PULSARS

CORRECTION INDÉPENDANTE

# Centrale-Supélec 2026 — Informatique (MPI)

| SESSION | NIVEAU | DURÉE | FILIÈRES |
| --- | --- | --- | --- |
| 2026 | CPGE | — | MPI |

## Partie A — Base de données (Q1 à Q5)

Deux tables : `commune(Nom, INSEE)` et `conseiller(Monde, INSEE, Département)`. Le monde `0` est le monde réel, les mondes `1` à `20` ceux des conseillers.

- **Q1** — les conseillers proposant de créer le Gévaudan.

> **Q1.** Une proposition appartient à un monde idéal dès que `Monde > 0`, et `DISTINCT` élimine les doublons dus aux communes multiples.
>
> ```
> SELECT DISTINCT Monde
> FROM conseiller
> WHERE Département = 'Gévaudan' AND Monde > 0;
> ```
>
> *La condition `Monde > 0` est en principe superflue — le Gévaudan n'existe pas dans le monde réel — mais l'écrire rend la requête indépendante des données.*

- **Q2** — ceux rattachant le Mont-Saint-Michel à l'Ille-et-Vilaine.
- **Q3, Q4** — les communes qui *peuvent* faire partie de la Sarthe, puis celles qui le peuvent sans en faire partie réellement.

> **Q2.** Il faut passer par la table `commune` pour convertir le nom en numéro INSEE. L'énoncé garantit l'unicité du nom, ce qui dispense d'un `DISTINCT` sur la jointure.
>
> ```
> SELECT DISTINCT c.Monde
> FROM conseiller AS c
> JOIN commune AS m ON c.INSEE = m.INSEE
> WHERE m.Nom = 'Mont-Saint-Michel'
>   AND c.Département = 'Ille-et-Vilaine'
>   AND c.Monde > 0;
> ```

- **Q5** — les rattachements *obligatoires* qui ne sont pas réalisés.

> **Q5.** « Obligatoire » signifie « vrai dans **les vingt** mondes idéaux ». On regroupe donc par couple et l'on compte les mondes distincts.
>
> ```
> SELECT INSEE, Département
> FROM conseiller
> WHERE Monde > 0
> GROUP BY INSEE, Département
> HAVING COUNT(DISTINCT Monde) = 20
> EXCEPT
> SELECT INSEE, Département FROM conseiller WHERE Monde = 0;
> ```
>
> *Le `COUNT(DISTINCT Monde)` plutôt que `COUNT(*)` protège contre d'éventuelles lignes en double pour un même monde. C'est le cœur de la définition : un rattachement n'est obligatoire que si **aucun** conseiller ne le conteste.*

> **Q3.** « Pouvoir » signifie « dans au moins un monde idéal » : c'est une simple sélection, l'existence étant assurée par la présence même de la ligne.
>
> ```
> SELECT DISTINCT INSEE
> FROM conseiller
> WHERE Département = 'Sarthe' AND Monde > 0;
> ```
>
> **Q4.** On retranche les communes qui sont dans la Sarthe dans le monde réel.
>
> ```
> SELECT DISTINCT INSEE FROM conseiller
> WHERE Département = 'Sarthe' AND Monde > 0
> EXCEPT
> SELECT INSEE FROM conseiller
> WHERE Département = 'Sarthe' AND Monde = 0;
> ```

## Partie B · I — Analyse lexicale en OCaml (Q6 à Q9)

- Q6 — `appartient`, sans utiliser `String.contains`.

> **Q6.** Un parcours suffit. L'évaluation paresseuse du `||` arrête la recherche dès que le caractère est trouvé.
>
> ```
> let appartient (s : string) (c : char) : bool =
>   let n = String.length s in
>   let rec aux i = i < n && (s.[i] = c || aux (i + 1)) in
>   aux 0
> ```

- Q7 — `sous_chaine`, sans utiliser `String.sub`, avec levée d'exception.

> **Q7.** `String.init` construit une chaîne à partir d'une fonction d'indice. L'assertion couvre les trois débordements possibles.
>
> ```
> let sous_chaine (s : string) (debut : int) (longueur : int) : string =
>   assert (debut >= 0 && longueur >= 0 && debut + longueur <= String.length s);
>   String.init longueur (fun i -> s.[debut + i])
> ```
>
> *Sur `sous_chaine "abc" 1 3`, la condition `1 + 3 <= 3` est fausse : l'assertion échoue, comme demandé.*

- **Q8 —** `detecte_lexeme`, qui fait tourner l'automate donné.

> **Q8.** L'automate se lit ainsi : on boucle sur l'état `0` tant qu'on rencontre des espaces, une parenthèse mène à l'état `3` — lexème d'un seul caractère —, une lettre mène à l'état `1`, et à partir de là on accepte lettres et `_`.
>
> ```
> exception Erreur_Lexicale
>
> let detecte_lexeme (chaine : string) (debut : int) : int * int =
>   let n = String.length chaine in
>   let d = ref debut in
>   while !d < n && appartient espaces chaine.[!d] do incr d done;   (* etat 0 *)
>   if !d >= n then raise Erreur_Lexicale;
>   let c = chaine.[!d] in
>   if c = '(' || c = ')' then (!d, !d + 1)                          (* etat 3 *)
>   else if appartient alphabet c then begin                         (* etat 1 *)
>     let f = ref (!d + 1) in
>     while !f < n && (appartient alphabet chaine.[!f] || chaine.[!f] = '_')
>     do incr f done;                                                (* etats 1, 2 *)
>     (!d, !f)
>   end
>   else raise Erreur_Lexicale
> ```
>
> Le point à ne pas manquer : le caractère `_` **n'est accepté qu'après une lettre**, ce que traduit le fait qu'on exige `appartient alphabet c` avant d'entrer dans la seconde boucle. L'automate le dit en n'autorisant `_` que sur la transition `1 → 2`, jamais depuis `0`.
>
> La boucle interne cherche le **plus long** lexème, conformément à l'énoncé.

- **Q9 —** `analyse_lexicale`, qui produit la liste des lexèmes.

> **Q9.** On enchaîne les appels en repartant de la fin du lexème précédent, en s'arrêtant quand il ne reste que des espaces.
>
> ```
> let analyse_lexicale (chaine : string) : lexeme list =
>   let n = String.length chaine in
>   let rec aux i acc =
>     let j = ref i in
>     while !j < n && appartient espaces chaine.[!j] do incr j done;
>     if !j >= n then List.rev acc
>     else
>       let (d, f) = detecte_lexeme chaine i in
>       aux f (produit_lexeme chaine d f :: acc)
>   in
>   aux 0 []
> ```
>
> L'accumulateur est construit à l'envers puis retourné une seule fois : la complexité reste **linéaire**, là où une concaténation à droite à chaque étape serait quadratique.

## Partie B · II — Deux grammaires (Q10 à Q12)

`G` et `H` sont deux grammaires non contextuelles pour les formules déontiques, la seconde imposant des parenthèses autour de chaque conjonction.

- **Q10** — les mots `m_1` et `m_2` dérivent-ils de `G` ? de `H` ?

> **Q10.** *Le mot `m_1` = « x doit ( y et z ) ».* Aucune des deux grammaires ne permet de **juxtaposer** deux formules sans opérateur entre elles : les seules règles binaires sont `E → A et A` pour `G` et `B → E et E` pour `H`, toutes deux exigeant le terminal `et`. Ici `x` et `doit (…)` se suivent sans connecteur :
>
> `m_1` ne dérive **ni de `G`, ni de `H`**.
>
> *Le mot `m_2` = « x et doit ( non y et z ) ».* Il dérive de `G` :
>
> `A → E → A et A`, le premier `A` donnant `n = x`, le second `A → E → doit A`, puis `A → ( E )` avec `E → A et A` produisant `non y` — par `E → non A` — et `z`.
>
> Il ne dérive **pas de `H`** : le symbole initial y est `E`, dont les règles sont `n`, `( B )` et `C`. Or `m_2` ne commence pas par une parenthèse, n'est pas une variable seule, et ne commence ni par `non` ni par `doit`. La conjonction n'est accessible que **sous parenthèses**, via `E → ( B )`.

- **Q11** — engendrent-elles le même langage ?

> **Q11. Non.** Le mot `m_2` est engendré par `G` et pas par `H` : les deux langages diffèrent.
>
> `H` impose des parenthèses autour de chaque conjonction, `G` non — c'est précisément ce qui rend `G` ambiguë.

- **Q12** — sont-elles ambiguës ?

> **Q12. `G` est ambiguë.** Le mot « x et y et z » admet deux arbres de dérivation distincts :
>
> - `E → A et A` avec `A_1 = x` et `A_2 = « y et z »` ;
> - `E → A et A` avec `A_1 = « x et y »` et `A_2 = z`.
>
> Aucune règle de priorité ou d'associativité ne départage ces deux lectures.
>
> **`H` n'est pas ambiguë.** Le **premier lexème** d'un mot détermine à lui seul la règle applicable :
>
> | Premier lexème | Règle forcée |
> |---|---|
> | une variable `n` | `E → n` |
> | `(` | `E → ( B )` |
> | `non` ou `doit` | `E → C` |
>
> Le choix étant déterministe à chaque étape, la dérivation est unique — c'est d'ailleurs ce qui rend possible l'analyse par descente récursive de la partie III.

## Partie B · III — Analyse syntaxique par descente récursive (Q13 à Q15)

Trois fonctions mutuellement récursives, une par symbole non terminal de `H`.

- **Q13, Q14** — compléter les deux blocs manquants.
- **Q15** — la fonction `analyse_syntaxique`.

> **Q15.** `derive_E` renvoie un arbre et **le reste non consommé**. Une analyse correcte doit consommer toute la liste : si le reste n'est pas vide, la formule était mal formée — par exemple des lexèmes en trop après une formule complète.
>
> ```
> let analyse_syntaxique (l : lexeme list) : formule =
>   match derive_E l with
>   | (f, []) -> f
>   | _ -> raise Erreur_Syntaxique
> ```
>
> *Oublier le test sur le reste est l'erreur classique : `analyse_syntaxique` sur `[Lex_var "x"; Lex_var "y"]` renverrait alors `Var "x"` en silence, au lieu de signaler une erreur.*

> **Q13.** Le premier bloc se place après avoir reconnu une première formule `f1` suivie du lexème `et` : il reste à analyser la seconde opérande, conformément à la règle `B → E et E`.
>
> ```
> let (f2, q2) = derive_E q in
> (Et (f1, f2), q2)
> ```
>
> Le reste `q2` est propagé tel quel : c'est ce qui permet à l'appelant — ici `derive_E`, qui attend la parenthèse fermante — de poursuivre l'analyse au bon endroit.
>
> **Q14.** Le second bloc implémente la règle `C → non E | doit E`. Le lexème de tête décide, et l'absence des deux mène à l'échec.
>
> ```
> and derive_C lexemes = match lexemes with
>   | Lex_non :: q -> let (f, q2) = derive_E q in (Non f, q2)
>   | Lex_doit :: q -> let (f, q2) = derive_E q in (Doit f, q2)
>   | _ -> raise Erreur_Syntaxique
> ```

## Partie C · I — Manipuler les formules en C (Q16 à Q19)

Les formules sont des arbres alloués sur le tas, de type `formule`.

- **Q16** — `cree_formule`.

> **Q16.**
>
> ```
> formule *cree_formule(int op, int variable, formule *fg, formule *fd) {
>     formule *f = malloc(sizeof(formule));
>     if (f == NULL) exit(EXIT_FAILURE);
>     f->op = op;
>     f->variable = variable;
>     f->gauche = fg;
>     f->droite = fd;
>     return f;
> }
> ```
>
> Le test sur le retour de `malloc` n'est pas une formalité : sans lui, un échec d'allocation provoquerait un déréférencement de `NULL`.

- **Q17** — `compter_peut`, qui compte les occurrences de $\diamond$.

> **Q17.** Un parcours de l'arbre, en additionnant les deux sous-arbres. Le test `f == NULL` traite d'un coup les feuilles et les opérateurs unaires, dont le fils droit vaut `NULL`.
>
> ```
> int compter_peut(formule *f) {
>     if (f == NULL) return 0;
>     int n = (f->op == PEUT) ? 1 : 0;
>     return n + compter_peut(f->gauche) + compter_peut(f->droite);
> }
> ```

- **Q18** — `free_formule`, qui libère toutes les sous-formules.

> **Q18.** L'ordre est **impératif** : il faut libérer les fils **avant** le père, sinon les pointeurs vers les sous-formules seraient perdus.
>
> ```
> void free_formule(formule *f) {
>     if (f == NULL) return;
>     free_formule(f->gauche);
>     free_formule(f->droite);
>     free(f);
> }
> ```
>
> *C'est un parcours en profondeur postfixe. L'inverse — libérer le père d'abord — compile sans avertissement et lit de la mémoire libérée : le pire des bogues, celui qui ne plante pas systématiquement.*

- **Q19** — `retirer_implications`, qui produit une **copie** sans implications.

> **Q19.** Les trois contraintes de l'énoncé imposent une **copie profonde** : on ne réutilise aucun nœud de `f`, et on n'en modifie aucun.
>
> ```
> formule *retirer_implications(formule *f) {
>     if (f == NULL) return NULL;
>     if (f->op == IMPLIQUE) {
>         formule *g = retirer_implications(f->gauche);
>         formule *d = retirer_implications(f->droite);
>         return cree_formule(OU, -1, cree_formule(NON, -1, g, NULL), d);
>     }
>     return cree_formule(f->op, f->variable,
>                         retirer_implications(f->gauche),
>                         retirer_implications(f->droite));
> }
> ```
>
> L'implication `φ → ψ` devient `¬φ ∨ ψ`, ce qui ajoute un nœud `NON` au-dessus de la copie du membre gauche.

## Partie C · II — Univers de Kripke et modèles (Q20 à Q25)

Un *univers* est un graphe de mondes ; `□φ` se lit « `φ` dans tous les mondes idéaux », `◇φ` « dans au moins un ».

- **Q20** — `satisfaction`, pour une formule purement propositionnelle.
- **Q21, Q22** — l'univers `E` de l'énoncé satisfait-il deux formules données ?
- **Q23 à Q25** — satisfiabilité de `θ_1`, `θ_2` et de leurs négations, puis la conséquence logique.

> **Q20.** Un parcours récursif de l'arbre, la valuation du monde donnant la valeur des feuilles.

```
bool satisfaction(monde *m, formule *f) {
    switch (f->op) {
        case VARIABLE: return m->valuation[f->variable];
        case NON:      return !satisfaction(m, f->gauche);
        case ET:       return satisfaction(m, f->gauche)
                           && satisfaction(m, f->droite);
        default:       exit(EXIT_FAILURE);   /* hors du fragment traite */
    }
}
```

**Q21. Non.** Vérifions $\Box\Box Q$ au monde $i$, dont les successeurs sont $j$ et $k$.

Il faudrait que $\Box Q$ soit vrai en $j$. Or le seul successeur de $j$ est $l$, où $Q$ est **faux**. Donc $\Box Q$ est faux en $j$, et $\Box\Box Q$ est faux en $i$.

**Q22. Oui.** Il suffit d'exhiber un successeur de $i$ satisfaisant $P \wedge \Box\neg P$.

Le monde $k$ convient : $P$ y est vrai, et son unique successeur $l$ vérifie $\neg P$, donc $\Box\neg P$ est vrai en $k$. Comme $R(i, k)$, on a bien $E \models_i \Diamond(P \wedge \Box\neg P)$.

*Le monde $j$ ne convient pas : $P$ y est faux.*

**Q23.** *$\theta_1 = \Box\Box P \to \Box P$ est satisfiable.* Prenons l'univers à un seul monde $m$ avec l'arc $m \to m$, et $P$ faux en $m$. Alors $\Box P$ est faux en $m$, donc $\Box\Box P$ aussi : l'implication est vraie par prémisse fausse.

*Sa négation est satisfiable.* Prenons $i \to j$, où $j$ **n'a aucun successeur**, et $P$ faux en $j$.

- $\Box P$ en $j$ est vrai **par vacuité** : il n'y a aucun monde idéal à examiner.
- Donc $\Box\Box P$ en $i$ est vrai, $j$ étant le seul successeur.
- Mais $\Box P$ en $i$ demande $P$ vrai en $j$, ce qui est faux.

L'implication est donc fausse en $i$ : $\neg\theta_1$ y est satisfaite.

*La vacuité est le ressort de tout l'exercice : un monde sans idéal satisfait toutes les obligations.*

**Q24.** *$\theta_2 = \Box(\Box P \to P)$ est satisfiable* : un seul monde $m$, l'arc $m \to m$ et $P$ vrai en $m$. L'implication est vraie en $m$, donc $\theta_2$ aussi.

*Sa négation est satisfiable* : reprenons $i \to j$ avec $j$ sans successeur et $P$ faux en $j$. En $j$, $\Box P$ est vrai par vacuité et $P$ est faux : l'implication $\Box P \to P$ est **fausse** en $j$. Comme $j$ est un successeur de $i$, $\theta_2$ est faux en $i$.

**Q25.** *$\theta_1$ est conséquence logique de $\theta_2$.* Soit un univers et un monde $i$ tels que $\theta_2$ y soit vrai. Supposons $\Box\Box P$ vrai en $i$ et montrons $\Box P$.

Soit $j$ un successeur quelconque de $i$. De $\theta_2$ on tire que $\Box P \to P$ est vrai en $j$ ; de $\Box\Box P$ on tire que $\Box P$ est vrai en $j$. Le modus ponens donne $P$ vrai en $j$.

Ceci valant pour tout successeur, $\Box P$ est vrai en $i$ : $\theta_1$ est vrai.

*La réciproque est fausse.* Il faut un univers où $\theta_1$ est vrai et $\theta_2$ faux. Prenons $i$ avec **deux** successeurs :

- $j$, sans successeur, avec $P$ faux — il rend $\theta_2$ faux, comme en **Q24** ;
- $j'$, avec un successeur $l$ où $P$ est faux — donc $\Box P$ est faux en $j'$.

Alors $\Box\Box P$ est faux en $i$ — le monde $j'$ en témoigne —, donc $\theta_1$ est vrai par prémisse fausse, tandis que $\theta_2$ est faux à cause de $j$.

## Partie C · III — Formules réduites (Q26 à Q29)

Dans une formule *réduite*, les négations ne portent que sur des variables et il n'y a plus d'implication.

- **Q26** — les règles de De Morgan.

> **Q26.** Pour les connecteurs propositionnels :
>
> $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ et $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$.
>
> Et leurs analogues modaux, que l'énoncé signale déjà pour $\Diamond$ :
>
> $\neg\Box\varphi \equiv \Diamond\neg\varphi$ et $\neg\Diamond\varphi \equiv \Box\neg\varphi$.
>
> *Ces deux dernières sont les mêmes règles : $\Box$ est une conjonction sur les mondes idéaux, $\Diamond$ une disjonction.*

- **Q27** — toute formule déontique équivaut à une formule réduite.
- **Q28, Q29** — les fonctions `est_reduite` et `satisfaction2`.

> **Q27.** Récurrence sur la structure de la formule, en deux temps.
>
> D'abord, on **élimine les implications** par $\varphi \to \psi \equiv \neg\varphi \vee \psi$ — c'est exactement ce que fait la fonction de **Q19**.
>
> Ensuite, on **fait descendre les négations** vers les feuilles :
>
> - devant une variable, on s'arrête : $\neg P$ est réduite ;
> - devant $\wedge$ ou $\vee$, on applique De Morgan et l'on recommence sur les deux sous-formules, strictement plus petites ;
> - devant $\Box$ ou $\Diamond$, on applique la dualité modale, puis on recommence ;
> - devant une négation, $\neg\neg\varphi \equiv \varphi$ supprime les deux.
>
> Chaque étape fait décroître la taille de la formule sous la négation : le procédé termine, et le résultat est réduit par construction.

**Q28.** La définition est structurelle, la traduction est directe. Le seul cas subtil est celui de `NON`, qui n'est admis **que** devant une variable.

```
bool est_reduite(formule *f) {
    switch (f->op) {
        case VARIABLE: return true;
        case NON:      return f->gauche->op == VARIABLE;
        case ET:
        case OU:       return est_reduite(f->gauche) && est_reduite(f->droite);
        case DOIT:
        case PEUT:     return est_reduite(f->gauche);
        default:       return false;          /* IMPLIQUE */
    }
}
```

**Q29.** Les deux modalités se traduisent par une boucle sur les mondes, filtrée par la matrice d'adjacence : universelle pour `□`, existentielle pour `◇`.

```
bool satisfaction2(univers *u, int i, formule *f) {
    switch (f->op) {
        case VARIABLE: return u->mondes[i].valuation[f->variable];
        case NON:      return !u->mondes[i].valuation[f->gauche->variable];
        case ET: return satisfaction2(u, i, f->gauche)
                     && satisfaction2(u, i, f->droite);
        case OU: return satisfaction2(u, i, f->gauche)
                     || satisfaction2(u, i, f->droite);
        case DOIT:
            for (int j = 0; j < u->nb_mondes; j++)
                if (u->matrice[i][j] && !satisfaction2(u, j, f->gauche))
                    return false;
            return true;                       /* vrai par vacuite si aucun arc */
        case PEUT:
            for (int j = 0; j < u->nb_mondes; j++)
                if (u->matrice[i][j] && satisfaction2(u, j, f->gauche))
                    return true;
            return false;
        default: exit(EXIT_FAILURE);
    }
}
```

Le cas `NON` exploite la forme réduite : son fils est **nécessairement** une variable, on lit donc directement la valuation sans rappel récursif.

## Partie C · IV — Décidabilité par dépliage arborescent (Q30 à Q34)

On déplie un univers en un arbre de profondeur `N`.

- **Q30, Q31** — la taille de l'arbre, puis sa construction effective.
- **Q32, Q33** — la satisfaction se transporte à l'arbre.
- **Q34** — la satisfiabilité d'une formule déontique est décidable.

> **Q34.** Soit `φ` une formule déontique, `N` son nombre de modalités et `V` le nombre de variables qui y figurent.
>
> D'après **Q27**, on peut supposer `φ` réduite. D'après **Q32**, si `φ` est satisfiable, elle l'est dans un univers **arborescent de profondeur `N`**, dont le degré sortant peut être borné par le nombre de sous-formules `◇ψ` — un témoin par sous-formule suffit.
>
> Il n'existe qu'un **nombre fini** de tels arbres, à valuations près : au plus `2^V` valuations par monde, et un nombre borné de mondes. L'algorithme qui les énumère tous et teste la satisfaction par `satisfaction2` **termine** et répond correctement.
>
> Le problème est donc **décidable**.

**Q30.** La récurrence de l'énoncé se traduit littéralement : un arbre de profondeur `0` compte un monde ; sinon, la racine plus les arbres issus de chaque successeur.

```
int taille(univers *u, int i, int N) {
    if (N == 0) return 1;
    int s = 1;
    for (int j = 0; j < u->nb_mondes; j++)
        if (u->matrice[i][j]) s += taille(u, j, N - 1);
    return s;
}
```

*Cette taille peut être **exponentielle** en `N` — de l'ordre de `d^N` pour un degré sortant `d`. C'est le prix du dépliage, et c'est pourquoi la partie V cherchera des témoins plus petits.*

**Q32.** Récurrence sur la structure de `φ`, l'hypothèse portant sur toutes les profondeurs à la fois.

Les cas des variables et des négations de variables sont immédiats : la copie préserve la valuation. Les cas `∧` et `∨` s'obtiennent par hypothèse de récurrence sur les deux sous-formules.

Pour `□ψ` : si `U ⊨_i □ψ`, alors `ψ` est vrai dans tous les successeurs de `i`. Chaque successeur du nœud `r` dans l'arbre est la copie d'un successeur de `i`, et l'arbre restant sous lui a la profondeur `N − 1`, suffisante puisque `ψ` contient une modalité de moins que `□ψ`. L'hypothèse de récurrence conclut. Le cas `◇ψ` est identique, avec un seul témoin au lieu de tous.

*C'est ici que la forme **réduite** est indispensable : sans elle, une négation pourrait porter sur une modalité, et l'implication changerait de sens.*

**Q33.** Ne pas dupliquer deux univers identiques revient à **partager** des sous-arbres isomorphes. Or la relation de satisfaction ne dépend que de la structure **accessible** depuis le monde considéré : deux sous-arbres isomorphes satisfont exactement les mêmes formules.

Le raisonnement de **Q32** est donc inchangé, et l'univers obtenu est plus petit.

## Partie C · V — NP-complétude (Q35 à Q39)

Un univers est *total* lorsque tout monde est idéal pour tout monde.

- **Q35** — DSAT est NP-complet si et seulement s'il est dans NP.

> **Q35.** Le sens direct est trivial : être NP-complet, c'est être dans NP **et** NP-difficile.
>
> Pour la réciproque, il suffit de montrer que DSAT est **NP-difficile**, ce qui ne dépend pas de son appartenance à NP. Or une formule propositionnelle est un cas particulier de formule déontique — sans aucune modalité —, et elle est satisfiable au sens propositionnel si et seulement si elle l'est au sens déontique : il suffit de prendre un univers à un seul monde.
>
> Cette réduction, calculable en temps constant, ramène **SAT** à DSAT. Donc DSAT est NP-difficile, et s'il est dans NP, il est NP-complet.

- **Q36** — une formule satisfiable dans aucun univers total.
- **Q37, Q38** — vérification rapide, et existence d'un témoignage de taille polynomiale.

> **Q36.** La formule `P ∧ □¬P` convient.
>
> *Elle est satisfiable* : prenons `i → j` avec `P` vrai en `i` et `P` faux en `j`. Alors `P` est vrai en `i`, et `□¬P` aussi puisque `j` est le seul monde idéal.
>
> *Elle ne l'est dans aucun univers total* : dans un tel univers, tout monde est idéal pour lui-même, donc `i` est son propre successeur. `□¬P` en `i` imposerait alors `¬P` en `i`, en contradiction avec `P`.
>
> *C'est la formule de l'introduction : « la couronne a été volée, mais elle n'aurait pas dû l'être ». Elle n'a de sens que si le monde réel n'est pas lui-même idéal.*

- **Q39** — TOTAL-DSAT est NP-complet.

> **Q39.** *TOTAL-DSAT est dans NP.* Le certificat est un univers total d'au plus `|φ| + 1` mondes avec leurs valuations — de taille polynomiale d'après **Q38** —, et sa vérification coûte `O(N|φ|)` d'après **Q37**, donc un temps polynomial.
>
> *TOTAL-DSAT est NP-difficile* par la même réduction qu'en **Q35** : une formule propositionnelle est satisfiable si et seulement si elle l'est dans un univers total à un seul monde.
>
> Donc **TOTAL-DSAT est NP-complet**.

**Q37.** Dans un univers **total**, tout monde voit tous les autres. La valeur de $\Box\psi$ ne dépend donc **pas** du monde où on l'évalue : elle vaut « $\psi$ est vrai partout », et $\Diamond\psi$ vaut « $\psi$ est vrai quelque part ».

On procède alors par un parcours **ascendant** de l'arbre de $\varphi$ : pour chaque sous-formule, on calcule sa valeur dans chacun des $N$ mondes. Il y a $|\varphi|$ sous-formules et $N$ mondes, soit $N|\varphi|$ cases.

Chaque case propositionnelle se calcule en temps constant à partir des cases déjà remplies. Les cases modales se calculent **une seule fois** pour toute la sous-formule, par un parcours des $N$ mondes, puis se recopient. Le coût total est

$O(N|\varphi|)$.

**Q38.** Pour chaque sous-formule $\Diamond\psi$ de $\varphi$ vraie en $i$, choisissons **un** monde témoin où $\psi$ est vraie. Il y a au plus $|\varphi|$ telles sous-formules, donc au plus $|\varphi|$ témoins, auxquels on ajoute $i$ : au plus $|\varphi| + 1$ mondes.

Soit $T$ le sous-graphe total induit sur ces mondes. La satisfaction est préservée, par récurrence sur $\varphi$ réduite :

- pour $\Diamond\psi$, le témoin choisi appartient à $T$, donc $\Diamond\psi$ y reste vraie ;
- pour $\Box\psi$, **restreindre** l'ensemble des mondes ne peut que faciliter : moins de mondes à vérifier.

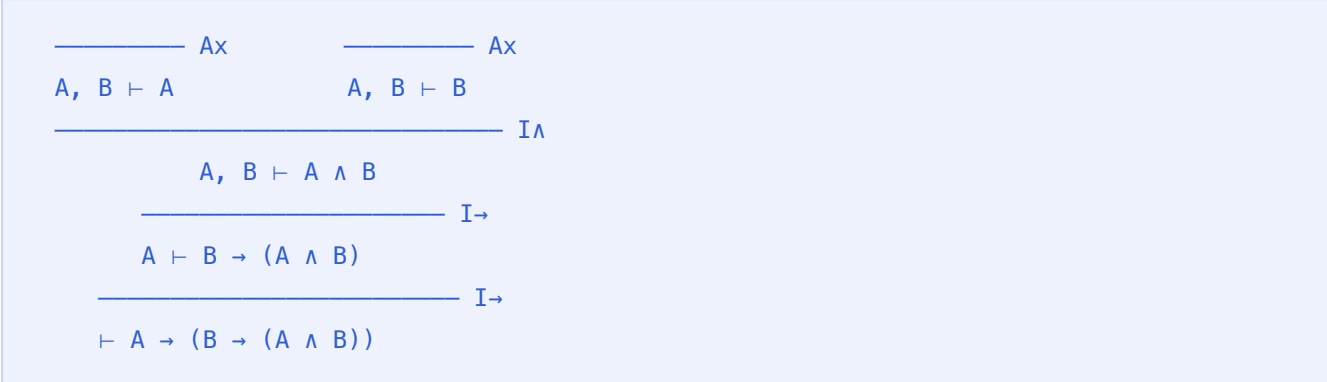
C'est ici que la forme réduite compte à nouveau : sans elle, une négation devant $\Box$ inverserait ce raisonnement.

La taille de $T$ est **linéaire** en $|\varphi|$, donc polynomiale.

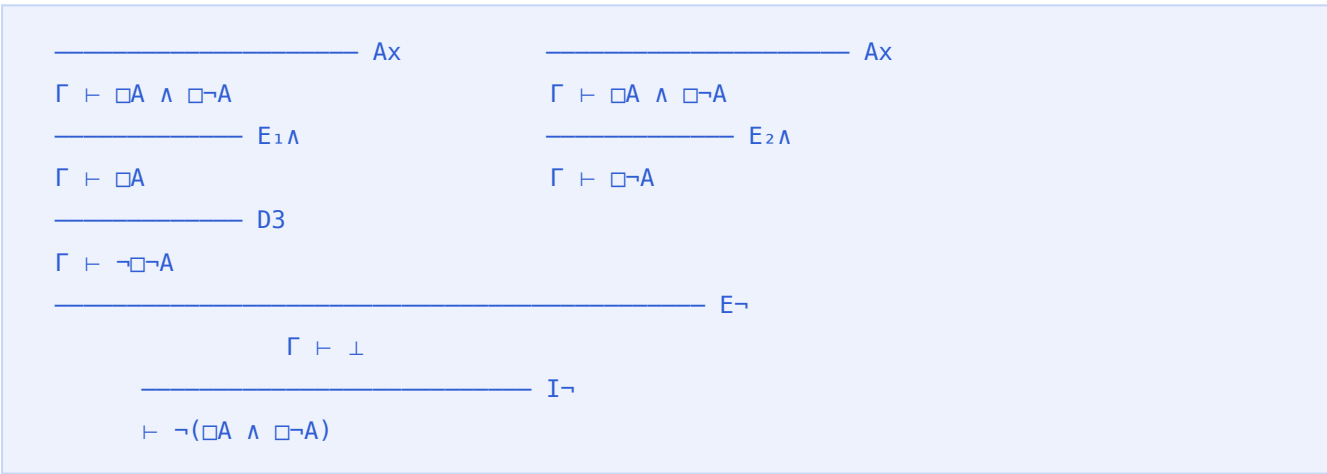
## Partie D — Déduction naturelle déontique (Q40 à Q42)

Le système `D` étend la déduction naturelle classique par trois règles portant sur `□`.

- **Q40** — un arbre de preuve pour `⊢ A → (B → (A ∧ B))`.

> **Q40.** On part des axiomes et l'on décharge les hypothèses une à une.
>
> 
> 
>
> Les deux `I→` déchargent `B` puis `A`, dans cet ordre — l'inverse produirait `B → (A → (A ∧ B))`.

- **Q41** — le système interdit les obligations contradictoires.

> **Q41.** Posons `Γ = {□A ∧ □¬A}` et dérivons une contradiction.
>
> 
> 
>
> La règle **D3** est le pivot : elle dit que ce qui est obligatoire n'est pas interdit. Avec `□¬A`, qui affirme que `¬A` est obligatoire, on obtient l'absurdité.
>
> *Autrement dit, `D` garantit qu'aucun système d'obligations cohérent ne peut imposer une chose et son contraire — ce qui est bien le minimum qu'on attende d'une logique du devoir.*

- **Q42** — l'équivalence $\Box(A \wedge B) \leftrightarrow (\Box A \wedge \Box B)$.

> **Q42.** *Sens direct :* $\vdash \Box(A \wedge B) \to (\Box A \wedge \Box B)$.
>
> La formule $(A \wedge B) \to A$ est un **théorème** : de $A \wedge B \vdash A$ par $\mathrm{Ax}$ puis $\mathrm{E}_1\wedge$, on tire $\vdash (A \wedge B) \to A$ par $\mathrm{I}\to$. La règle **D1** — sans contexte à gauche — donne alors
>
> $\vdash \Box((A \wedge B) \to A)$,
>
> puis **D2** donne $\vdash \Box(A \wedge B) \to \Box A$. Le même raisonnement avec $\mathrm{E}_2\wedge$ donne $\vdash \Box(A \wedge B) \to \Box B$.
>
> En posant $\Gamma = \{\Box(A \wedge B)\}$, deux $\mathrm{E}\to$ fournissent $\Gamma \vdash \Box A$ et $\Gamma \vdash \Box B$, puis $\mathrm{I}\wedge$ donne $\Gamma \vdash \Box A \wedge \Box B$ et $\mathrm{I}\to$ conclut.
>
> *Sens réciproque :* $\vdash (\Box A \wedge \Box B) \to \Box(A \wedge B)$.
>
> On part du théorème de **Q40**, $\vdash A \to (B \to (A \wedge B))$. La règle **D1** le rend obligatoire :
>
> $\vdash \Box(A \to (B \to (A \wedge B)))$,
>
> et **D2** le distribue : $\vdash \Box A \to \Box(B \to (A \wedge B))$.
>
> Posons $\Gamma = \{\Box A \wedge \Box B\}$. Alors $\Gamma \vdash \Box A$ par $\mathrm{E}_1\wedge$, donc $\Gamma \vdash \Box(B \to (A \wedge B))$ par $\mathrm{E}\to$. Une seconde application de **D2** donne $\Gamma \vdash \Box B \to \Box(A \wedge B)$, et comme $\Gamma \vdash \Box B$ par $\mathrm{E}_2\wedge$, un dernier $\mathrm{E}\to$ donne
>
> $\Gamma \vdash \Box(A \wedge B)$,
>
> puis $\mathrm{I}\to$ conclut. Les deux implications établissent l'équivalence.
>
> *L'usage répété de D1 puis D2 est le schéma de raisonnement de toute cette logique : on prouve un théorème, on le rend obligatoire, puis on distribue l'obligation à travers l'implication.*