On se donne un ensemble de variables booléennes $\mathtt{V} = \{\mathtt{v_1}, \ldots, \mathtt{v_n}\}$. On définit par induction la notion de formule logique :

- Si $\mathtt{v} \in \mathtt{V}$ alors `v` est une formule logique.
- Conjonction : si `f` et $\mathtt{f'}$ sont des formules logiques alors $\mathtt{f} \wedge \mathtt{f'}$ est une formule logique.
- Disjonction : si `f` et $\mathtt{f'}$ sont des formules logiques alors $\mathtt{f} \vee \mathtt{f'}$ est une formule logique.
- Négation : si `f` est une formule logique alors $\neg\mathtt{f}$ est une formule logique.

Codage arborescent

Les formules seront codées sous la forme d'un arbre grâce au type :

```
type formule = | Var of int
               | Et of formule * formule
               | Ou of formule * formule
               | Non of formule;;
```

1) Définir la variable `exemple_formule` de type `formule` qui permet de coder la formule logique :

$$\neg(\mathtt{e_3} \wedge (\mathtt{e_1} \vee (\neg\mathtt{e_2})))$$

2) On se donne un tableau de booléens `contexte : bool array` qui permet de stocker les valeurs booléennes des variables $\mathtt{v_1}, \ldots, \mathtt{v_n}$: la valeur de la variable $\mathtt{v_i}$ est stockée dans `contexte.(i)`. On remarquera que la case d'indice 0 ne sert à rien et que le tableau `contexte` doit être de taille $n+1$.
Écrire une fonction `evalue` telle que `evalue f contexte` permet d'évaluer la formule `f` lorsque les valeurs des variables $\mathtt{v_1}, \ldots, \mathtt{v_n}$ sont stockées dans `contexte`.

Satisfaisabilité par énumération des cas

Nous allons désormais chercher à savoir si une formule est satisfaisable, c'est–à–dire s'il existe des valeurs que l'on peut donner aux variables $\mathtt{v_1} \ldots \mathtt{v_n}$ de manière à ce qu'une formule s'évalue en `true`.

3) Écrire une fonction `max_indice_var` qui permet de calculer le maximum des indices des variables apparaissant dans une formule.

4) Écrire une fonction récursive :

```
teste_depuis_indice : int -> int -> bool array -> formule -> bool
```

dont les arguments sont l'entier `n` qui borne les indices des variables apparaissant dans la formule, un entier `k`, un contexte $[|\mathtt{v_0}; \mathtt{v_1}; \ldots; \mathtt{v_k}; \mathtt{v_{k+1}}; \ldots; \mathtt{v_n}|]$ et une formule, et qui teste tous les contextes possibles en modifiant les valeurs de $\mathtt{v_{k+1}}, \ldots, \mathtt{v_n}$ mais en gardant $\mathtt{v_0}, \mathtt{v_1}, \ldots, \mathtt{v_k}$ inchangés. S'il existe un tel contexte dans lequel la formule s'évalue en `true` alors la valeur de retour de la fonction doit être `true` et le tableau `contexte` doit contenir en fin d'exécution des valeurs pour lesquelles la formule s'évalue en `true`. Sinon la fonction doit renvoyer `false` et les valeurs stockées dans `contexte` n'ont pas d'importance. On remarquera qu'il y a au plus $2^{\mathtt{n}-\mathtt{k}}$ contextes à tester.

5) Écrire une fonction `satisfaisable` qui prend en entrée une formule et qui renvoie un couple `(res, contexte)` défini ainsi : `res` est un booléen qui permet de savoir si la fonction `f` est satisfaisable ou non, et, dans le cas où `f` est satisfaisable, `contexte` est un tableau de booléens qui permet de satisfaire la formule. Si `f` n'est pas satisfaisable, les valeurs stockées dans `contexte` n'ont pas d'importance.
On rappelle de la fonction `Array.make : int -> 'a -> 'a array` permet de créer un nouveau tableau. Son premier paramètre est la taille du tableau, et son second paramètre est la valeur à laquelle est égale chaque case du tableau nouvellement créé.

6) Le club écossais.
Il existe en écosse un club privé qui obéit aux règles suivantes :

A) Tout membre non écossais joue au golf
B) Tout membre porte un kilt ou ne joue pas au golf
C) Les membres mariés ne sortent pas le dimanche
D) Un membre sort le dimanche si et seulement s'il est écossais
E) Tout membre qui porte un kilt est écossais et marié
F) Tout membre écossais porte un kilt

a) Montrer que le club ne comporte aucun membre.
Pour simplifier la saisie, on pourra écrire une fonction
`implique : formule -> formule -> formule`
telle que `implique p q` renvoie une formule équivalente à $p \Rightarrow q$ ainsi qu'une fonction
`etliste : formule list -> formule`
telle que si `p1`, ..., `pn` sont des formules, la fonction `eliste [p1,...,pn]` renvoie une formule équivalente à $p1 \wedge \cdots \wedge pn$.

b) On considère la nouvelle règle :

Ebis) Tout membre qui porte un kilt est écossais ou marié

Que se passe-t-il si on remplace E) par Ebis) ?

Forme normale conjonctive

Un *littéral* est une variable ou la négation d'une variable, une *clause* est une disjonction de littéraux et une formule est *sous forme normale conjonctive* si c'est une conjonction de clauses. Récapitulons :

- Littéral : $\mathtt{v}$ ou $\neg\mathtt{v}$ avec $\mathtt{v} \in \mathtt{V}$
- Clause : $\mathtt{l_1} \vee \cdots \vee \mathtt{l_k}$, les $\mathtt{l_i}$ étant des littéraux.
- Forme normale conjonctive : $\mathtt{c_1} \wedge \cdots \wedge \mathtt{c_p}$, les $\mathtt{c_j}$ étant des clauses.

On remarquera qu'une clause $\mathtt{c_j}$ est une tautologie si et seulement si il existe une variable $\mathtt{v_i} \in \mathtt{V}$ telle que les littéraux $\mathtt{v_i}$ et $\neg\mathtt{v_i}$ apparaissent simultanément dans $\mathtt{c_j}$. On remarquera aussi qu'une forme normale conjonctive est une tautologie si et seulement si chacune de ses clauses est une tautologie.

7) Dans une première étape nous allons chercher à faire "descendre" les négations à l'intérieur de l'arbre. En utilisant les règles de De Morgan, écrire une fonction `descend_negations` qui prend en argument une formule logique et qui renvoie une formule logique équivalente, mais dans laquelle tout nœud correspondant à une négation ne peut avoir comme fils qu'une feuille correspondant à une variable. Vérifier que sur l'exemple de la question 1 on obtient $(\neg\mathtt{e_3}) \vee ((\neg\mathtt{e1}) \wedge \mathtt{e_2})$.

8) Écrire une fonction `forme_normale_conjonctive` qui prend en argument une formule logique et qui renvoie une formule équivalente sous forme normale conjonctive. On pourra utiliser la distributivité du "ou" sur le "et".
Attention : c'est un peu plus difficile qu'il n'y paraît. Tester avec la formule $(\mathtt{e_1} \wedge \mathtt{e_2} \wedge \mathtt{e_3}) \vee (\mathtt{e_4} \vee \mathtt{e_5})$.

9) Écrire une fonction `tautologie_clause` dont le paramètre est une clause et qui renvoie un booléen indiquant si la clause est une tautologie ou non. Dans le cas où on testerait cette fonction avec une formule qui ne serait pas une clause, le résultat n'a pas d'importance.

10) Écrire une fonction `liste_clauses : formule -> formule list` qui prend en argument une formule logique sous forme normale conjonctive, et qui retourne la liste des clauses dont elle est la conjonction. Dans le cas où on testerait cette fonction avec une formule qui ne serait pas sous forme normale conjonctive, le résultat n'a pas d'importance.

11) En déduire une fonction `tautologie` dont le paramètre est une formule (quelconque) et qui renvoie un booléen indiquant si cette formule est une tautologie ou non.

12) En déduire une autre programmation de la fonction `satisfaisable` qui permet de savoir si une formule est satisfaisable ou non.