Améliorons notre proposition en demandant que les trois points soient différents. Cela donne

```
triangle_v2(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1.
```

Ceci est très naturel : en plus des conditions précédentes, nous demandons que les trois points `P1`, `P2` et `P3` soient deux à deux différents.

Malheureusement,

```
?- triangle_v2(P1, P2, P3).
P1 = a, P2 = b, P3 = d
...
```

construit une solution formée d'un **triangle dont les points sont alignés**.

Améliorons notre proposition en demandant de plus que les trois segments soient différents. Cela donne

```
triangle_v3(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1,
    P1_P2 \= P2_P3,
    P2_P3 \= P1_P3,
    P1_P3 \= P1_P2.
```

Ceci est très naturel : en plus des conditions précédentes, nous demandons que les trois segments `P1_P2`, `P1_P3` et `P2_P3` soient deux à deux différents.

Malheureusement,

```
?- triangle_v3(P1, P2, P3).
P1 = a, P2 = b, P3 = c ;
...
P1 = b, P2 = a, P3 = c ;
...
```

construit des solutions qui représentent **le même triangle**. Ceci n'est pas acceptable pour la problématique de dénombrement.

Pour éviter les solutions qui correspondent à des mêmes triangles, adoptons la convention que nos points sont totalement ordonnés selon l'ordre alphabétique et que tout triangle est spécifié par ses trois points dans le bon ordre.

Par exemple, les trois points `a`, `d` et `g`, dans cet ordre, forment un triangle. À l'inverse, les trois points `d`, `g`, `a`, dans cet ordre, ne forment pas un triangle.

Soit `inf/2` le prédicat tel que `inf(P1, P2)` si le point `P1` est inférieur pour l'ordre alphabétique au point `P2`.

Pour le définir, utilisons le prédicat `suivant/2` tel que `suivant(P1, P2)` si le point `P1` est suivi par le point `P2` dans l'alphabet.

Voici leurs définitions :

```
suivant(a, b).
suivant(b, c).
suivant(c, d).
suivant(d, e).
suivant(e, f).
suivant(f, g).
suivant(g, h).
```

```
inf(P, P).
inf(P1, P2) :-
    suivant(P1, P3),
    inf(P3, P2).
```

Le second prédicat est récursif.

Voici notre prédicat pourvu de l'amélioration précédente, avec la requête de génération des solutions :

```
triangle_v4(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1,
    P1_P2 \= P2_P3,
    P2_P3 \= P1_P3,
    P1_P3 \= P1_P2,
    inf(P1, P2),
    inf(P2, P3).
```

```
?- triangle_v4(P1, P2, P3).
P1 = a, P2 = b, P3 = c ;
P1 = a, P2 = b, P3 = f ;
P1 = a, P2 = b, P3 = e ;
P1 = a, P2 = b, P3 = g ;
P1 = a, P2 = d, P3 = g ;
P1 = b, P2 = d, P3 = g ;
P1 = a, P2 = c, P3 = f ;
P1 = a, P2 = c, P3 = d ;
P1 = a, P2 = c, P3 = g ;
P1 = a, P2 = e, P3 = g ;
P1 = c, P2 = e, P3 = g ;
P1 = b, P2 = c, P3 = e ;
P1 = b, P2 = c, P3 = d ;
P1 = b, P2 = c, P3 = g ;
P1 = b, P2 = f, P3 = g ;
P1 = c, P2 = f, P3 = g ;
false.
```

La réponse au problème initial est donc 16.

Exercice : expérimenter avec d'autres figures géométriques (en mettant à jour `segment_point/2` et `suivant/2` adéquatement).

Nous allons programmer ici une petite bibliothèque de manipulation des **entiers naturels**.

Nous allons considérer l'atome `z` qui représente l'entier 0 et l'atome `s` qui permet de construire les autres entiers comme successeurs itérés de 0.

Exemples

- Le terme `s(z)` représente l'entier 1.
- Le terme `s(s(z))` représente l'entier 2.
- Le terme `s(s(s(z)))` représente l'entier 4.

Soit `entier/1` le prédicat tel que `entier(N)` si `N` est un entier. Il se définit par

```
entier(z).
entier(s(N)) :-
    entier(N).
```

Exemples

```
?- entier(s(s(z))).
true.
```

```
?- entier(miaou).
false.
```

```
?- entier(s(s(miaou))).
false.
```

Soit `inf/2` le prédicat tel que `inf(N1, N2)` si l'entier `N1` **est inférieur** à l'entier `N2`. Il se définit par

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

Le `_` se comporte comme un **joker** (mettre une variable ici ferait qu'elle serait non utilisée dans le corps de la clause, ce qui provoquerait un avertissement de « variable singleton »).

Exemples

```
?- inf(z, s(s(z))).
true.
```

```
?- inf(s(z), s(z)).
true.
```

```
?- inf(miaou, s(z)).
false.
```

De façon similaire, nous pouvons définir le prédicat `infs/2` tel que `infs(N1, N2)` si `N1` est strictement inférieur à `N2`.

Il est aussi possible de définir les prédicats duaux `sup/2` et `sups/2`.

Exercice : implanter ces trois prédicats supplémentaires.

Une problématique se pose lorsque nous souhaitons **manipuler des nombres concrets** : leur construction comme nombres de Peano est de la taille du nombre lui-même, ce qui peut être peu maniable.

Pour répondre à cela, il est intéressant d'avoir des prédicats `entier_n/1` tels que `entier_n(N)` si `N` est l'entier `n`. Nous pouvons alors écrire

```
entier_n(N), t1, ..., tk
```

où `t1, ..., tk` est une conjonction de termes. La partie `entier_n(N)` fait que la variable `N` est spécialisée à la valeur qui rend `entier_n(N)` vrai, et donc à la valeur `n` dans `t1, ..., tk`.

Une implantation possible des premiers prédicats de cette forme est

```
entier_0(z).
entier_1(s(X)) :- entier_0(X).
```

```
entier_2(s(X)) :- entier_1(X).
entier_3(s(X)) :- entier_2(X).
```

```
entier_4(s(X)) :- entier_3(X).
...
```

Exemples

```
?- entier_1(N1), entier_3(N2), inf(N1, N2).
N1 = s(z), N2 = s(s(s(z))).
```

Les deux premières parties de la requête font que `N1` (resp. `N2`), devant vérifier le prédicat `entier_1/1` (resp. `entier_2/1`), doit se spécialiser en `s(z)` (resp. `s(s(z))`). Ainsi, dans `inf(N1, N2)` les variables `N1` et `N2` sont spécialisées en les valeurs voulues.

Soit `add/3` le prédicat tel que `add(N1, N2, N3)` si l'entier `N1` **additionné** à l'entier `N2` est égal à l'entier `N3`. Il se définit par

```
add(z, N, N) :-
    entier(N).
add(s(N1), N2, s(N3)) :-
    add(N1, N2, N3).
```

Exemple

```
?- entier_4(N1), entier_2(N2), add(N1, N2, X).
N1 = s(s(s(s(z)))), N2 = s(s(z)), X = s(s(s(s(s(s(z)))))).
```

Le prédicat `add/3` est utilisé ici pour calculer le résultat d'une addition.

Exemple

```
?- entier_3(N), add(X1, X2, N).
N = X2, X2 = s(s(z)), X1 = z ;
N = s(s(s(z))), X1 = s(z), X2 = s(s(z)) ;
N = s(s(s(z))), X1 = s(s(z)), X2 = s(z) ;
N = X1, X1 = s(s(s(z))), X2 = z ;
false.
```

Le prédicat `add/3` est utilisé différemment : comme il n'y a pas de notion d'entrée et de sortie pour un prédicat, nous spécialisons son dernier argument et laissons les deux 1^{ers} variables.
Ceci permet de rechercher toutes les manières de décomposer 3 comme somme de deux entiers naturels.

Soit `mul/3` le prédicat tel que `mul(N1, N2, N3)` si l'entier `N1` **multiplié** à l'entier `N2` est égal à l'entier `N3`. Il se définit par

```
mul(z, N, z) :-
    entier(N).
mul(s(N1), N2, N3) :-
    mul(N1, N2, N4),
    add(N4, N2, N3).
```

Exemple

```
?- entier_3(N1), entier_2(N2), mul(N1, N2, R).
N1 = s(s(s(z))), N2 = s(s(z)), R = s(s(s(s(s(z))))).
```

Le prédicat `mul/3` est utilisé ici pour calculer le résultat d'une multiplication.

Exemple

```
?- entier_6(N), inf(X1, N), inf(X2, N), mul(X1, X2, N).
N = X2, X2 = s(s(s(s(s(z))))), X1 = s(z) ;
N = s(s(s(s(s(z))))), X1 = s(s(z)), X2 = s(s(s(z))) ;
N = s(s(s(s(s(z))))), X1 = s(s(s(z))), X2 = s(s(z)) ;
N = X1, X1 = s(s(s(s(s(z))))), X2 = s(z) ;
false.
```

Ceci recherche les factorisations de 4.
Dans la requête, nous forçons `X1` et `X2` à être inférieurs à `N`.
Exercice : observer ce qu'il se passe sans ces contraintes.

Soit `fact/2` le prédicat tel que `fact(N1, N2)` si la **factorielle** de l'entier `N1` est l'entier `N2`. Il se définit par

```
fact(z, s(z)).
fact(s(N1), N2) :-
    fact(N1, N3),
    mul(s(N1), N3, N2).
```

Exemple

```
?- entier_3(N), fact(N, X).
N = s(s(s(z))), X = s(s(s(s(s(z))))).
```

Le prédicat `fact/2` est utilisé ici pour calculer le résultat d'une factorielle.

Exemple

```
?- entier_5(N), inf(X, N), fact(X, N).
false.

?- entier_6(N), inf(X, N), fact(X, N).
N = s(s(s(s(s(z))))), X = s(s(s(z))) ;
false.
```

Ceci recherche les antécédents de 5 et de 6 pour la factorielle.
Dans les requête, nous forçons `X` à être inférieur à `N`, pour les mêmes raisons que dans le 2e exemple de la page précédente.

Soit `fibo/2` le prédicat tel que `fibo(N1, N2)` si le terme en position `N1` de la **suite de Fibonacci** est l'entier `N2`. Il se définit par

```
fibo(z, s(z)).
fibo(s(z), s(z)).
fibo(s(s(N1)), N2) :-
    fibo(s(N1), N3),
    fibo(N1, N4),
    add(N4, N3, N2).
```

Exemple

```
entier_5(N), fibo(N, X).
N = s(s(s(s(s(z))))), X = s(s(s(s(s(s(s(s)))))))).
```

Le prédicat `fibo/2` est utilisé ici pour calculer le résultat de la fonction de Fibonacci.

Exemple

```
?- entier_6(N), inf(X, N), fibo(X, N).
false.

?- entier_5(N), inf(X, N), fibo(X, N).
N = s(s(s(s(z)))))), X = s(s(s(z)))) ;
false.
```

Ceci recherche les antécédents de 5 et de 6 pour la fonction Fibonacci.
Dans les requête, nous forçons `X` à être inférieur à `N`, pour les mêmes raisons que dans le 2e exemple de la page précédente.

4.2. Bases théoriques

4.2.1. Unification

Une pièce fondamentale intervenant dans la **résolution** des requêtes est l'**unification de termes**.

Intuitivement, *unifier* deux termes t_1 et t_2 consiste à trouver une manière de remplacer simultanément les variables de t_1 et de t_2 par des termes de sorte à les rendre égaux.

Exemple

Considérons les deux termes $t_1 := \mathrm{add}(X, Y, \mathrm{z})$ et $t_2 := \mathrm{add}(\mathrm{s}(\mathrm{z}), Z, T)$.

Pour les rendre égaux, il faut identifier

- X et $\mathrm{s}(\mathrm{z})$;
- Y et Z ;
- z et T.

En remplaçant dans t_1 et t_2, X par $\mathrm{s}(\mathrm{z})$, Y par Z et T par z, nous obtenons le même terme $t_2 := \mathrm{add}(\mathrm{s}(\mathrm{z}), Z, \mathrm{z})$.

Exemple

Considérons les deux termes $t_1 := \mathrm{inf}(\mathrm{z}, \mathrm{s}(X))$ et $t_2 := \mathrm{inf}(\mathrm{s}(Y), Z)$.

Ces deux termes ne sont pas unifiables car il faudrait identifier z et $\mathrm{s}(X)$, ce qui est impossible.

En effet, ces deux sous-termes ont comme racines des atomes différents.

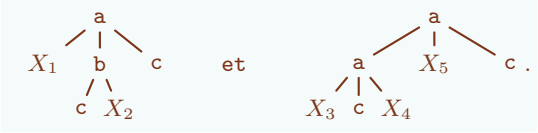
Ceci revient à résoudre une **équation sur les termes** dont les **variables sont les inconnues**.

Il est possible de visualiser comment deux termes peuvent s'unifier en tentant de superposer leurs représentations sous forme d'arbres.

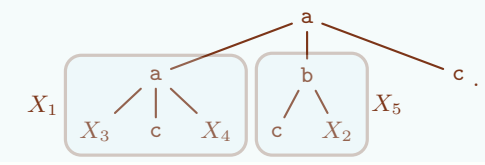
Il faut que tout atome se superpose soit sur une variable, soit sur un atome identique.

Exemple

Soient les termes $\mathrm{t}_1 := \mathrm{a}(X_1, \mathrm{b}(\mathrm{c}, X_2), \mathrm{c})$ et $\mathrm{t}_2 := \mathrm{a}(\mathrm{a}(X_3, \mathrm{c}, X_4), X_5, \mathrm{c})$.
Leurs représentations graphiques respectives sont



La superposition de ces termes donne



Les termes t_1 et t_2 sont donc unifiables, car, en remplaçant dans ces deux termes X_1 par $\mathrm{a}(X_3, \mathrm{c}, X_4)$ et X_5 par $\mathrm{b}(\mathrm{c}, X_2)$, nous obtenons le même terme.

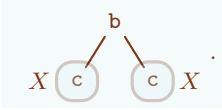
L'exemple précédent était facile à traiter car les deux termes à unifier ne possédaient pas de variable en commun.

Voici quelques exemples où cela arrive.

Exemple

Soient les termes $\mathrm{t}_1 := \mathrm{b}(X, \mathrm{c})$ et $\mathrm{t}_2 := \mathrm{b}(\mathrm{c}, X)$.
Leurs représentations graphiques respectives sont

La superposition de ces termes donne

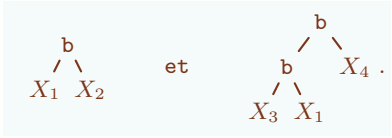


Les termes t_1 et t_2 sont donc unifiables, car, en remplaçant dans ces deux termes X par c et X par c (cette répétition est volontairement mentionnée), nous obtenons le même terme.

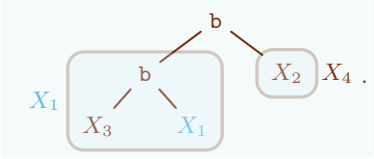
Exemple

Soient les termes $t_1 := b(X_1, X_2)$ et $t_2 := b(b(X_3, X_1), X_4)$.

Leurs représentations graphiques respectives sont



La superposition de ces termes donne



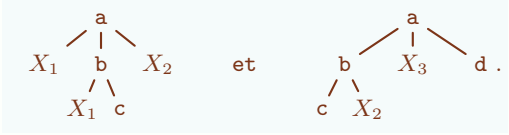
Pour unifier t_1 et t_2, il faut donc unifier X_1 et $b(X_3, X_1)$. Comme ce dernier terme contient la variable X_1, ceci est impossible.

Les termes t_1 et t_2 ne sont donc pas unifiables.

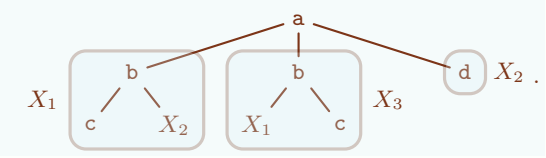
Voici un exemple un peu plus complexe.

Exemple

Soient les termes $t_1 := a(X_1, b(X_1, c), X_2)$ et $t_2 := a(b(c, X_2), X_3, d)$.
Leurs représentations graphiques respectives sont



La superposition de ces termes donne



Pour unifier t_1 et t_2, il faut donc unifier

- X_1 avec $b(c, X_2)$;
- X_3 avec $b(X_1, c)$;
- X_2 avec d.

En simplifiant, ceci revient à unifier

- X_1 avec $b(c, d)$;
- X_3 avec $b(b(c, d), c)$;
- X_2 avec d.

Les termes t_1 et t_2 sont donc unifiables.

Mettons un peu de formalisme à tout cela.

Un *substitution* est une fonction σ dont le domaine est l'ensemble des variables et le codomaine est l'ensemble des termes.

Si t est un terme, l'*application* de σ à t est le terme $t \cdot \sigma$ obtenu en remplaçant simultanément, pour toute variable X, toutes les occurrences de X dans t par $\sigma(X)$.

Exemples

- Soit σ la substitution telle que $\sigma(X) = g(Y)$. Nous avons $f(X) \cdot \sigma = f(g(Y))$.
- Soit σ la substitution telle que $\sigma(X_1) = b(d, d)$, $\sigma(X_2) = a(c, X_4, X_4)$ et $\sigma(X_3) = X_4$. Nous avons

$$a(X_1, X_2, b(X_1, X_3)) \cdot \sigma = a(b(d, d), a(c, X_4, X_4), b(b(d, d), X_4)).$$

Une substitution σ *unifie* deux termes t_1 et t_2 si $t_1 \cdot \sigma = t_2 \cdot \sigma$.

Exemple

Les termes $b(X, c)$ et $b(c, X)$ sont unifiés par toute substitution σ telle que $\sigma(X) = c$.

La notation pratique

$$\sigma = \{X_1 \mapsto \mathrm{t}_1, \ldots, X_n \mapsto \mathrm{t}_n\}$$

signifie que σ est la substitution vérifiant

$$\sigma(X) = \begin{cases} \mathrm{t} & \text{si } X \mapsto \mathrm{t} \in \sigma, \\ X & \text{sinon.} \end{cases}$$

Exemple

La définition

$$\sigma := \{X_1 \mapsto \mathrm{b}(X_1, X_2), \quad X_2 \mapsto \mathrm{d}, \quad X_3 \mapsto \mathrm{a}(X_4, X_5, \mathrm{b}(\mathrm{d}, X_2))\},$$

spécifie la substitution σ vérifiant $\sigma(X_1) = \mathrm{b}(X_1, X_2)$, $\sigma(X_2) = \mathrm{d}$, $\sigma(X_3) = \mathrm{a}(X_4, X_5, \mathrm{b}(\mathrm{d}, X_2))$ et $\sigma(X) = X$ pour toute variable X différente de X_1, X_2 et X_3.

La *substitution identité* ι est définie par $\iota := \emptyset$. Ainsi, pour toute variable X, $\iota(X) = X$.

La *composition* de deux substitutions σ_1 et σ_2 est la substitution $\sigma_1 \circ \sigma_2$ qui est telle que, pour toute variable X,

$$(\sigma_1 \circ \sigma_2)(X) = (X \cdot \sigma_2) \cdot \sigma_1.$$

Ainsi, l'image de la variable X par $\sigma_1 \circ \sigma_2$ se calcule en considérant le terme obtenu en appliquant σ_2 à X, puis en considérant le terme obtenu en appliquant σ_1 sur ce dernier.

Exemple

Soient les deux substitutions

$$\sigma_1 := \{X_2 \mapsto \mathtt{a}(X_1, X_2, \mathtt{d}),\ \ X_3 \mapsto \mathtt{b}(X_1, X_2)\}$$

et

$$\sigma_2 := \{X_1 \mapsto \mathtt{b}(X_2, X_3),\ \ X_2 \mapsto \mathtt{d}\}.$$

Nous avons

$$\sigma_1 \circ \sigma_2 = \{X_1 \mapsto \mathtt{b}(\mathtt{a}(X_1, X_2, \mathtt{d}), \mathtt{b}(X_1, X_2)),\ \ X_2 \mapsto \mathtt{d},\ \ X_3 \mapsto \mathtt{b}(X_1, X_2)\}.$$

Une substitution σ_1 est *plus générale* qu'une substitution σ_2 s'il existe une substitution σ_3 telle que $\sigma_2 = \sigma_3 \circ \sigma_1$. Nous disons aussi dans ce cas que σ_2 est *plus spécifique* que σ_1.

Exemple

Soient les substitutions

$$\sigma_1 := \{X_1 \mapsto X_2,\ X_2 \mapsto \mathsf{c}(X_3)\} \quad \text{et} \quad \sigma_2 := \{X_1 \mapsto \mathsf{c}(\mathsf{d}),\ X_2 \mapsto \mathsf{c}(\mathsf{e}),\ X_3 \mapsto \mathsf{e}\}.$$

En posant

$$\sigma_3 := \{X_2 \mapsto \mathsf{c}(\mathsf{d}),\ X_3 \mapsto \mathsf{e}\},$$

nous avons $\sigma_2 = \sigma_3 \circ \sigma_1$. Ainsi, σ_1 est plus générale que σ_2.

Cette notion est importante car pour **unifier** deux termes, nous cherchons la **substitution la plus générale** possible qui fait l'affaire.

Exemple

Soient $t_1 := \mathsf{b}(X_1, \mathsf{d})$ et $t_2 := \mathsf{b}(X_2, X_3)$. Nous pouvons vérifier facilement que les substitutions

$$\sigma_1 := \{X_1 \mapsto X_2,\ X_3 \mapsto \mathsf{d}\} \quad \text{et} \quad \sigma_2 := \{X_1 \mapsto \mathsf{d},\ X_2 \mapsto \mathsf{d},\ X_3 \mapsto \mathsf{d}\}$$

unifient t_1 et t_2.

Cependant, comme σ_1 est plus générale que σ_2, la substitution σ_1 est préférable.

L'**algorithme d'unification** se décrit ainsi.

Il prend deux termes t_1 et t_2 en entrée et renvoie la substitution la plus générale qui unifie t_1 et t_2 si et seulement si ces deux termes sont unifiables.

```
Fonction unifier(t1, t2)
    Si t1 et t2 sont tous deux égaux à la même variable X
        Renvoyer ι
    Sinon si (t1 est une variable qui apparaît dans t2) ou (t2 est une variable qui apparaît dans t1)
        Renvoyer Échec
    Sinon si t1 est une variable X
        Renvoyer {X ↦ t2}
    Sinon si t2 est une variable X
        Renvoyer {X ↦ t1}
    Sinon si les racines de t1 et t2 sont des atomes différents
        Renvoyer Échec
    Sinon
        Soit σ := ι
        Pour chaque couple (s1,s2) de fils en mêmes positions respectivement de t1 et t2
            Soit σ := unifier(s1 · σ, s2 · σ) ∘ σ
        Renvoyer σ
    Fin
Fin
```

Exemple

Soient les termes $t_1 := a(e(X_1), X_1, X_2)$ et $t_2 := a(X_2, c, e(X_3))$.

1. L'appel $\texttt{unifier}(t_1, t_2)$ déclenche le tout dernier cas.
2. Initialement, $\sigma := \iota$.
3. Ensuite, nous considérons les fils en 1res positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(e(X_1) \cdot \sigma, X_2 \cdot \sigma) \circ \sigma = \texttt{unifier}(e(X_1), X_2) = \{X_2 \mapsto e(X_1)\}.$$

4. Ensuite, nous considérons les fils en 2es positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(X_1 \cdot \sigma, c \cdot \sigma) \circ \sigma = \texttt{unifier}(X_1, c) \circ \sigma = \{X_1 \mapsto c\} \circ \sigma = \{X_1 \mapsto c,\ X_2 \mapsto e(c)\}.$$

5. Ensuite, nous considérons les fils en 3es positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(X_2 \cdot \sigma, e(X_3) \cdot \sigma) \circ \sigma = \texttt{unifier}(e(c), e(X_3)) \circ \sigma = \{X_3 \mapsto c\} \circ \sigma$$
$$= \{X_1 \mapsto c,\ X_2 \mapsto e(c),\ X_3 \mapsto c\}.$$

Cette substitution σ est le résultat de l'appel initial.

Exercice : appliquer l'algorithme sur les termes $t_1 := a(e(X_1), X_1, X_1)$ et $t_2 := a(X_2, c, e(X_3))$.

En PROLOG, il est possible de **demander explicitement une unification** via l'opérateur `=`.

Exemples

Voici quelques requêtes, directement soumises à SWI-PROLOG :

```
?- a(X, Y) = a(X, t).
Y = t.
```

```
?- a(X, Y) = a(Y, t).
X = Y, Y = t.
```

```
?- a(X, Y) = b(X, Y).
false.
```

```
?- a(X, a(Z, Z)) = a(Y, Y).
X = Y, Y = a(Z, Z).
```

Exemple

Voici deux définitions équivalentes du prédicat `inf/2` sur les entiers de Peano :

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

```
inf(N1, _) :-
    N1 = z.
inf(N1, N2) :-
    N1 = s(P1),
    N2 = s(P2),
    inf(P1, P2).
```

Dans la version de droite, l'unification est explicite. Les deux clauses du prédicat acceptent des paramètres génériques `N1` et `N2` qui sont contraints par la suite.

Dans la version de gauche, les deux clauses possèdent des arguments qui sont d'emblée spécialisés en la forme souhaitée. Cette version est en général bien préférable.

4.2.2. Résolution

Une *clause de Horn* est une formule de la forme

$$(H_1 \wedge H_2 \wedge \cdots \wedge H_n) \rightarrow C$$

où H_1, H_2, ..., H_n et C sont des termes.

Exemples

Quelques clauses de Horn :

- $(\texttt{divise}(2, X) \wedge \texttt{divise}(3, X)) \rightarrow \texttt{divise}(6, X)$;
- $(\texttt{fainéant}(Chat) \wedge \texttt{gourmand}(Chat) \wedge \texttt{jeune}(Chat)) \rightarrow \texttt{gros}(Chat)$.

Les H_i sont les *hypothèses* (elles forment le *corps* de la clause) et C est la *conclusion* (ou encore *tête* de la clause).

Cette formule explique le fait que **si** chaque H_i est vraie, **alors** C est vraie.

La véracité de chacune des hypothèses est une **condition suffisante** pour la conclusion.

Attention : les hypothèses ne sont pas forcément des conditions nécessaires à la conclusion.

Lorsque $n = 0$, les hypothèses étant vides, C est vraie **inconditionnellement**. Dans ce cas, la clause de Horn exprime un **fait**.

Soit $q := q_1 \wedge \cdots \wedge q_n$ une requête appelée *but*. **Résoudre** ce but signifie calculer toutes les substitutions σ qui sont telles que pour tout $1 \leqslant i \leqslant n$, tous les $q_i \cdot \sigma$ sont vrais simultanément.

Pour cela, PROLOG utilise l'algorithme de *résolution linéaire sélective de clauses définies* (SLD). Expliquons-le pas à pas.

L'algorithme part du but q et tente de **justifier** chaque sous-but q_i en commençant par q_1.

Pour cela, il cherche dans le programme la 1re clause `t :- t1, ..., tk` telle que q_1 et `t` sont unifiables. Il s'agit de la clause *sélectionnée*.

Exemple

Considérons le programme

```
parent(marvin, randy).
parent(marvin, jimbo).
parent(randy, shelly).
parent(randy, stan).
grand_parent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Considérons le but à résoudre

```
grand_parent(marvin, X).
```

Ce but est tout d'abord utilisé pour être unifié avec les quatre 1res clauses, ce qui échoue.

Ensuite, vient la 5e clause où une unification entre `grand_parent(marvin, X)` et `grand_parent(X1, Y1)` (obtenue en renommant ses variables de manière unique) est tentée.

La substitution résultante est $\sigma := \{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X\}$.

Une fois cette sélection effectuée, l'algorithme **remplace** $q_1, q_2, \ldots, q_n$ par

$$\texttt{t1} \cdot \sigma, \ldots, \texttt{tk} \cdot \sigma, \quad q_2 \cdot \sigma, \ldots, q_n \cdot \sigma$$

où σ est la substitution calculée par l'unification de q_1 et de `t`.

Exemple

Depuis l'exemple précédent, le but `grand_parent(marvin, X)` est remplacé par `parent(marvin, Z), parent(Z, X)`.

Ceci donne un nouveau but, qu'il faut continuer à résoudre de la même manière. Nous obtiendrons après la sélection d'une clause une substitution σ' par l'unification du 1er sous-but et de la tête de la clause. La **substitution courante** deviendra $\sigma' \circ \sigma$.

Exemple

Depuis l'exemple précédent, le but est `parent(marvin, Z), parent(Z, X)` et la substitution en cours de calcul est $\sigma := \left\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X\right\}$.
La clause sélectionnée (qui est un fait ici) est `parent(marvin, randy).` et la substitution calculée par l'unification de `parent(marvin, Z)` avec la tête de ce fait est $\sigma' = \{Z \mapsto \texttt{randy}\}$.
La substitution courante est $\left\{X^1 \mapsto \texttt{marvin}, Y^1 \mapsto X, Z \mapsto \texttt{randy}\right\}$ et le but courant est `parent(randy, X)`.

Les étapes de sélection, unification, et mise à jour du but et de la substitution courante sont poursuivies jusqu'à obtenir un **but vide**, ce qui signifie un succès dans la requête initiale. La substitution courante est alors une solution à la requête initiale.

Exemple

Depuis l'exemple précédent, nous avons `parent(randy, X)` comme but courant et $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X,\ Z \mapsto \texttt{randy}\}$ comme substitution courante.
La clause sélectionnée est `parent(randy, shelly).` et la substitution calculée par l'unification de `parent(randy, X)` avec la tête de cette clause est $\{X \mapsto \texttt{shelly}\}$.
La nouvelle substitution courante est $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto \texttt{shelly},\ Z \mapsto \texttt{randy},\ X \mapsto \texttt{shelly}\}$ et le nouveau but est vide.
Ainsi, cette dernière substitution est une solution au but initial.

Lorsqu'une substitution est un résultat, seules les **variables qui apparaissent dans le but initial** sont mentionnées.

Exemple

Depuis l'exemple précédent, $\{X \mapsto \texttt{shelly}\}$ est une solution à la requête initiale `grand_parent(marvin, X)`.

Une fois qu'une solution est trouvée, l'algorithme effectue un **retour arrière** au dernier endroit où il a sélectionné une clause de sorte à vérifier si une autre peut être sélectionnée.

Exemple

Depuis l'exemple précédent, depuis le but courant `parent(randy, X)` et la substitution courante $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X,\ Z \mapsto \texttt{randy}\}$, la prochaine clause sélectionnée est `parent(randy, stan)`.

La substitution calculée par l'unification est $\{X \mapsto \texttt{stan}\}$.

La nouvelle substitution courante est $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto \texttt{stan},\ Z \mapsto \texttt{randy},\ X \mapsto \texttt{stan}\}$ et le nouveau but est vide.

Ainsi, $\{X \mapsto \texttt{stan}\}$ est une solution à la requête initiale `grand_parent(marvin, X)`.

L'algorithme procède en continuant de cette manière pour **rechercher toutes les solutions**.

Lorsque aucune clause ne peut être sélectionnée de sorte que sa tête puisse être unifiée avec le 1er sous-but, une **situation d'échec** est rencontrée.

La recherche continue ensuite par retour arrière s'il reste des clauses à considérer jusqu'à épuisement.

L'**algorithme de résolution** se décrit ainsi.

Il prend une suite de clauses P et un but $q_1 \wedge \cdots \wedge q_n$ en entrée, et renvoie l'ensemble des solutions de ce but.

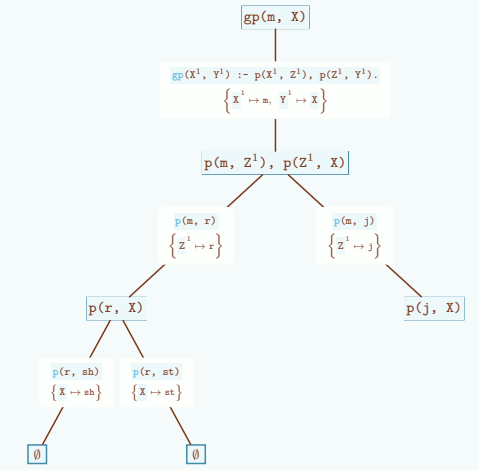
```
Fonction résolution(P, q_1 ∧ ··· ∧ q_n)
    Si n = 0
        Renvoyer {ι}
    Sinon
        Soit res := ∅
        Pour chaque clause t := t1, ..., tk de P
            Soit t' :- t1', ..., tk' := rafraichir(t :- t1, ..., tk)
            Soit σ := unifier(q_1, t')
            Soient t1'' := t1'·σ,  ...,  tk'' := tk'·σ
            Soit res' := résolution(P, t1'' ∧···∧ tk''  ∧ q_2 ∧···∧ q_n)
            Pour chaque σ' de res'
                Soit res := res ∪ {σ ∘ σ'}
            Fin
        Fin
        Renvoyer res
    Fin
Fin
```

La fonction `rafraichir` prend une clause en entrée et renomme toutes ses variables par des variables non encore utilisées.

Il est possible d'illustrer la résolution par un arbre, l'*arbre de résolution*. Chaque nœud contient le but courant et chaque arête est décorée par la clause sélectionnée et la substitution associée.

Exemple

Depuis l'exemple précédent, nous avons l'arbre de résolution



Les atomes ont été abrégés de sorte que

- `gp` signifie `grand_parent` ;
- `p` signifie `parent` ;
- `m` signifie `marvin` ;
- `r` signifie `randy` ;
- `j` signifie `jimbo` ;
- `sh` signifie `shelly` ;
- `st` signifie `stan`.

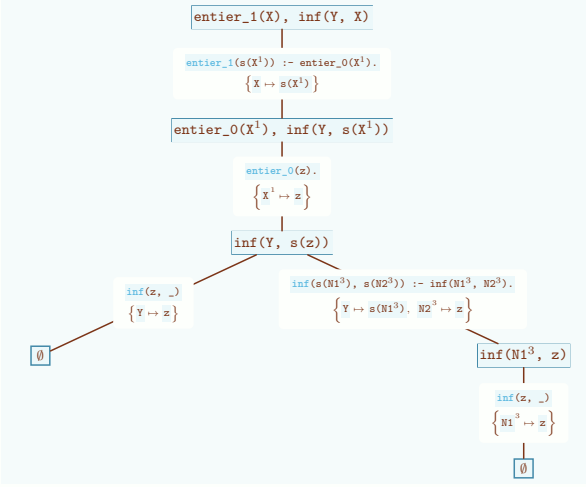
Reprenons les entiers de Peano et plus spécifiquement les deux prédicats et la requête suivants :

```
entier_0(z).
entier_1(s(X)) :-
    entier_0(X).
```

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

```
?- entier_1(X), inf(Y, X).
X = s(z), Y = z ;
X = Y, Y = s(z) ;
false.
```

L'arbre de résolution de cette requête est



Il contient deux solutions :

- $\{Y \mapsto z, X \mapsto s(z)\}$;
- $\{Y \mapsto s(z), X \mapsto s(z)\}$.

Elles s'obtiennent en composant les substitutions du bas vers le haut qui aboutissent aux succès $\emptyset$.

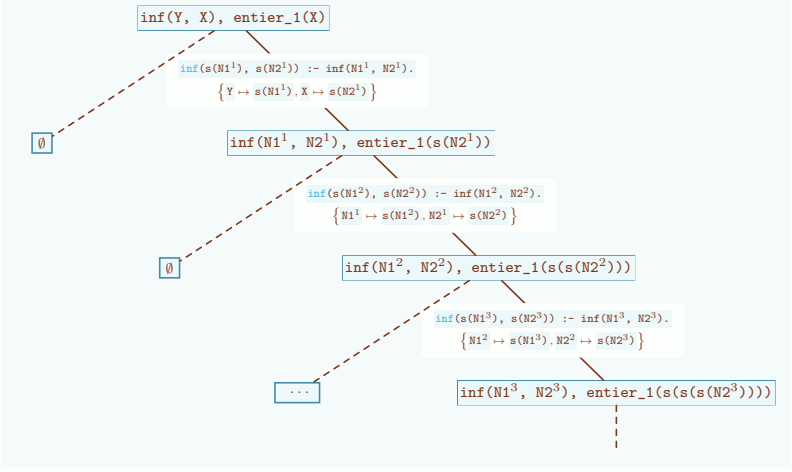
L'interpréteur imprime `false.` après la dernière solution car le but `inf(N1³, z)` est développé en sélectionnant la règle `inf(s(N1⁴), s(N2⁴)) :- inf(N1⁴, N2⁴).` de manière infructueuse.

La requête obtenue en changeant l'ordre de ses sous-buts donne

```
?- inf(Y, X), entier_1(X).
Y = z, X = s(z) ;
Y = X, X = s(z) ;
```

De manière surprenante, la résolution semble continuer après le calcul de ces deux solutions.

Ceci s'explique par l'arbre de résolution de cette requête qui est de la forme



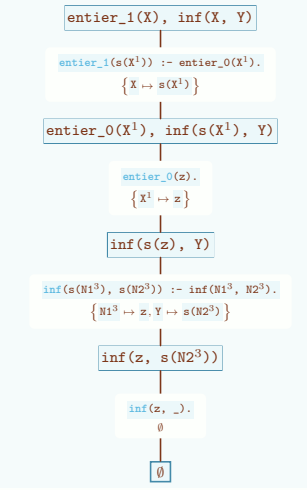
L'arbre de résolution possède une **branche infinie**.
Les deux solutions calculées correspondent aux deux buts $\emptyset$ rencontrés en considérant la clause `inf(z, _).` bifurquant de la branche infinie en son 1er et 2e nœud.
Il y a de plus, pour chaque autre nœud de la branche principale, une sélection de la clause `inf(z, _).` qui ne produit néanmoins aucune solution.
Point important : mettre en premier les parties les plus contraignantes dans les corps des clauses.

La requête obtenue en changeant l'ordre des variables dans `inf` donne

```
?- entier_1(X), inf(X, Y).
X = s(z), Y = s(_).
```

De manière surprenante, la résolution se termine après le calcul de cette solution.

Ceci s'explique par l'arbre de résolution de cette requête qui est



L'arbre de résolution est bien **fini**.

La solution calculée est $\{X \mapsto z, Y \mapsto s(N2^3)\}$. Comme $N2^3$ peut prendre n'importe quelle valeur (elle ne dépend ni de `X` ni de `Y`), SWI-Prolog l'affiche sous la forme d'un **joker** `_`.

Cette solution est finalement très cohérente : la requête initiale demande les `Y` qui sont supérieurs à `s(z)`. La réponse est l'ensemble des termes de la forme `s(_)`. Chacun de ces termes respecte la contrainte spécifiée.

Reprenons les prédicats `entier/1` et `add/3` la requête suivants :

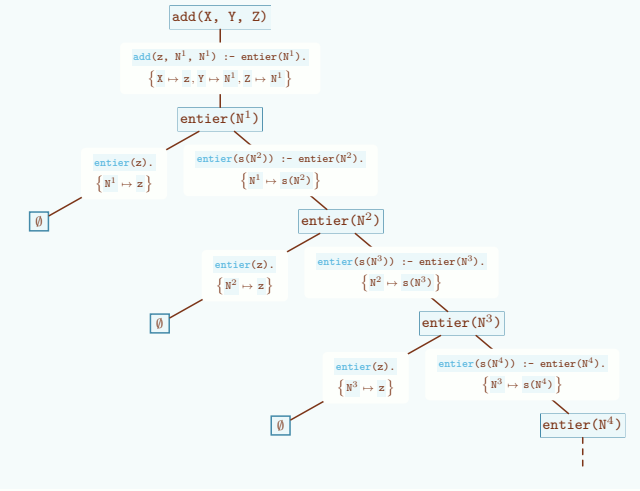
```
entier(z).
entier(s(N)) :- entier(N).
```

```
add(z, N, N) :-
    entier(N).
add(s(N1), N2, s(N3)) :-
    add(N1, N2, N3).
```

```
?- add(X, Y, Z).
X = Y, Y = Z, Z = z ;
X = z, Y = Z, Z = s(z) ;
X = z, Y = Z, Z = s(s(z))
```

Il y a une infinité de solutions.

Ceci s'explique par l'arbre de résolution de cette requête qui est de la forme



Il y a une **branche infinie** apportant une **infinité de solutions**.

À sa racine, la clause `add(s(N1), N2, s(N3)) :- add(N1, N2, N3).` n'est pas sélectionnée car la clause `add(z, N, N) :- entier(N).` produit un arbre infini.

Seules sont trouvées les solutions de la forme $\{X \mapsto z, Y \mapsto s(\ldots(z)\ldots), Z \mapsto s(\ldots(z)\ldots)\}$.

L'arbre de résolution est visité en **profondeur**. Un parcours en profondeur à profondeur bornée trouverait les autres solutions.