Programmation logique : Fiche

Sommaire

1. Introduction

- La programmation logique est un paradigme de programmation qui se concentre sur la logique et le raisonnement déductif pour résoudre des problèmes informatiques.
- Elle utilise des règles logiques pour décrire les relations entre différents éléments d'un système.
- Le langage de programmation le plus couramment utilisé pour la programmation logique est **Prolog**, mais il existe d'autres langages de programmation logique tels que **Datalog** et **Mercury**.
- Les programmes écrits en programmation logique sont constitués de faits (des déclarations sur les relations entre différents éléments) et de règles (des instructions sur la façon de déduire de nouveaux faits à partir des faits existants).
- La programmation logique est souvent utilisée pour la résolution de problèmes de logique, tels que les jeux de logique, les systèmes d'inférence et les systèmes d'expertise. Elle est également utilisée dans des domaines tels que l'intelligence artificielle, la bio-informatique et la vérification formelle de logiciels.
- Les avantages de la programmation logique incluent la capacité à modéliser des problèmes de manière naturelle et intuitive, la facilité de maintenir et de modifier le code et la possibilité de

résoudre des problèmes complexes en utilisant des techniques de raisonnement déductif.

2. Prolog

On utilise **Prolog** comme un démonstrateur de théorèmes pour clauses de Horn.

> Une **clause** est une formule logique de la forme $L_1 \vee L_2 \vee \cdots \vee L_n$ où L_i est un littéral (une variable ou une négation d'une variable).
>
> Une **clause de Horn** a au plus un littéral non négatif (ou positif) et est donc soit un littéral positif soit une disjonction de littéraux négatifs.
>
> Un littéral est dit négatif s'il est de la forme $\neg P_i$, il est positif s'il est de la forme P_i.

- **fait** : clause ayant un seul littéral
- **règle** : clause ayant un littéral positif et un ou plusieurs littéraux négatifs
- **requête** : clause sans littéral positif

2.1. Ecriture

- $\vee$: `;`
- $\wedge$: `,`
- Variable muette (utilisée lorsqu'il n'est pas utile de connaître la valeur d'une variable) : `_`

2.1.1. Faits

Les **faits** sont de la forme : `fait(X)`

- Les règles sont de la forme : `règle(X) :- fait(X), fait(X)`
- Les règles doivent être déclarées par paquets :

$$\begin{cases} A :- B_1^1, B_2^1, \ldots, B_{n_1}^1 \\ A :- B_1^2, B_2^2, \ldots, B_{n_2}^2 \\ \vdots \\ A :- B_1^m, B_2^m, \ldots, B_{n_m}^m \end{cases}$$

$$\Longleftrightarrow$$

$$(-B_1^1, B_2^1, \ldots, B_{n_1}^1) \vee (-B_1^2, B_2^2, \ldots, B_{n_2}^2) \vee \cdots \vee (-B_1^m, B_2^m, \ldots, B_{n_m}^m) \rightarrow A$$

> L'ordre est important en Prolog.

2.1.2. Arithmétique

- `X is Y` : Cette expression est utilisée pour effectuer une évaluation arithmétique et attribuer le résultat à X. Par exemple, `X is 3 + 4` affecterait la valeur 7 à la variable X.
- `X = Y` : Cette expression est utilisée pour unifier les variables X et Y. Si X et Y ont déjà des valeurs, elles doivent être égales pour que l'unification réussisse. Sinon, l'unification va établir une équation entre les deux variables, ce qui signifie que si X change, Y changera également pour rester égale à X.
- `X == Y` : Cette expression est utilisée pour tester l'égalité entre X et Y, en vérifiant si leurs valeurs sont les mêmes. Si X et Y sont des variables non instanciées, l'expression renvoie `true` si les variables sont unifiables, sinon elle renvoie `false`.
- `X =:= Y` : Cette expression est utilisée pour tester l'égalité arithmétique entre X et Y, en vérifiant si leurs valeurs sont égales en termes de calcul arithmétique. Par exemple, `2 + 2 =:= 4` renverrait `true`, tandis que `2 + 2 =:= 5` renverrait `false`.

- `X =\= Y` : Cette expression est utilisée pour tester la non-égalité entre X et Y. Si X et Y sont des variables non instanciées, l'expression renvoie true si les variables ne sont pas unifiables.

2.1.3. Listes

- `|` : constructeur de liste. `[Tête | Queue]` et `Queue` est une liste

2.1.4. Exercice - Construction d'une liste

Construire `[1, 2, 3, 4]`, `[a, b, c, d, [e, f, g, [h, [], i]]]` et `[[[[]]]]`.

```
% [1, 2, 3, 4]
[1|[2|[3|[4|[]]]]]

% [a, b, c, d, [e, f, g, [h, [], i]]]
[a|[b|[c|[d|[[e|[f|[g|[[h|[[]|[i|[]]]]]]]]]]]]]

% [[[[[]]]]]
[[[[]|[]]|[]]|[]]
```

2.1.5. Exercice - Egalité de deux listes

```
% egales(L1, L2)

% Règle 1
egales([],  []).

% Règle 2
egales([T|Q1], [T|Q2]) :-
    egales(Q1, Q2).
```

2.1.6. Exercice - Appartenance à une liste

```
% membre(X, L)

% Règle 1
membre(X, [X|_]).

% Règle 2
membre(X, [_|Q]) :-
    membre(X, Q).
```

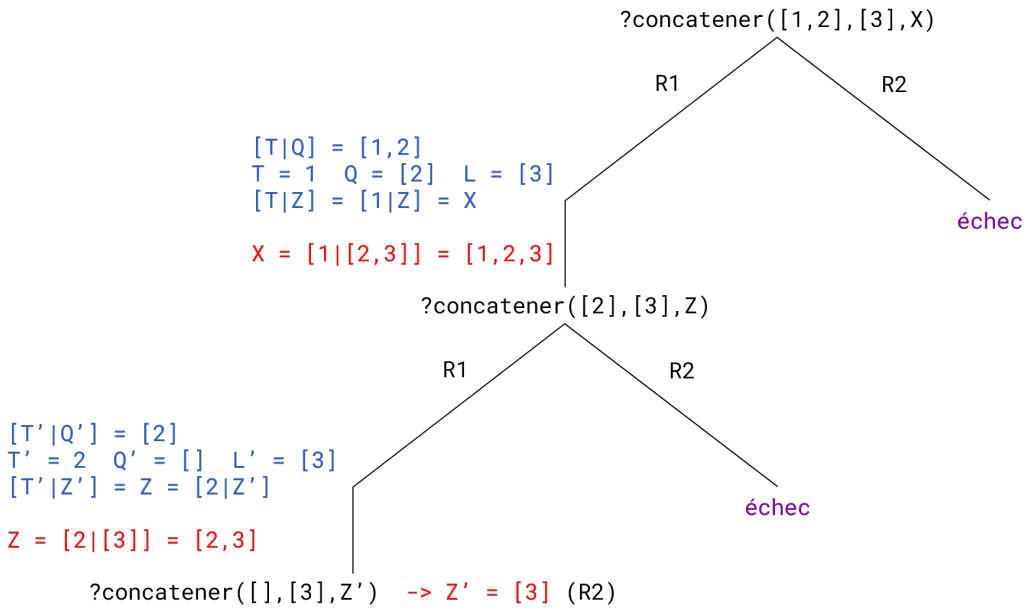
2.1.7. Exercice - Concaténation de deux listes

```
% concatener(L1, L2, L3)

% Règle 1
concatener([T|Q], L, [T|Z]) :-
    concatener(Q, L, Z).
```

```
% Règle 2
concatener([], L, L).
```

Arbre de recherche :



2.1.8. Exercice - Suppression d'un élément d'une liste

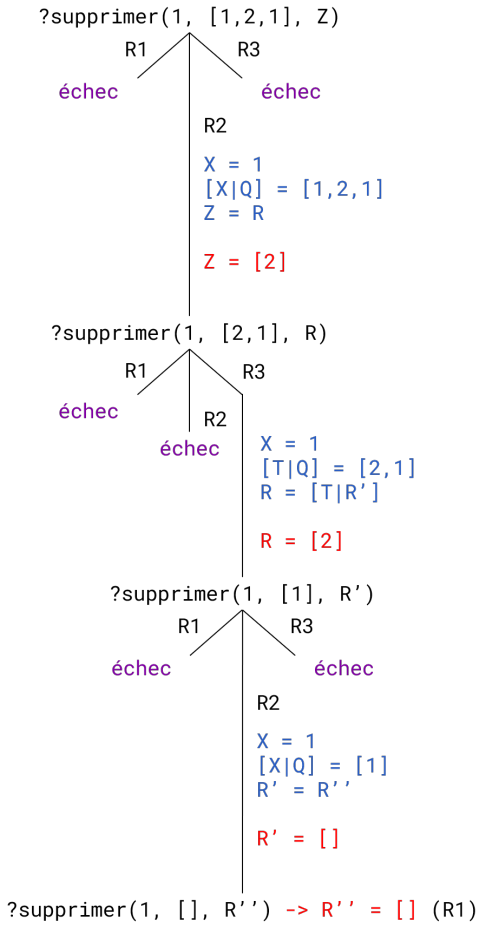
```
% supprimer(X, L, R)

% Règle 1
supprimer(_, [], []).

% Règle 2
supprimer(X, [X|Q], R) :-
    supprimer(X, Q, R).

% Règle 3
supprimer(X, [T|Q], [T|R]) :-
    X \= T,
    supprimer(X, Q, R).
```

Arbre de recherche :



2.1.9. Exercice - Tri d'une liste de nombres (Tri Fusion)

```
% scinder(L, L1, L2)

% Règle 1
scinder([], [], []).    % Pair

% Règle 2
scinder([X], [X], []).  % Impair

% Règle 3
scinder([X,Y|Q], [X|L1], [Y|L2]) :-
    scinder(Q, L1, L2).
```

```
% fusionner(L1, L2, L)

% Règle 1
fusionner([], [], []).

% Règle 2
fusionner([], L, L).

% Règle 3
fusionner(L, [], L).

% Règle 4
fusionner([T1|Q1], [T2|Q2], [T1|Q]) :-
    T1 < T2,
```

```
    fusionner(Q1, [T2|Q2], Q).

% Règle 5
fusionner([T1|Q1], [T2|Q2], [T2|Q]) :-
    T1 >= T2,
    fusionner([T1|Q1], Q2, Q).
```

```
% trier(L, LT)

% Règle 1
trier([], []).

% Règle 2
trier(L, LT) :-
    scinder(L, L1, L2),
    trier(L1, L1T),
    trier(L2, L2T),
    fusionner(L1T, L2T, LT).
```

2.1.10. Exercice - Création de la liste [1, 2, 3, ..., N]

```
construit(0, []).

construit(N, L) :-
    N > 0,
    construit2(1, N, L).


construit2(C, N, [C|L]) :-
    C =< N,
    C1 is C + 1,
    construit2(C1, N, L).

construit2(C, N, []) :-
    C > N.
```

2.1.11. Exercice - Réprésentation des ensembles d'entiers par des listes

- ∈ : `membre`
- ∪ :

```
% union(L1, L2, L)

union([], [], L).

union([T|Q1], L2, [T|Q3]) :-
    \+membre(T, L2),
    union(Q1, L2, Q3).

union([T|Q1], L2, Q3) :-
    membre(T, L2),
```

```
    union(Q1, L2, Q3).
```

- ∩ :

```
% intersection(L1, L2, L3)

intersection([], _, []).

intersection([T|Q1], L2, [T|Q3]) :-
    membre(T, L2),
    intersection(Q1, L2, Q3).

intersection([T|Q1], L2, Q3) :-
    \+membre(T, L2),
    intersection(Q1, L2, Q3).
```

- ⊆ :

```
inclus([], _).

inclus([T|Q1], L2) :-
    membre(T, L2),
    inclus(Q1, L2).
```

- Card :

```
card([], 0).

card([T|Q], X) :-
    card(Q, Y),
    X is Y + 1.
```

- × :

```
calc(_, [], []).

calc(X, [T|Q], [[X, T]|R]) :-
    calc(X, Q, R).
```

2.1.12 Prédicat `selectionner`

> On définit le prédicat `selectionner(,,)` :

```
selectionner(X, [X|L], L).
selectionner(X, [Y|LY], [Y|LZ]) :-
    selectionner(X, LY, LZ).
```

Exemples :

```
? selectionner(A, [1,2,3], B).      %true

/* Résultats :
Premier résultat : A = 1, B = [2, 3]
Deuxième résultat : A = 2, B = [1, 3]
Troisième résultat : A = 3, B = [1, 2]
*/
```

```
? selectionner(1, [1,2,3], B).      %true

/* Résultats :
Premier résultat : B = [2, 3]
*/
```

```
? selectionner(A, [1,2,3], [3]).    %false
```

```
? selectionner(1, L, [2,3]).        %true

/* Résultats :
Premier résultat : L = [1, 2, 3]
Deuxième résultat : L = [2, 1, 3]
Troisième résultat : L = [2, 3, 1]
*/
```

```
? selectionner(1, L, L2).           %true

/* Résultats :
Premier résultat : L = [1], L2 = []
Deuxième résultat : L = [1, _G001], L2 = [_G001]
... Infinité de résultats
*/
```

2.1.13. Exercice - Calcul des permutations d'une liste

```
% permuter(L, LP).

% Règle 1
permuter([T|Q], PL) :-
    permuter(Q, PQ),
    selectionner(T, PL, PQ).

% Règle 2
permuter([], []).
```

2.1.14. Prédicat `renverser`

> Renverser une liste : `renverser(L1, L2)`.

```
renverser(L1, L2) :-
    renverser2(L1, [], L2).

% ---

renverser2([X|LX], Pile, LZ) :-
    renverser2(LX, [X|Pile], LZ).
renverser([], Pile, Pile).
```

2.2. Le CUT : `!`

> Le CUT : `!` permet de forcer la fin de la recherche d'une solution.

Exemple :

```
sup(_, [], []).

sup(X, [Y|L], [Y|Z]) :-
    Y =< X,
    !,
    sup(X, L, Z).

sup(X, [Y|L], Z) :-
    sup(X, L, Z).
```

2.3. Le monde clos

> Le monde clos est un principe de Prolog qui permet de limiter la recherche de solutions. Le `not` ne veut pas dire que la solution n'existe pas, mais qu'elle n'est pas trouvée.
>
> `\+` est la même chose que `not`.

2.4. Exercice - Le problème des N reines

Ecrire un programme qui permet de placer N reines sur un échiquier de taille $N \times N$ sans qu'elles ne se menacent.

On va utiliser deux listes : L et C qui contiennent les lignes et les colonnes des reines :

$$L = [1, 2, 3, ..., N] \text{ et } C = \text{permutation}(L)$$

```
pasDattaque([], []).
pasDattaque([L|L2], [C|C2]) :-
    pasDattaque2(L, C, L2, C2),
    pasDattaque(L2, C2).

% ---

pasDattaque2(_, _, [], []).
pasDattaque2(L, C, [X|L2], [Y|C2]) :-
    pasDattaque3(L, C, X, Y),
    pasDattaque2(L, C, L2, C2).

% ---
```

```
pasDattaque3(L, C, X, Y) :-
    abs(L - X) =\= abs(C - Y).

% ---

reine(N, LS, CS) :-
    construit(N, LS),
    permuter(LS, CS),
    pasDattaque(LS, CS).
```

Pour connaître le nombre de solutions, on peut utiliser le prédicat `findall` : `findall((L, C), reine(8, L, C), X), length(X, Y).`

2.5. Exercice - Jeu du Taquin 3 x 3.

On a un plateau de jeu 3×3 avec des cases numérotées de 1 à 8 et une case vide. Le but est de déplacer les cases pour les mettre dans l'ordre :

$$\begin{array}{ccc} 7 & 3 & 5 \\ 8 & 2 & 4 \\ 1 & & 6 \end{array} \quad \longrightarrow \quad \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{array}$$

3. Calcul propositionnel

3.1. Introduction

> Le **calcul propositionnel** ou **logique des propositions** a pour objet l'étude des formes de raisonnement dont la validité est indépendante de la structure des propositions composantes et résulte uniquement de leurs propriétés d'être **vraies** ou **fausses**.
>
> D'un point de vue syntaxique, le calcul propositionnel est un **système formel**.

3.2. Système formel

Un **système formel** $S = (\sum_S, F_S, A_S, R_S)$ est défini par :

- Les symboles qui sont utilisés $\sum_S$ (alphabet)
- La syntaxe des formules F_S (formules bien formées)
- La donnée d'un ensemble de formules appelées axiomes A_S (axiomes)
- Les méthodes qui permettent de produire / engendrer / inférer de nouvelles formules (les règles) R_S (règles d'inférence)

> Un système formel est **décidable** s'il existe un algorithme qui permet de déterminer si une formule est un théorème ou non.

3.3. Système formel $P0$: calcul propositionnel

> Le système formel $P0$ est défini par :
>
> - $\sum_{P0} = \{a, b, c, d, ai, bi, ci, di, \text{PAS COMPLET}\}$

3.4. Exercice 1

Soit la formule $\varphi = (a \to (b \to (c \vee \neg d))) \to \neg(b \to (\neg a \vee c))$.

Mettre φ sous forme normale conjonctive (FND) et sous forme normale disjonctive (FND).

FNC : $\wedge_j(\vee_i(l_i))$

FND : $\vee_j(\wedge_i(l_i))$

Soit $\alpha \to \beta \equiv \neg\alpha \vee \beta$.

Commençons par supprimer les implications :

$$\begin{aligned}\varphi &= (a \to (b \to (c \vee \neg d))) \to \neg(b \to (\neg a \vee c)) \\ &\equiv \neg(a \to (b \to (c \vee \neg d))) \vee \neg(b \to (\neg a \vee c)) \\ &\equiv \neg(\neg a \vee (b \to (c \vee \neg d))) \vee \neg(\neg b \vee (\neg a \vee c)) \\ &\equiv \neg(\neg a \vee (\neg b \vee (c \vee \neg d))) \vee \neg(\neg b \vee (\neg a \vee c))\end{aligned}$$

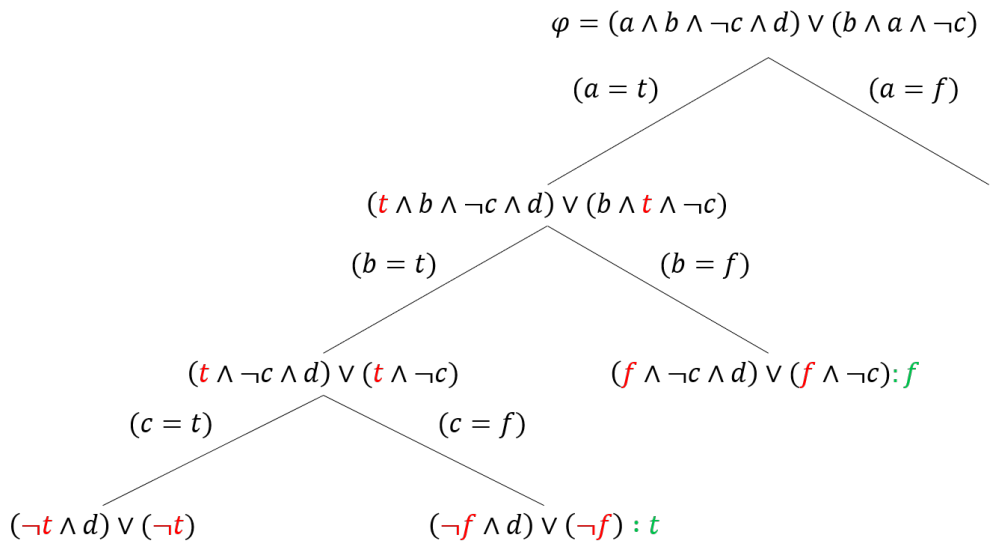
Puis on trouve la FND en supprimant les négations :

$$\begin{aligned}\varphi &\equiv \neg(\neg a \vee (\neg b \vee (c \vee \neg d))) \vee \neg(\neg b \vee (\neg a \vee c)) \\ &\equiv (a \wedge (b \wedge (\neg c \wedge d))) \vee (b \wedge (a \wedge \neg c)) \\ &\equiv (a \wedge b \wedge \neg c \wedge d) \vee (b \wedge a \wedge \neg c)\end{aligned}$$

Puis on distribue les $\wedge$ sur les $\vee$ pour la FNC :

$$\begin{aligned}\varphi &\equiv (a \wedge b \wedge \neg c \wedge d) \vee (b \wedge a \wedge \neg c) \\ &\equiv a \wedge b \wedge \neg c\end{aligned}$$

Vérifions avec l'algorithme de Quine :



$f \Rightarrow \varphi$ n'est pas une tautologie

3.5. Exercice 2

Soit $\varphi = ((a \to d) \to ((d \to e) \to (h \to g))) \to (((a \to d) \to (d \to e)) \to ((a \to d) \to (h \to g)))$

Par l'absurde, montrer que φ est une tautologie.

$$\begin{aligned}i(\varphi) = F \Rightarrow & i[(a \to d) \to ((d \to e) \to (h \to g))] = V \quad (1) \\ & i[((a \to d) \to (d \to e)) \to ((\to d) \to (h \to g))] = F \quad (2)\end{aligned}$$

$$\begin{aligned}(2) \Rightarrow & i[(a \to d) \to (d \to e)] = V \quad (3) \\ & i[(a \to d) \to (h \to g)] = F \quad (4)\end{aligned}$$

$$\begin{aligned}(4) \Rightarrow & i[a \to d] = V \quad (5) \\ & i[h \to g] = F \quad (6)\end{aligned}$$

On a donc :

- D'après (5) et (3) : $[a \to d] = V$ donc $[a] = V$ et $[d] = V$ et $[d \to e] = V$ donc $[e] = V$.
- D'après (4) : $[h \to g] = F$ donc $[h] = V$ et $[g] = F$.
- Si $h \to g$ est faux, alors $d \to e$ est faux pour pouvoir respecter (1) or pour respecter (3), $d \to e$ doit être vrai. On a donc une **contradiction**.

3.6. Exercice 3

> Un coffre-fort est équipé de N serrures différentes. On souhaite fournir les clés de ces serrures à 5 personnes A, B, C, D et E de manière que pour pouvoir ouvrir le coffre il faut que $(A \text{ et } B)$ ou $(A \text{ et } C \text{ et } D)$ ou $(B \text{ et } D \text{ et } E)$ soient simultanément présentes. Dans les autres configurations, le coffre reste fermé. Il peut y avoir plusieurs occurences de la même clé.
>
> Combien de clés sont au minimum nécessaires ? Comment les distribuer ?

Le coffre s'ouvre si :

$$
\begin{aligned}
&(A \wedge B) \vee (A \wedge C \wedge D) \vee (B \wedge D \wedge E) \quad \text{FND} \\
\Leftrightarrow &(A \wedge (A \vee C) \wedge (A \vee D) \wedge (B \vee A) \wedge (B \vee C) \wedge (B \vee D)) \vee (B \wedge D \wedge E) \\
\Leftrightarrow &(A \wedge (B \vee C) \wedge (B \vee D)) \vee (B \wedge D \wedge E) \\
\Leftrightarrow &(A \vee B) \wedge (A \vee D) \wedge (A \vee E) \wedge (B \vee C) \wedge (B \vee C \vee D) \wedge (B \vee C \vee E) \wedge (B \vee D) \wedge (B \vee D) \wedge (B \vee D \vee E) \\
\Leftrightarrow &(A \vee B) \wedge (A \vee D) \wedge (A \vee E) \wedge (B \vee C) \wedge (B \vee D) \quad \text{FNC}
\end{aligned}
$$

Il y a donc au minimum 5 serrures $S_1, S_2, S_3, S_4, \text{ et } S_5$ avec 10 clés réparties ainsi :

Personne	Clés
A	C_{11}, C_{21}, C_{31}
B	C_{12}, C_{41}, C_{51}
C	C_{42}
D	C_{22}, C_{52}
E	C_{32}

3.7. Exercice 4

> Mettre sous forme prénexe les formules :
>
> $$
> \begin{aligned}
> \varphi_1 &= (\forall x\ p(x) \to \neg \exists y\ q(x,y)) \to (\forall x\ \forall y\ q(x,y)) \\
> \varphi_2 &= (\forall x\ p(x) \wedge \exists x\ p(x)) \to (\exists x\ \exists y\ \forall z\ r(x,y,z)) \\
> \varphi_3 &= (\forall f\ p(f(x), y)) \to (\forall y\ g(x, f(x)))
> \end{aligned}
> $$

Pour φ_1 :

$$
\begin{aligned}
\varphi_1 &= (\forall x\ p(x) \to \neg \exists y\ q(x,y)) \to (\forall x\ \forall y\ q(x,y)) \\
&= \neg(\forall x\ p(x) \to \neg \exists y\ q(x,y)) \vee (\forall x\ \forall y\ q(x,y)) \\
&= (\forall x\ p(x) \wedge \exists y\ q(x,y)) \vee (\forall x\ \forall y\ q(x,y))
\end{aligned}
$$