Dans tout le TP, les ensembles d'éléments totalement ordonnés seront représentés comme des listes *triées sans doublon*. Sur cahier de prépa vous trouverez un fichier contenant un module `Set` permettant la manipulation de tels ensembles (ce module suppose que la relation d'ordre totale sur les éléments coïncide avec celle définie par `<` en OCaml).

Dans toute la suite on suppose défini le type :

```
type 'a set = 'a Set.t
```

Exercice 1 : Syntaxe et sémantique

1. Syntaxe

On rappelle que la logique propositionnelle est le plus petit ensemble obtenu par induction à partir des symboles suivants :

- $\bot|^0$
- $\top|^0$
- $\wedge|^2$
- $\vee|^2$
- $\rightarrow|^2$
- $\leftrightarrow|^2$
- $\neg|^1$
- $\mathcal{V}|^0_{\mathcal{P}}$

On vous fournit sur cahier de prépa un parseur de chaînes de caractères en formules. Par exemple :
`(parse "vv | q & r > ~(c & (t > b) | (x = (y)))")` produit la valeur OCaml :

```
Imply (
  Or (Var "vv", And (Var "q", Var "r")),
  Not (Or (
    And (Var "c", Imply (Top, Bot)),
    Equiv (Var "x", Var "y"))))
```

Ce parseur utilise les symboles suivants : `~` pour "non", `&` pour "et", `|` pour "ou", `>` pour "implique", `=` pour "équivaut", pour cet ordre de précédence (comprendre : un `"~p & q"` est lu `And(Not("p"), "q")`). Les lettres `t` et `b` sont réservées à `Top` et `Bot`, les autres lettres (ou séquences de lettres) sont interprétées comme des variables propositionnelles. Les espaces n'ont aucune importance.

Q. 1 Définir un type OCaml `type formule` permettant la représentation des formules de la logique propositionnelle. On pourra prendre $\mathcal{P} =$ `string` (l'ensemble des variables propositionnelles).

Q. 2 Définir une fonction `val print_formule : formule -> unit` permettant l'affichage d'une formule de la logique propositionnelle.

Q. 3 Définir une fonction `val vars : formule -> string set` donnant l'ensemble des variables d'une formule.

2. Sémantique

On représente un environnement propositionnel au moyen d'une liste d'associations des variables vers les booléens. On rappelle qu'une liste d'association d'un type A vers un type B est une liste de type : `(`A `*` B`) list`. La présence d'une paire (a, b) dans une telle liste indique l'association de la valeur b à la clé a. Aussi la liste d'association `[(`p`, true); (`q`, false)]` représente l'environnement propositionnel : $(p \mapsto \mathsf{V}, q \mapsto \mathsf{F})$.

Q. 4 Définir le type `type env_prop` permettant la représentation d'un environnement propositionnel..

Q. 5 Définir une fonction d'affichage `val print_env_prop : env_prop -> unit` permettant l'affichage d'un environnement propositionnel. On pourra par exemple avoir comme affichage :

```
{p ~> V, q ~> F}
```

Q. 6 Définir une fonction `val interprete : formule -> env_prop -> bool` permettant l'interprétation d'une formule dans un environnement propositionnel. On lèvera l'exception `Missing_Env` si l'environnement propositionnel ne contient pas toutes les variables utilisées dans la formule.

Q. 7 Définir une fonction `val all_envs : string list -> env_prop list` générant tous les environnements sur les variables propositionnelles passées en arguments.

Q. 8 Définir une fonction `val sat : formule -> env_prop` permettant de résoudre le problème SAT. On lèvera l'exception `Unsat` si la formule est insatisfiable.

Q. 9 Définir une fonction `val est_valide : formule -> true` permettant de tester si une formule est valide.

Q. 10 Définir une fonction `val est_cons_semantique : formule -> formule -> bool` permettant de tester si une formule est conséquence sémantique d'une autre.

Q. 11 Définir une fonction `val equiv : formule -> formule -> bool` permettant de tester si deux formules sont équivalentes.

Q. 12 Définir une fonction `val models : formule -> env_prop set` donnant l'ensemble des modèles d'une formule.

Exercice 2 : Construction d'une formule à partir d'une fonction

On définit le type `type fct_bool = env_prop -> bool`.

Q. 1 Donner une fonction `val formule_of_fct_bool : string list -> fct_bool -> formule` calculant une formule dont la sémantique est la fonction booléenne passée en argument. On utilisera dans cette question l'algorithme vu en classe et produisant une disjonction de conjonction de littéraux. On fournit en argument l'ensemble des variables propositionnelles sur lequel est définie la fonction booléenne.

Q. 2 Donner une fonction `val formule_of_fct_bool2 : string list -> fct_bool -> formule` calculant une formule dont la sémantique est la fonction booléenne passée en argument. On utilisera dans cette question l'algorithme vu en classe et produisant une conjonction de disjonction de littéraux. On fournit en argument l'ensemble des variables propositionnelles sur lequel est définie la fonction booléenne.

Exercice 3 : Application de règles de réécriture

On s'intéresse à l'utilisation de *règles de réécritures* pour la transformation automatique de formules.

Règles de réécriture. On appelle règle de réécriture (c'est une définition par l'exemple) un énoncé de la forme $H_1 \wedge H_2 \rightsquigarrow \neg((\neg H_1) \vee (\neg H_2))$. Une telle règle décrit comment des sous-formules d'une formule doivent être successivement transformées. Un ensemble de règles de réécriture $(g_1 \rightsquigarrow d_1, \ldots, g_n \rightsquigarrow d_n)$ décrit donc un algorithme :

Algorithme 1 : Application d'un système de réécriture $(g_1 \rightsquigarrow d_1, \ldots, g_n \rightsquigarrow d_n)$ à une formule H

Entrée : Une formule H

tant que *il existe une sous-formule* G *de* H *de la forme* g_i **faire**
  remplacer G dans H par d_i ;

Par exemple la règle de réécriture $H_1 \wedge H_2 \rightsquigarrow \neg((\neg H_1) \vee (\neg H_2))$ permet la transformation d'une formule en une formule équivalente ne contenant pas de connecteur $\wedge$.

On représente un ensemble de règles de réécriture en OCaml par une fonction de type `formule -> formule option`. On définit donc le type : `type rewrite = formule -> formule option`. Par exemple la règle de réécriture $H_1 \wedge H_2 \rightsquigarrow \neg((\neg H_1) \vee (\neg H_2))$ peut être encodé au moyen de la fonction :

```
let ex =
  fun (f: formule) -> match f with
  | And(h1, h2) -> Some(Not(Or(Not(h1), Not(h2))))
  | _ -> None
```

Ainsi lorsque la fonction retourne `None`, c'est qu'aucune règle de réécriture n'a pu être appliquée sur la “tête” de la formule, sinon c'est qu'une réécriture peut avoir lieu, auquel cas la fonction nous fournit la formule transformée.

L'ensemble de règles de réécriture $(H_1 \wedge H_2 \rightsquigarrow \neg((\neg H_1) \vee (\neg H_2)), H_1 \vee H_2 \rightsquigarrow \neg((\neg H_1) \wedge (\neg H_2)))$ peut être encodé au moyen de la fonction :

```
let ex =
  fun (f: formule) -> match f with
  | And(h1, h2) -> Some(Not(Or(Not(h1), Not(h2))))
  | Or(h1, h2) -> Some(Not(And(Not(h1), Not(h2))))
  | _ -> None
```

On remarque au passage que cet ensemble de règles de réécriture a le mauvais goût de fournir un algorithme ne terminant pas.

Q. 1 Donner une fonction `val rewrite_one : rewrite -> formule -> formule option`, permettant l'application d'*une* règle de réécriture sur la formule H passée en argument. S'il est possible d'appliquer des réécritures à plusieurs sous-formules de H on choisira d'appliquer la réécriture sur la première sous-formule de H rencontrée dans l'ordre de parcours préfixe des sous-formules. Si aucune sous-formule de H ne conduit à une réécriture, votre fonction devra alors s'évaluer à `None`.

Exemples :

> Avec le système de réécriture exemple ci-dessus, et la formule $H = (p \wedge q) \rightarrow (q \wedge r)$, la fonction doit calculer la formule $\neg(\neg p \vee \neg q) \rightarrow (q \wedge r)$

Q. 2 En déduire une fonction `val rewrite : rewrite -> formule -> formule` calculant le résultat de l'algorithme d'application des règles de réécriture décrit ci-avant. Votre fonction pourra ne pas terminer, en fonction du système de réécriture.

Application. On cherche à construire une formule équivalente qui ne contient que des variables propositionnelles et les connecteurs $\neg$ et $\wedge$.

Q. 3 Proposer un système de réécriture permettant cette transformation.

Q. 4 Tester votre système de réécriture avec votre fonction `rewrite` ci-avant définie.

Q. 5 Prouver la terminaison et la correction de votre système de réécriture.

Exercice 4 : Dessin de tables de vérités

Cette question est plus ouverte que les précédentes, elle pourra être laissée de côté en première lecture du TP.

Q. 1 Définir une fonction `val truth_table : formule -> unit` permettant l'affichage de la table de vérité d'une formule, par exemple pour la formule $(p \wedge q) \rightarrow (q \vee r)$:

```
+---+---+--------+---+--------+----------------------+
| p | q | p /\ q | r | q \/ r | (p /\ q) -> (q \/ r) |
+---+---+--------+---+--------+----------------------+
| F | F |   F    | F |   F    |          V           |
| F | F |   F    | V |   V    |          V           |
| F | V |   F    | F |   V    |          V           |
| F | V |   F    | V |   V    |          V           |
| V | F |   F    | F |   F    |          V           |
| V | F |   F    | V |   V    |          V           |
| V | V |   V    | F |   V    |          V           |
| V | V |   V    | V |   V    |          V           |
+---+---+--------+---+--------+----------------------+
```

Exercice 5 : Représentation de FNC et FND au moyen d'ensembles

Du fait de l'associativé et de la commutativité des opérateurs $\cdot$ et $+$ de l'algèbre de Boole, une formule normale conjonctive (resp. disjonctive) peut-être représentée comme un ensemble d'ensemble de littéraux. En effet l'ensemble $\{\{p, \neg q\}, \{\neg p, \neg q, \neg r\}\}$ représente la FNC : $(p \vee \neg q) \wedge (\neg p \vee \neg q \vee \neg r)$. Dans le cadre d'une FND il représente la formule : $(p \wedge \neg q) \vee (\neg p \wedge \neg q \wedge \neg r)$. Il n'est en effet pas nécessaire d'avoir des doublons de littéraux dans les clauses (disjonctives ou conjonctives) puisqu'il est possible de montrer facilement qu'une FNC (resp. une FND) est équivalente à une FNC (resp. une FND) dans laquelle toute clause ne fait apparaître chaque variable au plus qu'une fois par clause (soit dans un littéral p ou $\neg p$).

On remarque que l'ensemble $\{\emptyset\}$ représente la FNC $\perp$ (resp. la FND $\top$), et que $\emptyset$ représente la FNC $\top$ (resp. la FND $\perp$).

Vous trouverez sur moodle un fichier définissant la notion de littéral :

```
type litteral = {
  var : string;
  pn  : bool     (* true pour un littéral positif *)
}
```

Ainsi `{var : "p"; pn : true}` représente le littéral p et `{var : "q"; pn : false}` représente le littéral $\neg q$.

Et la notion de FNC, FND.

```
type fnc = litteral set set
type fnd = litteral set set
```

On assurera dans toute la suite que les clauses (disjonctives ou conjonctives) ne contiennent qu'une unique occurrence de chaque variable. Aussi il n'est pas possible d'avoir une clause contenant à la fois les littéraux p et $\neg p$.

Q. 1 Définir les valeurs `fnc_true: fnc` (resp. `fnc_false: fnc`, `fnd_true: fnd`, `fnd_false: fnd`) donnant une FNC (resp. FNC, FND, FND) équivalente à $\top$ (resp $\bot$, $\top$, $\bot$).

Q. 2 Définir les fonctions `fnc_mk_lit (l: litteral): fnc` (resp. `fnd_mk_lit (l: litteral): fnd`) permettant le calcul d'une FNC (resp. d'une FND) du littéral passé en argument.

Q. 3 Définir les fonctions `fnc_mk_and (fc1: fnc) (fc2: fnc): fnc` (resp. `fnd_mk_or (fd1: fnd) (fd2: fnd): fnd`) permettant le calcul d'une FNC de la formule $H_1 \wedge H_2$ où H_1 et H_2 sont des formules de FNC `fc1` et `fc2` (resp. d'une FND de la formule $H_1 \vee H_2$ où H_1 et H_2 sont des formules de FND `fd1` et `fd2`).

Q. 4 Définir les fonctions `fnc_mk_or (fc1: fnc) (fc2: fnc): fnc` (resp. `fnd_mk_and (fd1: fnd) (fd2: fnd): fnd`) permettant le calcul d'une FNC de la formule $H_1 \vee H_2$ où H_1 et H_2 sont des formules de FNC `fc1` et `fc2` (resp. d'une FND de la formule $H_1 \wedge H_2$ où H_1 et H_2 sont des formules de FND `fd1` et `fd2`).

Q. 5 Définir les fonctions `fnc_of_fnd (fd: fnd): fnc` (resp. `fnd_of_fnc (fc: fnc): fnd`) permettant la traduction d'une FND en une FNC équivalente (resp. d'une FNC en une FND équivalente).

Q. 6 Définir les fonctions `fnd_mk_not (f: fnd): fnd` (resp. `fnc_mk_not (fc: fnc): fnc`) permettant le calcul d'une FNC de la formule $\neg H$ où H est une formule de FNC `fc1` (resp. d'une FND de la formule $\neg H$ où H est une formule de FND `fd1`).

Q. 7 Définir une fonction `fnc_of_formule (f: formule): fnc` permettant la traduction, par induction, d'une formule en FNC.

Q. 8 Définir une fonction `sat : fnd -> env_prop` permettant de résoudre le problème SAT. On utilisera l'algorithme proposé en classe permettant la résolution de SAT dans le cas d'une FND. On lèvera l'exception `Unsat` si la formule est insatisfiable.

Exercice 6 : Algorithme de Quine

Cet exercice nécessite d'avoir traité l'exercice 5. En effet les types utilisés ici sont ceux de cet exercice.

Q. 1 Définir une fonction `assume: (l: litteral) (f: fnc): fnc`, prenant en arguments un littéral et une forme normale conjonctive et calculant une forme normale conjonctive équivalente à `f` $\wedge$ `l`. On appliquera un traitement clause par clause plutôt que de se reposer sur les résultats de l'exercice précédent.

Q. 2 Définir la fonction `is_true (f: fnc): bool` donnant vrai si et seulement si `f` est la FNC $\emptyset$. Définir la fonction `is_false (f: fnc): bool` donnant vrai si et seulement si `f` contient la clause $\emptyset$.

Q. 3 Définir une fonction `vars: fnc -> string set` calculant l'ensemble des variables d'une FNC.

Q. 4 Définir une fonction `find_trivial_clause (f: fnc): litteral option` retournant, s'il existe, un littéral l de `f` tel que $\{l\}$ est une des clauses de `f`. On renverra `None` si un tel littéral n'existe pas.

Q. 5 Définir une fonction `quine_sat : fnc -> env` permettant de résoudre le problème SAT sur une FNC, par algorithme de Quine. On lèvera l'exception `Unsat` si la formule est insatisfiable.

Q. 6 Définir une fonction `quine_valide : fnc -> bool` permettant de résoudre le problème VALIDE sur une FNC, par algorithme de Quine.