Ce TP traite de la logique propositionnelle (logique d'ordre zéro), en OCaml.

> **Ce TP est volontairement assez long, je vous encourage à allouer du temps pendant l'été pour le terminer. Vous pourrez m'écrire pour me montrer ce que vous aurez fait, n'hésitez pas !**

On travaillera en OCaml, depuis la machine virtuelle ClefAgreg2019 et Emacs, ou depuis `https://BetterOCaml.ml`.

Syntaxe des formules de la logique propositionnelle en OCaml

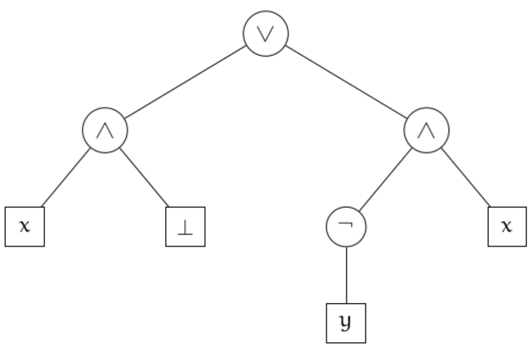
Dans tout le sujet, l'ensemble des variables propositionnelles $\mathcal{V}$ est $\{x_1, x_2, \ldots\} = \{x_i : i \in \mathbb{N}^*\}$.

Représentation

Comme en cours, on propose le type (récursif) suivant pour représenter les formules en OCaml :

```
type formule =
    | Vrai | Faux              (* constantes True et Bottom *)
    | Var of int               (* représente la variable x_i de numéro i >= 1 *)
    | Non of formule           (* Non(P) représente ¬ P *)
    | Et of formule * formule  (* Et(P,Q) représente P ^ Q *)
    | Ou of formule * formule  (* Ou(P,Q) représente P v Q *)
;;
```

1. Recopier ce type dans votre fichier `TP26.ml`.
2. Définir au moins quatre constantes `x1 = Var(1)`, ..., `x4 = Var(4)`, pour facilement écrire des formules dépendant de ces quelques variables.
3. Imaginons que l'on veuille représenter en OCaml la formule P représentée dans la figure ci-dessous (sous forme de son arbre de syntaxe). Expliquer quelle variable `x1`, ..., `x4` représentera les variables x et y, et définir une formule `formule_P` (de type `formule`) la représentant.



4. Quelles sont les hauteur $h(P)$ et $t(P)$ de cette formule P représentée par `formule_P` ?

Premières fonctions : hauteur et taille d'une formule

5. On rappelle que les fonctions de hauteur et taille d'une formule (notée $h(P)$ et $t(P)$) sont définies par induction structurelle sur les formules. Rappeler les définitions.
6. En déduire deux fonctions récursives `hauteur : formule -> int` et `taille : formule -> int`.
7. Tester les sur la formule `formule_P` et vérifier les résultats de la question 4.
8. Quelles sont les complexités temporelles de ces deux fonctions, en fonction des valeurs de $h(P)$ et/ou $t(P)$?

Ensemble des variables d'une formule ($\mathcal{V}(P)$)

On rappelle que pour une formule P, l'ensemble (fini) $\mathcal{V}(P)$ est l'ensemble des variables $x \in \mathcal{V}$ qui apparaissent dans P.

On va représenter un tel ensemble par l'idée la plus simple : une `int list` donnant les indices des x_i apparaissant dans P, dans un ordre quelconque.

9. Pour la formule P de l'exemple, donner $\mathcal{V}(P)$ ainsi que sa représentation en OCaml.
10. Sans chercher à renvoyer une `int list` qui n'auraient pas de doublons, écrire une fonction (récursive) `variables_dans : formule -> int list` qui renvoie tous les indices des variables x_i apparaissant dans une formule donnée. Tester la sur `formule_P`, on peut obtenir par exemple `[1; 2; 1]` (l'ordre n'importe pas).
11. Écrire une fonction `supprime_doublons : int list -> int list` qui supprime les doublons d'une liste d'entiers dans $\mathbb{N}^*$.
    - Par exemple `supprime_doublons [1;2;1]` renvoie `[1;2]` (l'ordre n'importe pas non plus).
    - On pourra commencer par écrire une solution naïve qui soit en temps quadratique ou $\mathcal{O}(n \log(n))$, puis chercher une solution plus efficace qui soit linéaire.
    - *Indication* : on sait que les entiers présents dans la liste sont bornés inférieurement par $i_0 = 1$, et en nombre fini, donc on peut trouver en temps linéaire la valeur la plus grande i_1. Quitte à allouer un tableau de taille i_1, on peut ensuite parcourir une seule fois la liste et compter le nombre d'occurrences de chaque valeur (en temps linéaire). Une fois cet "histogramme" obtenu, on peut le parcourir (encore en temps linéaire) pour en déduire la liste des valeurs présentes dans la liste initiale.

Affichage préfixe (quasi sérialisation)

On cherche à afficher une formule sous la forme suivante, qui permet de la redéfinir si on le copie-colle dans une console ou un fichier OCaml :

```
# affiche_prefixe formule_P;;
Ou (Et (Var 1, Faux), Et (Non (Var 2), Var 1))
- : unit = ()
```

12. Écrire une fonction (récursive) `repr_prefixe : formule -> string` qui construit une chaîne de caractères représentant en notation préfixe la formule donnée en argument. On

se rappellera que l'on peut concaténer deux chaînes avec `chaine1 ^ chaine2` (accent circonflexe).

13. En déduire une fonction `affiche_prefixe : formule -> unit` qui affiche la formule sur une ligne à part. On peut utiliser `print_endline (repr_prefixe p)` ou `Printf.printf "%s\n" (repr_prefixe p)`, au choix. Tester la sur `formule_P`.

Affichage infixe

On cherche à afficher une formule sous la forme infixe, qui correspond à la notation "naturelle" que l'on utilise généralement.

14. Donner (au brouillon) la notation infixe de la formule P de l'exemple.
15. Écrire une fonction (récursive) `repr_infixe : formule -> string` qui construit une chaîne de caractères représentant en notation infixe la formule donnée en argument.
16. En déduire une fonction `affiche_infixe : formule -> unit` qui affiche la formule sur une ligne à part. Tester la sur `formule_P`.

Substitutions $P[Q/x]$

La substitution $P[Q/x]$ est définie par induction structurelle sur la formule P. L'idée est simple : on propage la substitution à l'intérieur de l'arbre de syntaxe sans modifier les connecteurs unaire et binaires (e.g. $(P_1 \wedge P_2)[Q/x] = P_1[Q/x] \wedge P_2[Q/x]$) et aux feuilles on ne touche pas les constantes ni les variables $y \neq x$, mais on remplace les occurrences de la variable x par la formule Q.

17. Soit $Q = x_2 \wedge (x_3 \vee \neg x_2)$, donner au brouillon la formule $P[Q/x_1]$.
18. Écrire une fonction récursive (sur son premier argument) `substitution : formule -> int -> formule -> formule` telle que `substitution formule_P i formule_Q` donne la formule $P[Q/x_i]$.
19. Vérifier le résultat donné en Q17.

Sémantique des formules en OCaml

20. Écrire les trois fonctions suivantes : `non : bool -> bool`, `et : bool -> bool -> bool`, `ou : bool -> bool -> bool`.

Valuations ϕ : donner une valeur de vérité aux variables

Comme on a fixé l'ensemble $\mathcal{V}$ des variables à $\{x_i : i \in \mathbb{N}^*\}$, on peut représenter une valuation $\phi : \mathcal{V} \to \mathbb{B}$ par une fonction de type `int -> bool`, ou un tableau `bool array`, indicé de `0` à `n+1`, avec l'indice 0 inutilisé et l'indice i (pour $1 \leqslant i \leqslant n$) utilisé pour stocker la valeur booléenne de $\phi(x_i) \in \mathbb{B} = \{\texttt{false}, \texttt{true}\}$.

21. Donner au brouillon une valuation ϕ^+ telle que $\overline{\phi^+}$ évalue la formule P à `true`. De même pour ϕ^- qui évalue la formule P à `false`.

22. Définir des fonctions `phi_plus` et `phi_moins` de signature `int -> bool`, représentant en OCaml ces deux valuations. On pourra utiliser un `match i with ...` terminant par le motif `_ -> false` pour les cas des variables x_i n'apparaissant pas dans $\mathcal{V}(P)$.
23. Même question avec la représentation sous forme de tableau.

Évaluation $\overline{\phi}$ d'une formule

Pour une valuation $\phi : \mathcal{V} \to \mathbb{B}$ fixée, on définit la fonction d'évaluation $\overline{\phi}$ associée, qui évalue une formule P en un booléen, par induction structurelle sur P.

24. Écrire une fonction (récursive) `evaluation : (bool array) -> formule -> bool` telle que `evaluation phi formule_P` calcule $\overline{\phi}(P)$.
25. Quelle est sa complexité temporelle, en fonction de $h(P)$ et/ou $t(P)$?

Affichage d'une table de vérité

On souhaite pouvoir afficher une table de vérité d'une formule, sous la forme suivante (par exemple) :

```
| x1 | x2 | x1 v x2
| false | false | false
| false | true | true
| true | false | true
| true | true | true
```

26. Écrire une fonction récursive `toutes_les_valuations : int -> (bool list) list` qui donne (sous forme de `bool list`) toutes les valuations possibles de n variables. Voici les valeurs obtenues pour $n = 0$, 1, 2 et 3 :

```
toutes_les_valuations 0 = []
toutes_les_valuations 1 = [[false]; [true]]
toutes_les_valuations 2 = [
    [false; false]; [false; true];
    [true; false]; [true; true]
]
toutes_les_valuations 3 = [
    [false; false; false]; [false; false; true];
    [false; true; false]; [false; true; true];
    [true; false; false]; [true; false; true];
    [true; true; false]; [true; true; true]
]
```

L'algorithme pour `toutes_les_valuations` est assez simple : les cas de base pour $n = 0$ et 1 sont faciles, puis le cas récursif peut être traité en calculant d'abord `valuations_nM1` toutes les valuations pour $n-1$ variables, puis `valuations_n_F = List.map (fun phi -> false :: phi) valuations_nM1` donne toutes les valuations de n variables telles que la première variable soit évaluée à `false`. En faisant de même pour `true` en définissant `valuations_n_V`, on termine en renvoyant `valuations_n_F @ valuations_n_V`.

Cet algorithme est évidemment très (trop !) coûteux quand n devient grand, car il y a 2^n valuations à renvoyer.

27. En utilisant `Array.of_list : 'a list -> 'a array`, on peut convertir un tableau en une liste (dans le même ordre). Écrire une fonction `val_tab_of_list : bool list -> bool array` qui se contente de rajouter un `false` (inutilisé) en tête de la liste, puis de la transformer en tableau.
28. En déduire une fonction `affiche_table_verite : formule -> unit` qui affiche la table de vérité au format décrit ci-dessus. On pourra utiliser `List.iter` pour itérer sur la liste des valuations donnée par `toutes_les_valuations`.
29. Tester la pour `formule_P` de l'exemple, on devrait obtenir cela :

```
| x1 | x2 | ((x1 ^ Faux) v (¬x2 ^ x1))
| false | false | false
| false | true | false
| true | false | true
| true | true | false
```

Nature d'une formule : tautologique, satisfiable ou insatisfiable

On rappelle qu'une formule P est dite *tautologique* si toute valuation ϕ donne $\overline{\phi}(P) = V$, satisfiable s'il existe au moins une valuation évaluant P à vrai, et insatisfiable sinon.

30. Proposer un type énumération `nature_formule` en OCaml pour représenter la nature d'une formule, qui peut être `Tautologique`, `Satisfiable` (signifiant qu'elle est satisfiable mais pas tautologique), ou `Insatisfiable`.

Résolution naïve (exploration exhaustive) pour le problème SAT

31. En utilisant un code assez similaire à celui de la fonction `affiche_table_verite`, écrire une fonction `valuations_sat : formule -> (int list) list` qui calcule toutes les valuations qui évaluent P à `true`. En gros, cette fonction considère toutes les 2^n valuations ϕ (les lignes de la table de vérité de P), et filtre celles qui évaluent P à `true`.
32. Définir une fonction récursive `deux_puissance : int -> int` qui calcule 2^n. Note : on peut aussi utiliser `1 lsl n` qui calcule pareil.
33. En déduire une fonction `calcule_nature : formule -> nature_formule` qui calcule le nombre de valuations ϕ satisfaisant P et en déduit sa nature.

Conséquence logique et équivalence logique

On rappelle que $P \models Q$, qui se lit "Q est conséquence logique de P", signifie que toute valuation ϕ qui satisfait P satisfait aussi Q. Autrement dit, $P \to Q$ (l'implication logique) est une tautologie (ce que l'on peut noter $\models P \to Q$).

34. Écrire une fonction `est_consequence_logique : formule -> formule -> bool` qui calcule si $P \models Q$ ou non.
35. En déduire une fonction `sont_equivalents : formule -> formule -> bool` qui calcule si $P \equiv Q$ (on rappelle que cela signifie $P \models Q$ et $Q \models P$).

Autres connecteurs logiques : NAND, NOR, XOR. . .

36. Définir trois fonctions `nand : formule -> formule -> formule`, `nor : formule -> formule -> formule` et `xor : formule -> formule -> formule` qui calculent les arbres de syntaxe des formules P NAND Q, P NOR Q et P XOR Q.
37. Afficher les tables de vérité de x_1 NAND x_2, x_1 NOR x_2 et x_1 XOR x_2, que voici dans l'ordre (pour vérifier) :

```
# affiche_table_verite (nand x1 x2);;
| x1 | x2 | ¬(x1 ^ x2)
| false | false | true
| false | true | true
| true | false | true
| true | true | false
- : unit = ()

# affiche_table_verite (nor x1 x2);;
| x1 | x2 | ¬(x1 v x2)
| false | false | true
| false | true | false
| true | false | false
| true | true | false
- : unit = ()

# affiche_table_verite (xor x1 x2);;
| x1 | x2 | ((x1 ^ ¬x2) v (¬x1 ^ x2))
| false | false | false
| false | true | true
| true | false | true
| true | true | false
- : unit = ()
```