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

%Pour enlever les warning de Singleton variables
:- style_check(-singleton).

%Definition echo

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







% PREDICATS

% rule(E,R) : détermine la règle de transformation R qui s applique à l équation E, par exemple, le but ?-
rule(f(a) ?= f(b),decompose) réussit.
% occur_check(V,T) : teste si la variable V apparaît dans le terme T.
% reduct(R,E,P,Q) : transforme le système d'équations P en le système d équations Q par application de la
règle de transformation R à l'équation E.







%////////////////////////////////////////////////////////////////////////////////////////////////////
////////////



/* Le predicat rule :
Détermine la règle de transformation R qui s applique à l équation E, par exemple, le but ?- rule(f(a) ?=
f(b),decompose) réussit. */


rule(X ?= Y, rename) :-
        var(X),
        var(Y),
        X = Y, !.



rule(X ?= Y, simplify) :-
        var(X),
        atomic(Y),
        !.



rule(X ?= Y, expand) :-
        compound(Y),
        var(X),
        occur_check(X,Y),
        !.


rule(X ?= Y, orient) :-
```

```
        not(var(X)),
        var(Y),
        !.



rule(X ?= Y, decompose) :-
        compound(X),
        compound(Y),
        functor(X,N,A),
        functor(Y,M,B),
        (M == N),
        (A == B),
        !.



rule(X ?= Y, clash) :-
        compound(X),
        compound(Y),
        functor(X,A,_),
        functor(Y,B,_),
        A \= B,
        echo("clash : "),
        echo(X ?= Y),
        nl,
        !.



rule(X ?= Y, clash) :-
        compound(X),
        compound(Y),
        functor(X,_,N),
        functor(Y,_,M),
        N \= M,
        echo("clash : "),
        echo(X ?= Y),
        nl,
        !.



rule(X ?= Y, occur_check) :-
        var(X),
        echo("occur check : "),
        echo(X ?= Y),
        nl,
        not(occur_check(X, Y)),
        fail.



rule(X ?= Y, clean) :-
        atomic(X),
        atomic(Y),
        X == Y,
        !.







%////////////////////////////////////////////////////////////////////////////////////////////////////
////////////



/* Le predicat occur_check :
Teste si la variable V apparaît dans le terme T. */


occur_check(V,T) :-
        var(V),
```

```
        compound(T),
        not(var_into_arg(V,T)).



% Si T = V, alors la variable apparait dans le terme T
var_into_arg(V,T) :-
        var(T),
        V == T.



% Si T composé de plusieurs arguments, on vérifie si V apparait dans un de ces arguments
var_into_arg(V,T) :-
        compound(T),
        functor(T,_,A),
        var_into_term(V,T,A).



% On parcout les arguments A de T pour vérifier si V apparait dans T.
var_into_term(V, T, A) :-
        A > 0,
        arg(A,T,X),
        var_into_arg(V,X).



% Cas d arret
var_into_term(V, T, A) :-
        A \= 1,
        plus(A, -1, Y),
        var_into_term(V,T,Y).







%////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////



/* Le predicat reduce
Transforme le système d'équations P en le système d'équations Q par application de la règle de
transformation R à l'équation E. */


reduce(rename, X ?= Y, P, Q) :-
        echo("rename : "),
        echo(X ?= Y), nl,
        Q = P,
        X = Y,
        !.



reduce(simplify, X ?= Y, P, Q) :-
        echo("simplify : "),
        echo(X ?= Y),
        nl,
        Q = P,
        X = Y,
        !.



reduce(expand, X ?= Y, P, Q) :-
        echo("expand : "),
        echo(X ?= Y),
        nl,
        Q = P,
        X = Y,
```

```
        !.



reduce(orient, X ?= Y, P, Q) :-
        echo("orient : "),
        echo(X ?= Y),
        nl,
        append(P, [Y ?=X], Q),
        !.



reduce(decompose, X ?= Y, P, Q) :-
        echo("decompose : "),
        echo(X ?= Y),
        nl,
        functor(X,_,A),
        decomposition(X,Y,A,R),
        append(R,P,Q),
        !.



reduce(clean, X ?= Y, P, Q) :-
        echo("clean : "),
        echo(X ?= Y),
        nl,
        Q = P,
        !.



% Décomposition des arguments d une fonction en une liste d équations
decomposition(X, Y, N, Q) :-
        N \= 1,
        plus(N, -1, M),
        decomposition(X, Y, M, P),
        arg(N, X, A),
        arg(N, Y, B),
        append([A ?= B],P, Q).



% Cas d arret
decomposition(X, Y, N, Q) :-
        N == 1,
        arg(N, X, A),
        arg(N, Y, B),
        Q = [A ?= B].




%////////////////////////////////////////////////////////////////////////////////////////////////////
////////////



% Différenciation des Strategies
%Premier ici
unifie([X|T], Strategie) :-
        Strategie == premier,
        choix_premier([X|T]).



%Pondere ici
unifie([X|T], Strategie) :-
        Strategie == pondere,
        choix_pondere([X|T]).



%TRAITER LE cas ou aucun choix n est bon
unifie([X|T], Strategie) :-
```

```
        Strategie \== pondere,
        Strategie \== premier,
        write('\nStratégie invalide'),
        readStrategie([X|T],Strategie,Trace).



/* Predicat unifie(P) :
où P est un système d'équations à résoudre représenté sous la forme d'une liste [S1 ?= T1,...,SN ?= TN].
*/









/************************************ STRATEGIE CHOIX PREMIER
************************************************/



% Unifie avec une stratégie de base : prendre les équations dans l ordre de lecture de gauche à droite.
choix_premier([X|T]) :-
        echo("system : "),
        echo([X|T]),
        nl,
        rule(X, R),
        reduce(R, X, T, Q),
        choix_premier(Q).


% Cas d arret
choix_premier([]) :-
        write('Système d\'equation unifiable.'),
        !.




/*********************************** STRATEGIE CHOIX PONDERE
*****************************************************/



%Définition des différents poids matérialisant les priorités entre les différentes opérations


weight(clash,5).
weight(check,5).
weight(rename,4).
weight(simplify,4).
weight(orient,3).
weight(decompose,2).
weight(expand,1).


% Unifie avec une stratégie de préférence d'équations en fonction de leur opération.
choix_pondere(X) :-
        echo("system : "),
        echo(X),
        echo('\n'),
        maxWeight(X, R, E),
        extract(X, E, Res),
        reduce(R, E, Res, Q),
        choix_pondere(Q).


%Cas d arrêt
choix_pondere([]) :-
        write('Système d\'equation unifiable.'),
        !.
```

```
%Si P1 >= P2 On cherche à récupérer celle qui à le poids le plus fort.
maxWeight([X,Y|P], R, E) :-
        rule(X,R1),
        weight(R1,P1),
        rule(Y,R2),
        weight(R2,P2),
        P1>=P2,
        !,
        maxWeight([X|P], R, E).


%Si P1 =< P2
maxWeight([X,Y|P], R, E) :-
        rule(X,R1),
        weight(R1,P1),
        rule(Y,R2),
        weight(R2,P2),
        P1=<P2,
        !,
        maxWeight([Y|P], R, E).


%Cas d arrêt
maxWeight([X], R, X) :-
        rule(X,R),
        !.


%Récupère la bonne équation à traiter.
extract([T|R],X,Res) :-
        X == T,
        Res = R,
        !.


extract([T|R],X,Res) :-
        X \== T,
        extract(R,X,Res).


%Cas d arrêt
extract([],_,[]) :-
        !.




%////////////////////////////////////////////////////////////////////////////////////////////////////
////////////




%ACTIVATION DE LA TRACE
trace_unif(P,Strategie) :-
        set_echo,
        unifie(P,Strategie).



%DESACTIVATION DE LA TRACE
unif(P,Strategie) :-
        clr_echo,
        unifie(P,Strategie).



%Traitement choix user trace == non
trace(SystEq,Strategie,Trace) :-
        Trace == oui,
        trace_unif(SystEq,Strategie).
```

```
%Traitement choix user trace == oui
trace(SystEq,Strategie,Trace) :-
        Trace == non,
        unif(SystEq,Strategie).



%MAUVAIS CHOIX
trace(SystEq,Strategie,Trace) :-
        Trace \== non,
        Trace \== oui,
        write('Choix de la trace invalide\n'),
        choixTrace(SystEq,Strategie,Trace).




/********************************** DEROULEMENT PRINCIPAL DU PROGRAMME
*****************************************************/


run :-
    write('Programme réalisé par Antoine Courtil et Simon Hajek'),
    write('\nAlgorithme d'unification de Martelli-Montanari vu avec M. Galmiche'),
    begin.




begin:-
        repeat,
        write('\n\nEcrire le système que vous voulez unifier, par exemple : [f(X,Y) ?= f(g(Z),h(a)), Z ?=
        f(Y)].\n\n'),
        write('>> Systeme d\'equation à unifier : '),
        read(SystEq),
        readStrategie(SystEq,Strategie,Trace),
        choixTrace(SystEq,Strategie,Trace),
        lancementAlgo(SystEq,Strategie,Trace),
        write('\n\n>> Recommencer ? oui | non '),
        read(Recommencer),
        (Recommencer == non),
        !.



readStrategie(SystEq,Strategie,Trace) :-
        repeat,
        write('\n\nQuelle stratégie voulez-vous utiliser ? ( \'premier.\' OU \'pondere.\')\n'),
        write('>> Stratégie : '),
        read(Strategie),
        (Strategie == premier ; Strategie == pondere),
        write(Strategie),
        !.



choixTrace(SystEq,Strategie,Trace) :-
        repeat,
        write('\n\nVoulez-vous activer la trace ? (Ecrire \'oui\' OU \'non\')\n'),
        write('>> Trace : '),
        read(Trace),
        (Trace == oui ; Trace == non),
        write(Trace),
        write('\n'),
        !.



lancementAlgo(SystEq,Strategie,Trace) :-
        trace(SystEq,Strategie,Trace).
```