TP 16 : Logique propositionnelle, algorithme de Quine

Exercice 1 – Logique propositionnelle

▶ **Question 1** Proposez un type `formule` des formules propositionnelles, comprenant la possibilité d'écrire les formules $\top$, $\bot$, et indexant les variables propositionnelles par des entiers naturels.

On utilisera le type `type valuation = bool array` pour représenter une valuation sur les variables propositionnelles $p_0, \dots, p_{n-1}$, avec n la taille du tableau. Pour ν une valuation, on la représente par le tableau où $\nu(p_k)$ est inscrit dans la case d'indice k.

▶ **Question 2** Soit $\varphi = (\bot \vee p_0) \to (\neg p_1 \wedge p_2 \vee p_0) \wedge (\top \to p_2 \vee p_3)$. Écrivez-la dans une variable `exemple : formule`. Déterminer sa taille et sa hauteur.

▶ **Question 3** Écrire une fonction `taille : formule -> int` qui retourne la taille de la formule en entrée, c'est-à-dire le nombre de nœuds de l'arbre syntaxique associé.

▶ **Question 4** Écrire une fonction `ind_var_max : formule -> int` qui calcule l'indice maximal des variables propositionnelles dans la formule en entrée. La fonction renverra -1 si la formule en entrée n'a pas de variable propositionnelle.

▶ **Question 5** Écrire une fonction `valeur_verite : formule -> valuation -> bool` qui, pour une formule φ et une valuation ν des variables propositionnelles de la formule, renvoie si $\nu \vDash \varphi$.

▶ **Question 6** Déterminer sa complexité en fonction de la taille de la formule en entrée.

Exercice 2 – Satisfiabilité par force brute

On peut aussi voir une valuation comme un nombre binaire (positif), en interprétant `true` comme 1 et `false` comme 0. Ainsi, à une valuation ν sur $\mathcal{P} = [\![0, n-1]\!]$, on associe l'entier $x = \sum_{k=0}^{n-1} \nu(k) 2^{n-1-k}$.
On peut donc itérer sur les valuations en les incrémentant en tant que nombre binaire positif : la première valuation sera `[|false; false; ...|]`, la suivante `[|false; ...; false; true]`, et la dernière `[|true; true; ...; true|]`.
On utilise dans cette partie une exception, utilisée comme suis :

OCaml
```
exception Debordement
(* déclaration d'une nouvelle exception sans arguments, on peut aussi
 ↪  faire des exceptions avec [exception E of int * int] *)

let () =
  if 0 = 0 then raise Nom_de_mon_exception
  else ()
```

▶ **Question 1** Écrire une fonction `incrementer_valuation : valuation -> unit` qui transforme *en place* la valuation en entrée en une valuation correspondant au nombre $x + 1$. S'il n'est pas possible de l'incrémenter, on lève l'exception `Debordement`.

▶ **Question 2** En déduire une fonction `satisfaisable_brute : formule -> bool` qui résout le problème SAT.

▶ **Question 3** De la même façon, écrire une fonction `equivalent : formule -> formule -> bool` qui vérifie que les deux formules en entrée sont équivalentes.

▶ **Question 4** Écrire la fonction `supprimer_impl : formule -> formule` qui enlève les implications de la formule grâce à l'équivalence $\varphi \to \psi \equiv \neg\varphi \vee \psi$ et la fonction `repousser_neg : formule -> formule` qui repousse les négations vers les variables propositionnelles.

★ **Question 5** Écrire une fonction `fnc : formule -> formule` qui calcule la forme normale conjonctive de la formule en entrée. La tester sur l'exemple.

Exercice 3 – Algorithme de Quine

Pour une formule de la logique propositionnelle, on peut toujours la *simplifier* pour obtenir une formule équivalente égale à $\top$, $\bot$ ou ne contenant ni l'une ni l'autre. Pour cela, il suffit d'utiliser les équivalences suivantes :

Définition 1 – Règles de simplification de Quine

$$\neg\top \equiv \bot \quad \varphi \vee \top \equiv \top \quad \varphi \vee \bot \equiv \varphi \quad \top \wedge \varphi \equiv \varphi \quad \bot \wedge \varphi \equiv \bot$$
$$\neg\top \equiv \bot \quad \top \vee \varphi \equiv \top \quad \bot \vee \varphi \equiv \varphi \quad \varphi \wedge \top \equiv \varphi \quad \varphi \wedge \bot \equiv \bot$$

▶ **Question 1** Compléter ces règles de simplification afin d'autoriser la simplification de $\to$.

▶ **Question 2** Écrire une fonction `simplifier_constantes : formule -> formule` qui renvoie une formule équivalente à l'entrée qui soit égale à $\top$, $\bot$ ou ne contient ni l'un ni l'autre. *On pourra supposer que les constantes éventuelles sont au niveau* 0 *ou* 1 *de l'arbre syntaxique. Ainsi, la fonction n'aura pas besoin d'effectuer des appels récursifs.*

▶ **Question 3** Écrire une fonction `substitution : formule -> int -> formule -> formule`, telle que `substitution phi k psi` est évalué en la formule $\varphi\{p_k \mapsto \psi\}$[a].

▶ **Question 4** Adapter cette fonction à l'aide de la question 2 en une fonction `sub_et_simp : formule -> int -> formule -> formule` telle que `substitution phi k psi` est évalué en une formule équivalente $\varphi\{p_k \mapsto \psi\}$ qui est soit égale à $\top$ ou $\bot$, soit n'en contient aucun des deux. *On suppose que ϕ ne contient ni $\top$ ni $\bot$, et que $\psi = \top$ ou $\bot$.*

▶ **Question 5** Appliquer la substitution à `exemple`, p_0 et la formule $p_1 \vee \neg p_2$.

L'idée de l'algorithme de Quine est la suivante :

1. on choisit une variable propositionnelle p,
2. on lui affecte une valeur de vérité (en pratique, en la substituant par une constante $\top$ ou $\bot$ dans la formule),
3. on simplifie la formule obtenue avec les règles de la Définition 1,
4. on applique récursivement l'algorithme sur la formule obtenue (s'il reste une variable propositionnelle à fixer). Si cela ne donne rien, on revient en arrière pour choisir l'autre valeur de vérité

Les étapes 1 à 4 décrivent un algorithme de retour sur trace pour déterminer la satisfiabilité d'une formule de la logique propositionnelle. Dans ce TP, on va plutôt construire explicitement l'*arbre de décision* associé à cet algorithme : un arbre dans lequel chaque nœud étiqueté par k modélise le choix de la valeur de vérité de p_k : le fils gauche correspond au cas $\nu_{p_k} =$ `vrai`, et le fils droit le cas $\nu_{p_k} =$ `faux`. Pour l'implémenter, on va y utiliser une stratégie récursive :

Pseudo-code

```
1 : Fonction QUINE(φ)
2 :     ψ ← SIMPL(φ)
3 :     Si ψ ∉ {⊤, ⊥} alors
4 :         Soit p ∈ 𝒫 présente dans ψ,        ▷ qui existe, sinon ψ serait ⊤ ou ⊥
5 :         retourne l'arbre suivant :
              Noeud(p, QUINE(ψ{p ↦ ⊤}), QUINE(ψ{p ↦ ⊥}))
6 :     Sinon
7 :         retourne Feuille(ψ)                ▷ ψ ∈ {⊤, ⊥}
```

Maintenant, on a tout ce qu'il faut pour se lancer dans l'implémentation de l'algorithme de Quine, en utilisant les règles de la Définition 42. Pour représenter un arbre de décision, on va utiliser un type d'arbres binaires stricts, dont les feuilles sont étiquetées par des booléens.

OCaml

```
type arbre_decision =
  | Feuille of bool
  | Noeud of arbre_decision * int * arbre_decision
```

▶ **Question 6** Écrire une fonction `construire_arbre_decision : formule -> arbre_decision`.

▶ **Question 7** Proposer une heuristique[b] `choix_proposition : formule -> int` qui choisit une proposition qui sera la plus utile à substituer dans `construire_arbre_decision`.

▶ **Question 8** En déduire une fonction `construire_arbre_decision : formule -> arbre_decision` qui à une formule associe un arbre de décision.

▶ **Question 9** En choisissant toujours la proposition de plus grand index dans la formule d'entrée, proposer une suite de formules $\varphi_0, \varphi_1, \dots$ telle que l'arbre de décision calculé est de taille exponentielle.

▶ **Question 10** Écrire une fonction `satisfaisable_quine : formule -> bool` déterminant si une formule est satisfaisable.

▶ **Question 11** Déterminer sa complexité temporelle et spatiale.

▶ **Question 12** Écrire une fonction `modele_opt : formule -> valuation option` qui renvoie un modèle de l'entrée (`None` s'il n'en existe pas).

▶ **Question 13** Écrire une fonction `nombre_modeles : formule -> int` dénombrant les modèles d'une formule. On supposera que l'ensemble des variables propositionnelles est $\{p_0, \dots, p_k\}$ avec p_k la variable propositionnelle d'indice le plus grand apparaissant dans la formule en entrée.

a. C'est-à-dire la formule dans laquelle on a remplacé toutes les occurrences de p_k par la formule ψ.

b. « une fonction qui choisit une proposition en regardant de loin la formule, cette proposition étant choisie intelligemment, intelligemment n'ayant pas de définition particulière ici »