TP D'OPTION INFORMATIQUE 4
Logique propositionnelle

1 Syntaxe et sémantique

On représente les formules du calcul propositionnel (sans les constantes $\top$ et $\bot$) par le type Caml suivant :

```
type 'a formule =
      V of 'a
    | Neg of 'a formule
    | Et of 'a formule * 'a formule
    | Ou of 'a formule * 'a formule
    | Imp of 'a formule * 'a formule
    | Eq of 'a formule * 'a formule;;
```

Le type `'a` correspond au type des noms des variables apparaissant dans les formules. On pourra par exemple utiliser le type `char` à cet effet.

1. Représenter la formule $A \Rightarrow ((A \Rightarrow B) \Rightarrow B)$ en Caml.
2. Écrire une fonction `interpretation : 'a formule -> ('a -> bool) -> bool` prenant en argument une formule et une valuation (ie une fonction qui à chaque variable associe un booléen) et qui renvoie la valeur de vérité de cette formule sous cette valuation.

2 Mise en forme normale conjonctive

On rappelle qu'une forme normale conjonctive (FNC) est une conjonction de clause, une clause étant une disjonction de littéraux et un littéral étant une variable ou la négation d'une variable.

1. Donner une FNC équivalente à `f = (Eq(V 'A' , V 'B'))`.
2. Écrire une fonction `fnc1 : 'a formule -> 'a formule` prenant en argument une formule et renvoyant une formule équivalente sans constructeur `Imp` ou `Eq`.
3. Écrire une fonction `fnc2 : 'a formule -> 'a formule` prenant en argument une formule obtenue par `fnc1` et renvoyant une formule équivalente sans négation sauf devant une variable.
4. On donne la définition récursive de la troisième étape de la construction de la fnc :
   - $f_3(\phi) = \phi$ si ϕ est un littéral ;
   - $f_3(\phi \vee (\psi \wedge \theta)) = f_3(\phi \vee \psi) \wedge f_3(\phi \vee \theta)$;
   - $f_3((\psi \wedge \theta) \vee \phi) = f_3(\phi \vee \psi) \wedge f_3(\phi \vee \theta)$;
   - $f_3(\phi \wedge \psi) = f_3(\phi) \wedge f_3(\psi)$;
   - $f_3(\phi \vee \psi) = \phi \vee \psi$ si $f_3(\phi) = \phi$ et $f_3(\psi) = \psi$;
   - $f_3(\phi \vee \psi) = f_3(f_3(\phi) \vee f_3(\psi))$ sinon.

   Implémenter cette étape dans une fonction `fnc3`.
5. En déduire une fonction `fnc : 'a formule -> 'a formule` prenant en argument une formule et renvoyant sa forme normale conjonctive.
6. On définit la forme formatée d'une clause comme la liste de ses littéraux, et la forme formatée d'une FNC comme la liste de ses clauses.
   (a) Donner la forme formatée de la FNC de la formule `f` précédente.
   (b) Écrire une fonction `formater_fnc : 'a formule -> 'a formule list list` prenant en argument une FNC et renvoyant sa forme formatée.

3 Satisfiabilité

L'objet de cette partie est de tester la satisfiabilité d'une formule, en testant l'ensemble des valuations possibles. Pour cela, il faut commencer par disposer de l'ensemble des variables présentes dans la formule. Nous choisirons de représenter de tels ensembles de variables par des listes triées sans doublon.

1. Écrire une fonction `insertion : 'a list -> 'a -> 'a list` prenant en argument une liste triée sans doublons l et un élément x, et renvoyant la liste triée sans doublon obtenue à partir de l en insérant x (si x est déjà présent dans l, l est donc renvoyée inchangée). Justifier que cette fonction est de complexité linéaire.
2. En déduire une fonction `liste_variables : 'a formule -> 'a list` prenant en argument une formule et renvoyant la liste triée sans doublon de ses variables. On pourra utiliser une référence de liste.
3. Écrire une fonction `sat : 'a formule -> bool * int array` prenant en argument une formule et renvoyant un booléen indiquant si la formule est satisfiable. Pour cela on pourra :
   - obtenir la liste des variables de la formule, et en faire un tableau avec la fonction `Array.of_list` ;
   - utiliser un second tableau `tv` de même longueur, représentant une valuation en contenant 1 dans la case d'indice i si la variable d'indice i est affectée à vrai, et 0 sinon ;
   - considérer chaque valuation possible en énumérant toutes les valeurs possibles de `tv`, sur le modèle d'une succession d'incrémentations en base 2 ;
   - pour chaque valeur possible de `tv`, on peut alors en déduire la valuation correspondante et tester si elle satisfait la formule.

   En plus d'un booléen, la fonction renverra le tableau `tv` qui encode une valuation satisfaisant la formule, lorsque celle-ci existe.
4. Déduire de la fonction précédente une fonction `est_valide : 'a formule -> bool * int array` prenant en argument une formule et testant la validité. Sur le même modèle que précédemment, on renverra également un tableau encodant une valuation ne satisfaisant pas la formule, si celle-ci n'est pas valide.
5. On peut également tester la validité d'une formule en passant par la FNC.
   (a) justifier qu'une FNC est valide si et seulement si chacune de ses clauses est valide.
   (b) Justifier qu'une clause est valide si et seulement si elle contient un littéral et son opposé.
   (c) En déduire une fonction `fnc_valide : 'a formule list list -> bool` prenant en argument une FNC formatée et testant sa validité.
   (d) En déduire une fonction `sat2 : 'a formule -> bool` testant la satisfiabilité d'une formule.
   (e) La complexité de `sat2` dans le pire cas est-elle meilleure que celle de `sat` ?

4 Application au sudoku

On s'intéresse dans cette partie à la possibilité de résoudre un sudoku en le traduisant en un problème de satisfiabilité de formule.

1. Rappeler les variables et les formules considérées.
2. Estimer en ordre de grandeur le nombre de valuations à considérer pour un sudoku de taille 9×9, et perdre tout espoir.
3. Écrire une fonction `solution_sudoku` prenant en argument une matrice représentant un sudoku de taille 3×3 (les cases vides représentées par des 0), et renvoyant un booléen indiquant si le sudoku a une solution, et modifiant la matrice avec une telle solution le cas échéant. Afin de gagner en efficacité, on utilisera une variante de la fonction `sat` qui ne teste que les valuations vérifiant l'existence et l'unicité de la valeur de chaque case (ces contraintes n'auront donc pas besoin d'apparaître dans la formule).