

```
% trace_unif(P, S) permet de faire l'unification avec l'affichage des différentes règles
utiliser
   trace_unif(P, S) :-
      set_echo,              % activer l'affichage
      unifie(P, S).          % appeler ta version existante

% Un petit guide pour l'utilisateur
start :-
   nl,
   write('Bienvenue dans le système d\'unification Prolog !'), nl,
   write('Voici comment l\'utiliser :'), nl,
   write('1. Pour unifier sans afficher les étapes :'), nl,
   write('   ?- unif([equations], Strategie).'), nl,
   write('   (Si vous ne donnez pas de strategie, la strategie par défaut est
choix_premier)'), nl,

write('2. Pour unifier avec trace (affiche les règles appliquées) :'), nl,
   write('   ?- trace_unif([equations], Strategie).'), nl,
   write('3. Les stratégies de choix possibles sont : choix_premier, choix_pondere_1,
choix_pondere_2, choix_random'), nl,
   write('   Exemple avec strategie : ?- unif([equations], choix_pondere_1).'), nl,
   nl.



:- start.
```

Listing 10 - le fichier **tp_lmc.pl**

Tests

```
- [tp_lmc].  % charger ton code d'unification

% Fonction utilitaire pour tester et afficher le résultat
test_unification(NomTest, Systeme,  ResultatAttendu) :-
write('=== Test: '), write(NomTest), write(' ==='), nl,
   ( catch(unif(Systeme, choix_premier), _, fail) -> R1 = true ; R1 = false ),
   ( catch(unif(Systeme, choix_pondere_1), _, fail) -> R2 = true ; R2 = false ),
```

```
    ( catch(unif(Systeme, choix_pondere_2), _, fail) -> R3 = true ; R3 = false ),
    ( catch(unif(Systeme, choix_random), _, fail) -> R4 = true ; R4 = false ),    % on
considère vrai si toutes les stratégies réussissent    (R1, R2, R3, R4 -> Resultat = true ;
Resultat = false),
    ( Resultat == ResultatAttendu ->       write('Succès!'), nl
    ;   write('Échec! Résultat attendu : '), write(ResultatAttendu),nl
    ),
    nl.

% Liste de tests
run_all_tests :-
    % Tests simples
    test_unification('Test1', [X1 ?= a, Y1 ?= X1], true),

    % Tests composés
    test_unification('Test2', [f(X2,Y2) ?= f(g(Z2),h(a)), Z2 ?= f(Y2)], true),
    test_unification('Test3', [X3 ?= f(f(f(f(X3))))], false),
    test_unification('Test4', [X4 ?= f(g(h(X4, k(Y4)), Z4)), Y4 ?= f(X4)], false),
    test_unification('Test5', [X5 ?= f(Y5), Y5 ?= g(X5)], false),

    test_unification('Test6', [X6 ?= f(Y6), Y6 ?= g(Z6), Z6 ?= h(X6)], false),
    test_unification('Test7', [X7 ?= f(a, Y7), Y7 ?= f(b, Z7), Z7 ?= f(c, X7)], false),

    test_unification('Test8', [X8 ?= f(Y8, g(X8)), Y8 ?= h(X8, f(Y8))], false),
    test_unification('Test9', [X9 ?= f(g(X9, h(Y9))), Y9 ?= k(f(X9))], false),
```

```
  test_unification('Test10', [X10 ?= tree(L1, R1), Y10 ?= tree(L2, R2),
                        L1 ?= tree(a, b), L2 ?= tree(a, b),
                        R1 ?= tree(c, d), R2 ?= tree(c, d)], true),
  test_unification('Test11', [f(X11, g(Y11)) ?= f(a, g(b)), X11 ?= h(Z11), Y11 ?= h(W11)],
false),
  test_unification('Test12', [X12 ?= f(Y12), Y12 ?= g(Z12), Z12 ?= a, X12 ?= W12], true),

  test_unification('Test13', [X13 ?= f(Y13,Z13), Y13 ?= g(a), Z13 ?= h(b), X13 ?=
f(U13,V13)], true),
  test_unification('Test14', [f(X14, g(Y14)) ?= f(a, h(b)), X14 ?= Y14], false),
  test_unification('Test15', [X15 ?= f(a, b), X15 ?= f(Y15)], false),
test_unification('Test16', [f(X16, a) ?= f(g(Y16), Y16), Y16 ?= h(X16)], false),
  test_unification('Test17', [f(X17,Y17) ?= g(X17,Y17)], false),
  test_unification('Test18', [f(X18) ?= f(X18,Y18)], false),
  test_unification('Test19', [f(g(X19), h(Y19, a), Z19) ?= f(g(b), h(c, a), k(t))], true),

  test_unification('Test20', [X20 ?= Y20, Y20 ?= Z20, Z20 ?= h(f(X20))], false),
  test_unification('Test21', [X21 ?= f(Y21,Z21), Y21 ?= g(a), Z21 ?= h(Y21)], true),
  %test_unification('Test22', [f(a, X22) ?= Y22, Y22 ?= f(a, g(Z22))], true),
  test_unification('Test23', [X23 ?= f(Y23), Y23 ?= g(Z23, h(Z23)), Z23 ?= a], true),
  test_unification('Test24', [f(X24, Y24) ?= f(g(Y24), h(X24))], false),

  %test_unification('Test25', [f(X25, g(Y25), Z25) ?=
f(h(a), g(k(b)), m(T25)),
  %                      Y25 ?= k(b), Z25 ?= m(T25)], true),
  test_unification('Test26', [X26 ?= f(Y26,Z26), f(a,b) ?= X26, Z26 ?= b, Y26 ?= a], true),
  test_unification('Test27', [A27 ?= f(B27), B27 ?= g(C27), C27 ?= h(D27), D27 ?= A27],
false),

  test_unification('Test28', [f(X28, g(Y28), h(Z28,Z28)) ?= f(g(a), g(k(W28)), h(W28,
h(a))),
                        Z28 ?= h(a), Y28 ?= k(W28)], true).
```

Listing 11 : le fichier **test.pl**