CONCOURS CENTRALE•SUPÉLEC

Informatique

MPI

4 heures

Calculatrice autorisée

2026

Ce sujet traite de la logique *déontique*. Dans cette logique, on ne se demande pas seulement si une propriété est vraie ou fausse, on se demande aussi si elle *devrait* être vraie ou *devrait* être fausse. Ainsi, une propriété peut être fausse alors qu'elle devrait être vraie ou vraie alors qu'elle devrait être fausse. Par exemple : « La couronne de l'impératrice Eugénie a été volée, mais elle n'aurait pas dû être volée ».

Pour modéliser cette notion de « *devoir être vraie* », nous allons considérer plusieurs « mondes ». Il y a le monde réel, et plusieurs mondes idéaux. Une propriété est vraie si elle est vraie dans le monde réel, et on dit qu'une propriété devrait être vraie si elle est vraie dans tous les mondes idéaux. Ainsi, dire « La couronne de l'impératrice Eugénie ne devrait pas être volée » signifie « Dans aucun monde idéal, la couronne de l'impératrice Eugénie n'a été volée ».

Une affirmation qui *devrait* être vraie est dite *obligatoire*.

Une propriété qui n'a pas le devoir d'être fausse, est dite *autorisée*. Ainsi, si l'affirmation « Mon pull est jaune » est vraie dans au moins un monde idéal, alors elle est autorisée. On dira que l'affirmation est *possible* ou qu'elle *peut être vraie*. Une affirmation *autorisée* peut éventuellement être fausse dans un monde idéal, l'essentiel est qu'elle soit vraie dans au moins l'un d'entre eux.

Partie A – Base de données (SQL)

En 2042, la nouvelle présidente de la France, Maryam Turing, décide de lancer une grande réforme des départements. Elle demande à 20 conseillers, numérotés de 1 à 20 issus de bords politiques différents de proposer une réforme.

Elle demande à chaque conseiller quel département doit être conservé, quels nouveaux départements devraient être créés et quelle commune devrait être rattachée à quel département. Chaque conseiller va décrire son *monde idéal*, et expliciter, dans son monde idéal, quels départements existent, et quelle commune appartient à quel département.

Le monde réel sera le monde numéro zéro, et le monde idéal du i-ème conseiller sera le monde numéro i. Il y a donc 21 mondes au total.

On décrit les résultats dans une base de données qui contient plusieurs tables. La première table est la table `commune`. Cette table contient le nom de chaque commune, le numéro INSEE (dans le monde réel) de chaque commune. Ce numéro INSEE est un identifiant unique de la commune et la clef primaire de la table.

Nom	INSEE
Beauregard	1030
Malicorne-sur-Sarthe	72179
Marvejols	48092

Tableau 1 – Extrait de la table `commune`

La seconde table est la table `conseiller`. Elle indique, pour chaque monde, quelle commune est rattachée à quel département. Cette table a 3 colonnes :

- La colonne `INSEE` représente le numéro `INSEE` de la commune.
- La colonne `Département` indique à quel département est rattachée la commune.
- La colonne `Monde` indique dans quel monde l'information de cette ligne est vraie.

Par exemple, dans le monde réel, la commune de numéro 48092 — Marvejols — est en Lozère, mais dans le monde idéal du conseiller numéro 5, elle est dans le département du Gévaudan (département qui n'existe pas dans le monde réel).

On dira d'une proposition sur les communes qu'elle est *obligatoire* si elle est vraie dans tous les mondes idéaux des 20 conseillers (elle peut être fausse dans le monde réel) et qu'elle est *possible* ou *autorisée* si elle est vraie dans au moins un monde idéal d'un conseiller.

Monde	INSEE	Département
0	72179	Sarthe
0	48092	Lozère
5	48092	Gévaudan

Tableau 2 – Extrait de la table `conseiller`

Q1. Déterminer la liste (sans doublon) des numéros des conseillers proposant de créer le département du Gévaudan.

Q2. Déterminer les numéros des conseillers qui proposent que la commune du « Mont-Saint-Michel » soit rattachée au département d'Ille-et-Vilaine.
On pourra utiliser le fait qu'une seule commune porte ce nom.

Q3. Déterminer les communes (numéro INSEE) qui peuvent faire partie de la Sarthe, c'est-à-dire les communes qui, dans au moins un monde idéal, font partie de la Sarthe.

Q4. Déterminer les communes (numéro INSEE) qui peuvent faire partie de la Sarthe mais n'en font pas partie dans le monde réel.

Q5. Déterminer la liste des couples (numéro INSEE, département) tels qu'il est obligatoire que la commune appartienne au département, mais dont ce n'est pas le cas dans le monde réel.
On attend une table avec le numéro INSEE de la commune et le nom du département.
On rappelle qu'une affirmation est obligatoire dès lors qu'elle est vraie dans tous les mondes idéaux, peu importe qu'elle soit vraie dans le monde réel.

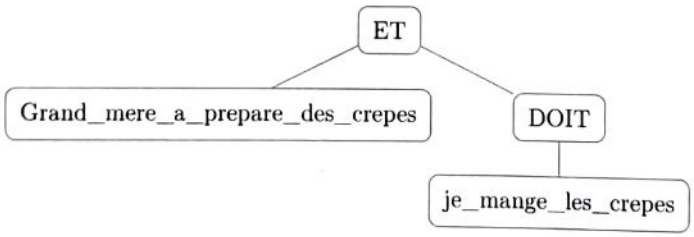
Partie B – Analyse lexicale et syntaxique d'une formule déontique (en OCaml)

L'objectif de cette partie est de construire des formules déontiques, sous forme d'arbres de syntaxe abstrait, à partir d'une représentation textuelle. On considère des formules écrites à l'aide des opérateurs logiques « non », « et », « doit » et de variables propositionnelles. La formule « doit P » signifie que P doit être vraie, c'est-à-dire que P est vraie dans tous les mondes idéaux.

Par exemple, la proposition « Grand-mère a préparé des crêpes, je dois les manger », se formalise par le texte suivant :

« (Grand_mere_a_prepare_des_crepes et doit je_mange_les_crepes) »

Ce qui donne finalement l'arbre :



Pour cela, on commence par s'intéresser au découpage d'une telle représentation en une suite de lexèmes, puis à l'analyse syntaxique de cette suite de lexèmes.

I – Analyse lexicale

L'objectif de cette sous-partie est de découper une chaîne de caractères en « lexèmes ». Les lexèmes sont :

- les opérateurs logiques décrits en texte : « non », « et », « doit » ;
- les deux symboles de parenthèses : « (», «) » ;

- les variables propositionnelles, par exemple « Grand_mere_a_prepare_des_crepes », décrites avec du texte sans aucun espace. On s'autorise les 26 lettres de l'alphabet (en majuscule ou minuscule) ainsi que le caractère spécial _. On impose que ce caractère spécial ne soit pas le premier caractère.

En OCaml, les lexèmes sont représentés par le type suivant :

```
type lexeme =
    Lex_var of string
  | Lex_et
  | Lex_non
  | Lex_doit
  | Lex_PO (* parenthèse ouvrante *)
  | Lex_PF (* parenthèse fermante *)
```

On introduit pour la suite quelques ensembles :

- $\mathcal{E}$ l'ensemble des caractères d'espacement : l'espace simple, la tabulation (`\t`) et le saut de ligne (`\n`) ;
- $\mathcal{A}$ l'ensemble des 26 lettres de l'alphabet (sans accent ni cédille) en majuscule ou en minuscule (soit 52 caractères au total) ;
- $\mathcal{P}$ l'ensemble contenant deux symboles, la parenthèse ouvrante et la parenthèse fermante.

Pour représenter $\mathcal{A}$ et $\mathcal{E}$, on utilise les chaînes de caractères suivantes en OCaml :

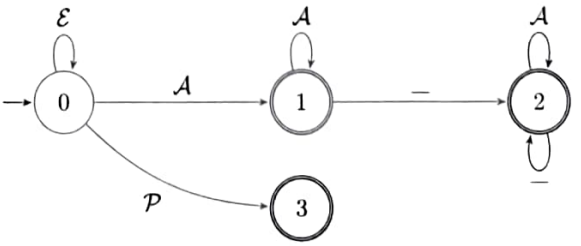
```
let alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
let espaces = " \t\n"
```

Q6. Écrire une fonction `appartient : string -> char -> bool` qui prend en entrée une chaîne de caractères et un caractère et qui teste si le caractère appartient à la chaîne. Cette fonction ne doit pas utiliser `String.contains`.

Q7. Écrire une fonction `sous_chaine : string -> int -> int -> string` qui prend en entrée une chaîne de caractères et deux entiers indiquant le début et la longueur de la sous-chaîne à renvoyer. Cette fonction ne doit pas utiliser `String.sub` et elle doit lever une exception directement ou par l'intermédiaire d'une assertion lorsqu'il n'est pas possible d'extraire une telle sous-chaine.

Par exemple, `sous_chaine "abracadabra" 3 4` doit renvoyer `"acad"` et `sous_chaine "abc" 1 3` est un appel qui lève une exception.

Pour reconnaître les lexèmes, nous utilisons l'automate suivant :



L'état initial est l'état 0. N'importe quel symbole de $\mathcal{E}$ permet de faire une transition de l'état 0 vers l'état 0. Le symbole _ permet de faire une transition de l'état 1 à l'état 2.

Q8. Écrire une fonction `detecte_lexeme : string -> int -> int * int` qui prend en entrée une chaîne de caractères `chaine` et un entier `debut` et qui renvoie un couple d'entiers `(debut_lexeme, fin_lexeme)` tels que :

- la sous chaîne de `chaine`, du caractère d'indice `debut` inclus au caractère d'indice `fin_lexeme` exclu est reconnue par l'automate ;
- il n'existe pas de plus grande sous-chaîne commençant en `debut` qui soit reconnue ;
- `debut_lexeme` est l'indice du premier caractère après l'indice `debut` qui n'est pas dans $\mathcal{E}$.

Cette fonction doit lever l'exception `Erreur_Lexicale` s'il n'existe pas de préfixe de `chaine` reconnu par l'automate. On supposera cette exception déjà définie.

Par exemple, l'expression `detecte_lexeme "non doit manger_crepe" 3` doit renvoyer `(4, 8)`, tout comme l'expression `detecte_lexeme "non doit manger_crepe" 4`.

On définit la fonction `produit_lexeme` comme suit :

```
let produit_lexeme chaine debut fin =
  match sous_chaine chaine debut (fin-debut) with
  | "et" -> Lex_et
  | "non" -> Lex_non
  | "doit" -> Lex_doit
  | "(" -> Lex_PO
  | ")" -> Lex_PF
  | x -> Lex_var x
```

Cette fonction prend en entrée une chaîne de caractères et les positions d'un lexème identifié grâce à la fonction `detecte_lexeme` et renvoie le lexème correspondant.

Q9. Écrire une fonction `analyse_lexicale : string -> lexeme list` qui prend en entrée une chaîne de caractères et qui renvoie en sortie sa décomposition en suite de lexèmes.

Cette fonction doit lever l'exception `Erreur_Lexicale` s'il n'est pas possible de convertir la chaîne de caractères en suite de lexèmes.

Par exemple, `analyse_lexicale "non doit manger_crepe"` doit renvoyer : `[Lex_non; Lex_doit; Lex_var "manger_crepe"]`.

II – Grammaire non contextuelle

Afin de procéder à l'analyse *syntaxique*, on souhaite définir une grammaire non contextuelle permettant de reconnaitre les formules déontiques.

Les symboles terminaux considérés sont les mots-clés « et », « non », « doit », ainsi que les parenthèses « (» et «) ». Les variables propositionnelles, par exemple « manger_crepes », sont représentées dans la grammaire par le symbole terminal générique n.

Les symboles A, E, B et C représentent des symboles non terminaux.

On définit une première grammaire, $\mathcal{G}$, de symbole initial A, et ayant les règles suivantes :

(i) $A \to (\ E\ ) \mid E$

(ii) $E \to n \mid A\ et\ A \mid non\ A \mid doit\ A$

On définit une seconde grammaire, $\mathcal{H}$, de symbole initial E, et ayant les règles suivantes :

(i) $E \to n \mid (\ B\ ) \mid C$

(ii) $B \to E\ et\ E$

(iii) $C \to non\ E \mid doit\ E$

Q10. Considérons les mots m_1 := « x doit (y et z) » et m_2 := « x et doit (non y et z) ». Pour chacun de ces mots, indiquez s'il dérive :

- de la grammaire $\mathcal{G}$;
- de la grammaire $\mathcal{H}$.

Q11. Les grammaires $\mathcal{G}$ et $\mathcal{H}$ engendrent-elles le même langage ? Justifiez votre réponse.

Q12. La grammaire $\mathcal{G}$ est-elle ambiguë ? La grammaire $\mathcal{H}$ est-elle ambiguë ? Justifiez votre réponse (on ne demande pas de preuve formelle dans le cas d'une non-ambiguïté, mais une brève justification).

Dans la suite de cette partie, nous considérons uniquement la grammaire $\mathcal{H}$.

III – Analyse syntaxique

Le principe de l'analyse syntaxique est de transformer une liste de lexèmes en un arbre syntaxique. On souhaite trouver l'arbre syntaxique d'une formule pour la grammaire $\mathcal{H}$.

Cet arbre syntaxique abstrait est représenté en OCaml par le type `formule` défini ci-après :

```
type formule =
    Var of string
  | Et of formule * formule
  | Non of formule
  | Doit of formule
```

Plus précisément, on souhaite créer plusieurs fonctions mutuellement récursives, une par symbole non terminal : `derive_E`, `derive_B`, `derive_C`. Chacune de ces fonctions prend en entrée une liste de lexèmes `lexemes`, et renvoie un couple `(f, reste)` constitué ainsi :

- `f` est un arbre syntaxique correspondant à un préfixe `prefixe` de la liste `lexemes`.
- `reste` est une liste telle que `prefixe @ reste = lexemes` ; ainsi `reste` est la liste `lexemes` à laquelle on a retiré la partie `prefixe` qui a servi à générer `f`.

Voici le code incomplet de ces fonctions.

```
exception Erreur_Syntaxique (* ceci définit une nouvelle exception *)

let rec derive_E lexemes = match lexemes with
  | Lex_var x :: q -> (Var x, q);  (* E -> n *)
  | Lex_PO :: q ->  (* E-> ( B ) *)
    begin
      match derive_B q with
        | f , Lex_PF :: q2 -> (f, q2)
        | _ -> raise Erreur_Syntaxique
    end
  | _ -> derive_C lexemes  (*  E -> C *)

and derive_B lexemes = match derive_E lexemes with
    | f1, Lex_et::q ->  (* B -> E et E *)
      begin
        (* bloc de code 1  *)
      end
    | _ -> raise Erreur_Syntaxique

and derive_C lexemes = match lexemes with
  (* bloc de code 2 *)
```

Par exemple, `derive_E [Lex_non; Lex_var "manger_crepe"; Lex_et; Lex_doit; Lex_var "remercier_mamie"; Lex_PF]` doit renvoyer :

```
(Non (Var "manger_crepe"), [Lex_et; Lex_doit; Lex_var "remercier_mamie"; Lex_PF]).
```

Q13. Compléter le « `bloc de code 1` » situé à la ligne 16 du code ci-dessus.

Q14. Compléter le « `bloc de code 2` » situé à la ligne 21 du code ci-dessus.

Q15. Écrire une fonction `analyse_syntaxique : lexeme list -> formule` qui prend en entrée une liste de lexèmes et qui renvoie, si possible, l'arbre syntaxique qui permet de générer cette liste de lexèmes et lève l'exception `Erreur_Syntaxique` sinon.

Partie C – Univers de Kripke (en C)

Dans cette partie, on redéfinit les formules déontiques avec un plus grand nombre d'opérateurs, et on donne des modèles aux formules afin d'étudier la satisfiabilité des formules déontiques.

I – Formules déontiques

On suppose disposer d'un ensemble $\Delta = \{X_0, \ldots, X_n, \ldots\}$ de variables propositionnelles. On définit par induction une *formule déontique* comme suit :

(i) Si P est une variable propositionnelle, alors P est une formule déontique.

(ii) Si φ et ψ sont des formules déontiques alors $(\varphi \wedge \psi)$ et $(\varphi \vee \psi)$ et $(\varphi \to \psi)$ sont des formules déontiques.

(iii) Si φ est une formule déontique alors $\neg\varphi$, $\Box\varphi$ et $\Diamond\varphi$ sont des formules déontiques.

$\Box\varphi$ se lit « φ est obligatoire » et $\Diamond\varphi$ se lit « φ est autorisée ».

Ces formules sont une variante des formules définies dans la partie précédente.

Pour représenter une formule en C, nous introduisons des constantes pour les opérateurs.

```
const int ET = 0;
const int OU = 1;
const int NON = 2;
const int VARIABLE = 3;
const int DOIT = 4;
const int PEUT = 5;
const int IMPLIQUE = 6;
```

Ce code définit les constantes `ET`, `OU`, `NON`, `VARIABLE`, `DOIT`, `PEUT` et `IMPLIQUE`, qui sont de type `int`. Ces constantes représentent respectivement les opérateurs $\wedge$, $\vee$, $\neg$, les variables, $\Box$, $\Diamond$ et $\rightarrow$.

Une formule sera représentée par le type `formule` défini ci-dessous :

```
struct formule_s {
  int op;
  struct formule_s *gauche;
  struct formule_s *droite;
  int variable;
};

typedef struct formule_s formule ;
```

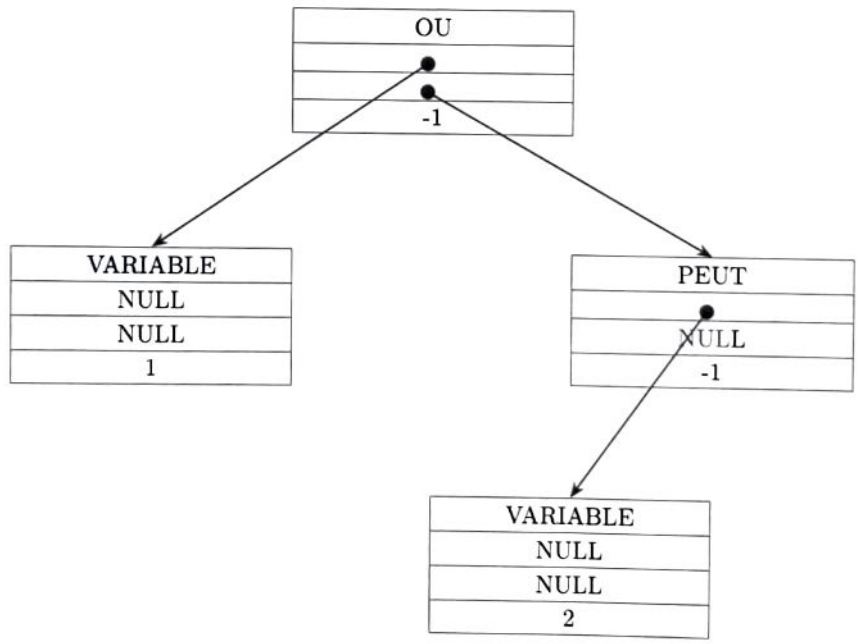
Le champ opérateur indique de quel opérateur il s'agit.

Si c'est un opérateur binaire ($\wedge$, $\vee$, $\rightarrow$), les champs `gauche` et `droite` indiquent les deux sous-formules tandis que le champ `variable` est affecté d'une valeur entière arbitraire (par exemple -1).

Si c'est un opérateur unaire, le champ `gauche` indique la sous-formule, le champ `droite` vaut `NULL` et le champ `variable` est affecté d'une valeur entière arbitraire (par exemple -1).

Si c'est une variable, gauche et droite valent `NULL`, et `variable` indique le numéro de la variable. Par exemple, si ce champ vaut 42, alors la variable est X_{42}.

Par exemple, la formule $X_1 \vee \Diamond X_2$ sera représentée par la structure suivante.



Q16. Écrire une fonction `formule *cree_formule(int op, int variable, formule *fg, formule *fd)` qui alloue sur le tas et renvoie une valeur du type `formule` avec les valeurs spécifiées pour ses champs.

Q17. Écrire une fonction `int compter_peut (formule *f)` qui prend en entrée une formule `f` et qui renvoie le nombre d'occurrences de l'opérateur $\Diamond$ dans cette formule.

Par exemple, sur la formule $X_1 \vee \Diamond X_2$ décrite précédemment, cette fonction doit renvoyer 1.

Q18. Écrire une fonction `void free_formule(formule *f)` qui prend en entrée une forme `f` et qui libère la mémoire occupée par cette formule. La fonction `free_formule` doit libérer la mémoire de toutes les sous-formules.

Par exemple, sur la formule $X_1 \vee \Diamond X_2$ décrite précédemment, cette fonction doit libérer 4 structures.

Pour simplifier l'étude des formules, on souhaite retirer toutes les implications, plus précisément, on souhaite remplacer les $\varphi \to \psi$ par $\neg\varphi \vee \psi$.

Q19. Écrire une fonction `formule *retirer_implications(formule *f)` qui prend en entrée une formule `f` et qui renvoie en sortie une formule `g` telle que :

- `g` est obtenue en remplaçant toutes les implications $\to$ de `f` ;
- la formule `f` n'a pas été modifiée ;
- aucune des sous-formules de `f` et de `g` ne partagent un même emplacement en mémoire.

II – Modèles d'une formule déontique

Pour interpréter une formule déontique, nous allons devoir formaliser la notion de « monde » présentée dans l'introduction du sujet. Intuitivement, dans un monde, chaque fait élémentaire est soit vrai, soit faux. Par exemple, dans un monde donné, soit la couronne de l'impératrice a été volée, soit elle n'a pas été volée.

Plus formellement, on appelle *monde* une valuation des variables propositionnelles, c'est-à-dire une fonction de l'ensemble des variables propositionnelles vers l'ensemble {**Vrai**,**Faux**}.

Pour représenter un monde en C, nous utiliserons le type `monde` suivant :

```
1 struct monde_s {
2   int taille;
3   bool *valuation;
4 };
5
6 typedef struct monde_s monde ;
```

Chaque structure `monde` en C ne va représenter qu'un nombre fini de variables.

`taille` indique le nombre de variables représentées.

`valuation` est un pointeur vers un tableau de booléens tel que `valuation[i]` soit le booléen associé à la valeur de vérité, **Vrai** ou **Faux**, de X_i.

Q20. Écrire une fonction `bool satisfaction(monde *m, formule *f)` qui prend en entrée une formule ne contenant que des variables propositionnelles, des $\wedge$ et des négations et qui renvoie `true` si la formule `f` est satisfaite dans le monde `m` et `false` sinon.

Un monde ne permet que d'interpréter une formule propositionnelle. Pour interpréter une formule déontique, nous avons besoin de plusieurs mondes. Par exemple, pour interpréter $\neg X_0 \wedge \Box X_0$, nous avons besoin de vérifier si X_0 est fausse dans le « monde réel » et X_0 est vraie dans tous les mondes idéaux. De surcroît, nous avons besoin de pouvoir interpréter des formules plus complexes, par exemple de la forme $\Box\Diamond X_1$. Pour interpréter $\Box\Diamond X_1$, on doit regarder si la formule $\Diamond X_1$ est vraie dans tous les mondes idéaux. Mais $\Diamond X_1$, signifie intuitivement qu'il existe au moins un monde idéal dans lequel X_1 est vraie.

Nous avons donc besoin d'avoir des mondes qui sont idéaux pour chaque monde idéal. Autrement dit, nous avons besoin d'une relation entre les mondes qui dise « le monde j est idéal pour le monde i ». Cela nous amène tout naturellement à construire un graphe de mondes, dans lequel un arc de i vers j signifie « le monde j est idéal pour le monde i ».

Plus formellement, on appelle *univers* un graphe de mondes. Plus précisément, un univers $\mathbb{U}$ est représenté par un couple $(S_\mathbb{U}, R_\mathbb{U})$, où :

- $S_\mathbb{U}$ est un ensemble fini non vide de mondes, c'est l'ensemble des sommets du graphe.
- $R_\mathbb{U}$ est une relation binaire. $R_\mathbb{U}(m_1,m_2)$ signifie que dans l'univers $\mathbb{U}$ il existe un arc du monde m_1 vers le monde m_2. On impose que, pour tout monde $m_1 \in S_\mathbb{U}$, il existe un monde $m_2 \in S_\mathbb{U}$ tel que $R_\mathbb{U}(m_1,m_2)$.

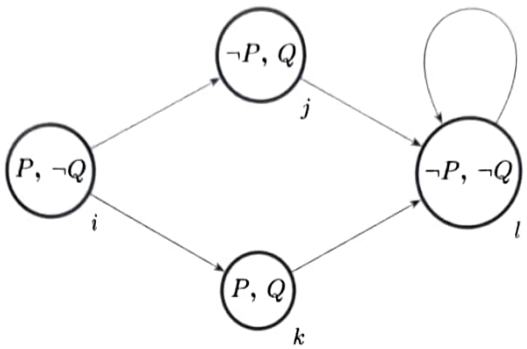
L'idée d'un univers est que la notion de monde idéal est relative : un monde j peut être idéal pour les habitants d'un monde i ; mais les habitants du monde j peuvent avoir des idéaux différents, et avoir pour monde idéal un autre monde k.

On définit inductivement le fait qu'un univers $\mathbb{U}$ satisfait une formule φ en un monde i par :

- $\mathbb{U} \vDash_i A$ si A est une variable propositionnelle associée à la valeur de vérité vraie dans le monde i.
- $\mathbb{U} \vDash_i \varphi \wedge \psi$ si $\mathbb{U} \vDash_i \varphi$ et $\mathbb{U} \vDash_i \psi$.

- $\mathbb{U} \vDash_i \varphi \vee \psi$ si $\mathbb{U} \vDash_i \varphi$ ou $\mathbb{U} \vDash_i \psi$.
- $\mathbb{U} \vDash_i \varphi \to \psi$ si $\mathbb{U} \vDash_i \neg\varphi \vee \psi$.
- $\mathbb{U} \vDash_i \neg\varphi$ si $\mathbb{U} \vDash_i \varphi$ est faux.
- $\mathbb{U} \vDash_i \Box\varphi$ si pour tout monde $j \in S_\mathbb{U}$ tel que $R_\mathbb{U}(i,j)$ on a $\mathbb{U} \vDash_j \varphi$.
- $\mathbb{U} \vDash_i \Diamond\varphi$ s'il existe un monde $j \in S_\mathbb{U}$ tel que $R_\mathbb{U}(i,j)$ on a $\mathbb{U} \vDash_j \varphi$.

Voici un exemple d'univers $\mathbb{E}$ qui satisfait la propriété $\neg Q \wedge \Box Q$ (« Q est faux mais Q devrait être vrai ») dans le monde i. Par souci de lisibilité du dessin, un monde (c'est-à-dire une interprétation) associant P à *true*, Q à *false*, etc sera codé par : P, $\neg Q$, etc. Par exemple, P est faux et Q est vrai dans le monde j, tandis que P et Q sont tous les deux vrais dans le monde k.



Q21. Cet univers est-il un modèle de la formule $\Box\Box Q$ dans le monde i ? Autrement dit, a-t-on $\mathbb{E} \vDash_i \Box\Box Q$?

Q22. Cet univers est-il un modèle de la formule $\Diamond(P \wedge \Box\neg P)$ dans le monde i ?

On dit qu'une formule φ est satisfiable s'il existe un univers $\mathbb{U}$ et un monde m tel que $\mathbb{U} \vDash_m \varphi$. On dit alors que le couple $(\mathbb{U}, m)$ est un modèle de φ.

On dit qu'une formule φ est la conséquence logique d'une formule ψ si tout modèle de ψ est un modèle de φ.

On considère que $\Box$ est prioritaire sur $\to$, ainsi $\Box P \to \Box Q$ est la formule $(\Box P) \to (\Box Q)$.

On définit les formules $\theta_1 = \Box\Box P \to \Box P$ et $\theta_2 = \Box(\Box P \to P)$.

Q23. Montrer que la formule θ_1 est satisfiable et que sa négation est satisfiable.

Q24. Montrer que θ_2 et sa négation sont satisfiables.

Q25. Montrer que θ_1 est la conséquence logique de θ_2 mais que θ_2 n'est pas la conséquence logique de θ_1.

On dit que deux formules sont logiquement équivalentes si elles ont exactement les mêmes modèles.

III – Formules réduites

On définit par induction une formule *réduite* comme suit :

(i) Si P est une variable propositionnelle alors P et $\neg P$ sont des formules réduites.

(ii) Si φ et ψ sont des formules réduites alors $(\varphi \wedge \psi)$ et $(\varphi \vee \psi)$ sont des formules réduites.

(iii) Si φ est une formule réduite alors $\Box\varphi$ et $\Diamond\varphi$ sont des formules réduites.

La principale différence entre les formules déontiques et les formules réduites est la place de la négation. Dans les formules réduites, les négations ne peuvent apparaître que devant les variables propositionnelles (il n'y a pas non plus d'implication dans les formules réduites).

On remarque que $\neg\Box\neg\varphi$ est logiquement équivalente à $\Diamond\varphi$, quelle que soit la formule déontique φ.

Q26. Rappeler les règles de De Morgan.

Q27. Montrer que toute formule déontique est logiquement équivalente à une formule réduite.

Q28. Écrire une fonction `bool est_reduite(formule *f)` qui prend en entrée une formule `f` et qui renvoie un booléen indiquant si la formule est réduite.

Un univers sera représenté par la structure suivante :

```
struct univers_s {
  int nb_mondes;
  bool **matrice;
  monde *mondes;
};

typedef struct univers_s univers;
```

Dans cette structure `nb_mondes` indique le nombre de mondes de l'univers, `matrice` est la matrice d'adjacence sous la forme d'un tableau bidimensionnel et `mondes` est le tableau des mondes. Ainsi, si `u` est un univers et `i`, `j` deux indices de mondes dans cet univers, `u.matrice[i][j]` vaut `true` si et seulement si `u.monde[j]` est idéal pour `u.monde[i]`.

Q29. Écrire une fonction `bool satisfaction2(univers *u, int i, formule *f)` qui renvoie vrai si le monde `i` de l'univers `U` satisfait la formule **réduite** `f`.

IV – Décidabilité de la satisfiabilité d'une formule déontique

L'objectif de cette section est de montrer que le problème de savoir si une formule déontique est satisfiable est décidable. Dans toute cette section, nous ne nous soucierons pas de la complexité.

On définit par récurrence un univers arborescent de profondeur n et de racine r par :

- Un univers arborescent de profondeur 0 est constitué d'un unique monde, et d'un unique arc de ce monde vers lui-même. Ce monde est la racine.
- Si $\mathbb{U}_1, \ldots, \mathbb{U}_k$ sont des univers arborescents deux à deux disjoints de profondeur n, de racines respectives $r_1, \ldots, r_k$ et que r est un monde qui n'appartient à aucun de ces univers, alors l'univers $\mathbb{U}$ défini comme suit est un univers arborescent de profondeur $n+1$:
  - les sommets de $\mathbb{U}$ sont la réunion des sommets de $\mathbb{U}_1, \ldots, \mathbb{U}_k$ à laquelle on ajoute le sommet r ;
  - les arcs de $\mathbb{U}$ est la réunion des arcs de $\mathbb{U}_1, \ldots, \mathbb{U}_k$ à laquelle on ajoute les arcs $(r,r_1), (r,r_2), \ldots, (r,r_k)$.

Intuitivement, un univers arborescent est un arbre.

L'algorithme suivant crée un univers arborescent $\mathbb{A}$ de profondeur N à partir d'un univers $\mathbb{U}$, d'un entier N et d'un monde i.

- arborescent(U, 0, i) :
  - Créer une nouvelle copie nommée r de i.
  - Créer un arc de r vers r.
  - Renvoyer le graphe ainsi obtenu.
- arborescent(U, N, i) :
  - Créer une nouvelle copie nommée r de i.
  - Pour chaque arc sortant (i,j) de i, créer A_j = arborescent(U, N-1, j).
  - Faire l'union des A_j puis rajouter le monde r puis, pour chaque racine r_j des A_j, créer un arc (r,r_j).

Q30. Écrire une fonction `int taille(univers *u, int i, int N)` qui prend en entrée un univers `u`, un numéro de monde `i` dans cet univers ainsi qu'un entier `N` et qui renvoie la taille, en nombre de mondes, qu'aurait un univers arborescent créé par l'algorithme précédent.

Q31. Écrire une fonction `univers *arborescent(univers *u, int i, int N)` qui implémente l'algorithme précédent. La racine du résultat doit être le monde numéro zéro.

Q32. Soient φ une formule réduite, N est le nombre de symboles $\Diamond$ et $\Box$ de la formule φ, $\mathbb{U}$ un univers, i un monde de $\mathbb{U}$ et $\mathbb{A}$ l'univers arborescent de racine r créé par l'algorithme précédent (autrement dit $\mathbb{A}$ = arborescent$(\mathbb{U}, N, i)$). Montrer que si $\mathbb{U} \vDash_i \varphi$ alors $\mathbb{A} \vDash_r \varphi$.

Q33. Montrer que la proposition précédente reste vraie si on modifie l'algorithme et qu'on ne garde pas dans les $\mathbb{U}_i$ deux univers identiques lors de la construction.

Q34. Montrer que le problème de la satisfiabilité d'une formule déontique est décidable.

V – NP-complétude

Dans cette section nous nous intéressons à nouveau à la complexité. L'objectif est de montrer qu'un problème plus simple que la satisfiabilité d'une formule déontique est NP-complet.

On note DSAT le problème de la satisfiabilité d'une formule déontique.

Q35. Montrer que DSAT est NP-complet si et seulement si DSAT est dans NP.

Dans la mesure où il a été prouvé que DSAT est complet pour une classe de complexité plus grande, on conjecture qu'il n'est pas dans NP. On s'intéresse alors à un sous-problème nommé TOTAL-DSAT.

On définit un univers *total* comme étant un univers dans lequel tout monde est idéal pour tout autre monde. Autrement dit, un univers $\mathbb{U}$ est total si, pour tous mondes i et j de $\mathbb{U}$, il existe un arc de i vers j (même dans le cas particulier où $i = j$).

On dit qu'une formule φ est satisfiable dans un univers total $\mathbb{U}$ s'il existe un monde i de $\mathbb{U}$ tel que $\mathbb{U} \vDash_i \varphi$.

Q36. Montrer qu'il existe une formule déontique satisfiable qui n'est satisfiable dans aucun univers total.

Q37. Soit φ une formule réduite, $\mathbb{U}$ un univers total et i un monde de $\mathbb{U}$, justifier qu'il est possible de vérifier $\mathbb{U} \vDash_i \varphi$ en $O(N|\varphi|)$ où N est le nombre de mondes de $\mathbb{U}$ et $|\varphi|$ la taille de φ en tant qu'arbre de formule.

Un univers $\mathbb{T}$ est un témoignage de la formule φ dans l'univers $\mathbb{U}$ si :

- $\mathbb{T}$ est un sous-graphe de $\mathbb{U}$;
- $\mathbb{T}$ est total ;
- pour toute sous-formule $\Diamond\psi$ de φ, s'il existe un ou plusieurs mondes k tels que $\mathbb{U} \vDash_k \psi$, alors au moins un de ces mondes est dans $\mathbb{T}$.

Q38. Soit φ une formule réduite et $\mathbb{U}$ un univers total. Montrer que, si i est un monde de $\mathbb{U}$ tel que $\mathbb{U} \vDash_i \varphi$, alors il existe un témoignage $\mathbb{T}$ de φ contenant i tel que $\mathbb{T} \vDash_i \varphi$ et $\mathbb{T}$ est de taille, en nombre de mondes, polynomiale en la taille de φ.

On s'intéresse au problème TOTAL-DSAT qui prend en entrée une formule déontique et qui renvoie **Vrai** s'il existe un univers total $\mathbb{B}$ et un monde i de $\mathbb{B}$ tels que $\mathbb{B} \vDash_i \varphi$ et qui renvoie **Faux** sinon.

Q39. Montrer que TOTAL-DSAT est NP-complet.

Partie D – Déduction naturelle pour la logique déontique

On rappelle dans la figure 1 l'ensemble des règles d'inférence de la déduction naturelle classique.

Q40. Donner un arbre de preuve pour le séquent $\vdash A \to (B \to (A \wedge B))$.

Pour modéliser l'obligation, on étend les règles de la figure 1 en ajoutant trois nouvelles règles spécifiques à l'opérateur $\Box$. Ce système étendu est noté $\mathcal{D}$.

$$\frac{\vdash \varphi}{\vdash \Box\varphi}\ \text{D1} \qquad \frac{\Gamma \vdash \Box(\varphi \to \psi)}{\Gamma \vdash \Box\varphi \to \Box\psi}\ \text{D2} \qquad \frac{\Gamma \vdash \Box\varphi}{\Gamma \vdash \neg\Box\neg\varphi}\ \text{D3}$$

Dans la règle D1, on notera l'absence de contexte à gauche de $\vdash$. L'hypothèse de la règle $\vdash \varphi$ est prouvable à partir de rien, c'est-à-dire que φ est un théorème. La règle signifie que dans un monde idéal, les théorèmes sont vrais.

Q41. Montrer que le système $\mathcal{D}$ empêche les obligations contradictoires. Pour cela, prouver le séquent suivant $\vdash \neg(\Box A \wedge \Box\neg A)$.

Q42. Prouver à l'aide de ce système de déduction l'équivalence $\Box(A \wedge B) \leftrightarrow (\Box A \wedge \Box B)$ en donnant des arbres de preuve pour les séquents $\vdash \Box(A \wedge B) \to (\Box A \wedge \Box B)$ et $\vdash (\Box A \wedge \Box B) \to \Box(A \wedge B)$. On pourra utiliser le résultat de la question **Q40** sans reproduire l'arbre de preuve.

Règles structurelles

$$\frac{}{\Gamma, A \vdash A}\ \text{Ax} \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{Aff}$$

Conjonction ($\wedge$) et Disjonction ($\vee$)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \text{I}\wedge \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \text{E}_1\wedge \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \text{E}_2\wedge$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \text{I}_1\vee \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \text{I}_2\vee \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \text{E}\vee$$

Implication ($\rightarrow$)

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \text{I}\rightarrow \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \text{E}\rightarrow$$

Négation ($\neg$) et Absurde ($\bot$)

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \text{I}\neg \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \text{E}\neg \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{RAA} \qquad \frac{}{\Gamma \vdash A \vee \neg A}\ \text{TE}$$

Figure 1 – Règles de la déduction naturelle classique

◇ Fin ◇