TP n°28 - Logique propositionnelle

Toutes les fonctions de ce TP (5 exercices) seront codées dans un unique fichier `NOM_logic.ml`.
On ne cherchera pas à rendre les fonctions récursives récursives terminales. On pourra essayer éventuellement à la maison mais ce n'est pas le but ici.
Votre fichier de code devra être déposé sur Moodle pour mercredi prochain.

Exercice 1 (Syntaxe des formules logiques).

1. Proposez un type récursif OCaml nommé `flog` permettant de représenter une formule de la logique propositionnelle. On ne s'occupera pas du connecteur logique $\leftrightarrow$. **On suppose que les variables propositionnelles sont étiquetées par des entiers strictement positifs.**
2. Construire les formules logiques suivantes, qui nous serviront pour tester nos fonctions :

$$\varphi_1 = (p_1 \vee p_2)\,, \quad \varphi_2 = ((p1 \vee p_2) \wedge p_3)\,, \quad \varphi_3 = ((p_1 \vee (p_2 \to (\neg p_3))) \wedge p_4)$$

3. Écrire une fonction `flog_to_string` qui affiche une formule logique à l'écran sous sa forme infixe stricte. Comme il n'est pas aisé d'afficher des caractères comme $\wedge$ ou $\vee$, on remplacera ces symboles par les suivants :

Variable propositionnelle p_2 par exemple	remplacé par	`p2`
$\bot$	remplacé par	`F`
$\top$	remplacé par	`T`
$\wedge$	remplacé par	`et`
$\vee$	remplacé par	`ou`
$\neg$	remplacé par	`non`
$\to$	remplacé par	`->`

   Par exemple, pour la formule φ_3, la fonction doit afficher :

```
((p1 ou (p2 -> (non p3))) et p4)
```

4. Écrire une fonction `flog_maximal_variable_index` qui renvoie l'indice maximal parmi les indices des variables propositionnelles présents dans une formule logique. Tester votre fonction sur les formules précédentes et sur la formule :

$$\varphi_4 = ((p_1 \vee (p_2 \to p_3)) \wedge (p_4 \to (p_2 \vee p_7)))$$

Exercice 2 (Évaluation des formules logiques).
Une valuation sera représentée par un tableau de booléens.

1. Écrire une fonction `flog_eval` qui évalue une formule logique pour une valuation donnée en entrée.
2. Tester votre fonction sur les formules φ_1, φ_2 et φ_3 et sur différentes valuations.
3. Écrire une fonction `flog_create_valuations: int -> bool array array` qui renvoie toutes les valuations possibles lorsque la formule comporte n variables propositionnelles différentes. On pourra utiliser la fonction `make_matrix` du module `Array`. *Indication : une valuation sur n variables propositionnelles peut être vue comme l'écriture binaire d'un entier sur n bits...*
4. Écrire une fonction `flog_table: flog -> bool array array * bool array` qui renvoie le tableau des valuations et le tableau associé des $[\![\varphi]\!]_v$
5. (Non prioritaire, peut être fait à la maison.) Écrire une fonction `flog_print_table: flog -> unit` qui utilise la fonction précédente et affiche proprement à l'écran la table de vérité d'une formule logique. Voici un exemple d'affichage agréable pour un utilisateur, pour la formule φ_3 :

```
Table de vérité de ((p1 ou (p2 -> (non p3))) et p4)
 p1 | p2 | p3 | p4 |
 F  | F  | F  | F  |-----> F
 V  | F  | F  | F  |-----> F
 F  | V  | F  | F  |-----> F
 V  | V  | F  | F  |-----> F
 F  | F  | V  | F  |-----> F
 V  | F  | V  | F  |-----> F
 F  | V  | V  | F  |-----> F
 V  | V  | V  | F  |-----> F
 F  | F  | F  | V  |-----> V
 V  | F  | F  | V  |-----> V
 F  | V  | F  | V  |-----> V
 V  | V  | F  | V  |-----> V
 F  | F  | V  | V  |-----> V
 V  | F  | V  | V  |-----> V
 F  | V  | V  | V  |-----> F
 V  | V  | V  | V  |-----> V
- : unit = ()
```

Exercice 3 (Substitution et simplifications).

1. Écrire une fonction `flog_substitute: flog -> int -> flog -> flog` qui prend en entrée une formule logique, un numéro i de variable propositionnelle et une formule de substitution et construit une nouvelle formule où la variable propositionnelle p_i a été remplacée par cette formule de substitution.
2. Nous allons essentiellement utiliser la fonction précédente en substituant une variable propositionnelle par la formule $\top$ et ou par la formule $\bot$. On pourra alors, dans certains cas, simplifier la formule, en utilisant des équivalences sémantiques très connues comme, par exemple celles-ci, pour le connecteur $\wedge$:
$$\bot \wedge p \equiv \bot, \quad \top \wedge \top \equiv \top$$
Pour faire cela, nous allons d'abord créer des constructeurs intelligents `smart_and`, `smart_or`, `smart_not`, `smart_imply`... qui ne construirons un vrai nœud `and`, `or`, `imply` ou `not` quand cela est vraiment nécessaire. Par exemple, si j'appelle ce constructeur intelligent pour construire une nœud `and(`f_1`, ⊤)`, ce constructeur intelligent repérera qu'il est inutile d'empiler un constructeur de plus car $f_1 \wedge \top \equiv f_1$, et il retournera donc simplement f_1. Cela permettra de réduire petit à petit la taille des formules lorsque nous ferons des substitutions dans l'algorithme de Quine, comme lorsque nous l'avons fait à la main.
Écrire ces 4 petits constructeurs intelligents :
   - `smart_and`
   - `smart_or`
   - `smart_not`
   - `smart_imply`

   qui créent un nœud logique avec le constructeur classique du début uniquement lorsque cela est vraiment nécessaire.
3. Écrire une fonction `flog_substitute_and_simplify` qui adapte la fonction `flog_substitute` pour qu'elle effectue également les simplifications évidentes lors de la substitution.

Exercice 4 (Résolution de $SAT(\varphi)$ par l'algorithme de Quine).

1. Implémenter l'algorithme de Quine sous la forme d'une fonction `flog_sat_quine: flog -> bool`. Cette première version renvoie simplement un booléen indiquant si la formule logique est satisfiable ou non.
2. Tester abondamment votre algorithme, en particulier sur les 3 formules indiquées en début d'énoncé.
3. Créer une deuxième version de cette fonction, nommée `flog_sat_quine_better: flog -> string list` qui renvoie la liste des valuations qui rendent la formule satisfiable. Une valuation sera donnée sous la forme d'une chaîne de caractères formée de `0` et de `1`. Par exemple, la chaîne `0110` correspond à une valuation sur $n = 4$ variables propositionnelles, où p_1 est fausse, p_2 et p_3 sont vraies et p_4 est fausse. **On ne s'attachera pas à optimiser cette fonction ni à la rendre récursive terminale.**
4. Tester abondamment.
5. Il y a de fortes chances pour que votre fonction affiche des chaînes de caractères incomplètes, c'est-à-dire ne faisant pas exactement n caractères, et que certaines valuations soient sous-entendues. Par exemple, pour la formule $p_1 \to p_2$, l'algorithme renvoie :

   ```
   string list = [''0''; ''11'']
   ```

   alors que l'on voudrait avoir :

   ```
   string list = [''01''; ''00''; ''11'']
   ```

   Comment améliorer votre code pour garantir un affichage propre avec des valuations représentées par des chaînes de n caractères exactement ? Vous pourrez créer une version modifiée appelée `flog_sat_quine_better_version2`. *Indication : on pourra utiliser le module* `Bytes` *vu en cours.*

Exercice 5 (Mise sous forme CNF ou DNF - Écriture au format DIMACS).

1. Proposez un autre type OCaml nommé `flog_norm`, permettant de représenter de manière minimaliste une formule logique sous une forme normalisée DNF ou CNF. Vous pourrez définir d'autres types intermédiaires qui seront utilisés par le type `flog_norm`. *Indication : une formule logique sous forme DNF ou CNF peut être vue comme une liste de clauses, chaque clause étant elle-même une liste de littéraux...*
2. Écrire une fonction `flog_to_dnf: flog -> flog_norm` qui fournit une représentation DNF de n'importe quelle formule logique donnée en entrée.
3. Écrire une fonction `flog_to_cnf: flog -> flog_norm` qui fournit une représentation CNF de n'importe quelle formule logique donnée en entrée.
4. Écrire une fonction `flog_write_dimacs: flog_norm -> string -> unit` qui prend en entrée une formule logique sous forme CNF ou DNF, un nom de base `basename` et écrit la formule logique dans un fichier `./basename.dimacs` au format DIMACS. Voici, en guise de rappel, un exemple de fichier au format DIMACS, ici pour une CNF à 4 variables et 9 clauses.

```
 p cnf 4 9
1 -2 -3 -4 0
-1 -2 -3 4 0
1 -2 -3 4 0
-1 2 -3 4 0
1 2 -3 4 0
-1 -2 3 4 0
1 -2 3 4 0
-1 2 3 4 0
1 2 3 4 0
```

   Les nombres négatifs correspondent à des variables propositionnelles auxquels un opérateur `not` est appliqué. Par exemple, ici, la premier clause est $(p_1 \vee (\neg p_2) \vee (\not{p}_3) \vee (\not{p}_4))$
5. Pour valider votre code :
   - choisissez une formule logique (pas trop compliquée non plus),
   - déclarez et construisez-la dans votre code
   - affichez sa table de vérité pour voir si elle est satisfiable et si oui, pour quelle(s) valuation(s).
   - utilisez ensuite votre code pour obtenir une forme CNF de cette formule
   - écrivez le fichier DIMACS correspondant à cette forme CNF
   - allez sur le site suivant, qui propose un solver SAT qui prend en entrée des formules logiques au format DIMACS :

     http://logicrunch.it.uu.se:4096/~wv/minisat/

   - copiez-coller le contenu de votre fichier DIMACS dans l'interface Web...et vérifiez que la réponse de ce solver SAT correspond bien à la table de vérité calculée. Ce solver SAT ne donne qu'une seule valuation modèle.
   - vérifiez enfin que votre algorithme de Quine amélioré fonctionne, et que la valuation modèle trouvée par le solver MiniSAT fait bien partie des valuations modèle trouvées par votre algorithme de Quine.
6. Écrire une fonction `flog_read_dimacs: string -> flog_norm` qui lit un fichier au format DIMACS et renvoie une formule de type `flog_norm`