TP1 - Révisions de logique propositionnelle

L'objectif de ce TP est de revoir les notions liées au codage d'une formule de la *logique propositionnelle* et de déterminer sa *satisfiabilité* à l'aide de l'algorithme de Quine. La notion de *smart constructor* est introduite afin de simplifier une formule logique dès que possible, sans l'évaluer. Enfin, le format de fichier DIMACS est présenté et adopté pour traiter un problème de satisfiabilité défini par un fichier de ce type.
Le langage de programmation est OCaml.

1 Formules propositionnelles

1.1 Types

Dans tout les programmes, on adopte les types suivants. En particulier, chaque variable propositionnelle est identifiée par un entier unique strictement positif. Si une formule contient n variables propositionnelles, elles sont identifiées par les entiers $1, 2, \dots, n$. Nous verrons plus loin l'intérêt de ce choix avec les fichiers DIMACS.

Programme 1

```
type binop = And | Or | Imp

type fmla =
  | Top
  | Bot
  | Var of int (* dans 1..n *)
  | Not of fmla
  | Bin of binop * fmla * fmla
```

1.2 Évaluation d'une formule

Une valuation est définie par un *tableau* de taille égale au nombre de variables propositionnelles d'une formule.

Question 1.
□ **1.1.** Écrire une fonction `eval : bool array -> fmla -> bool` qui renvoie la valuation d'une formule logique, c'est-à-dire soit `true`, soit `false`.
□ **1.2.** Tester votre solution sur des exemples pertinents.

1.3 Variable propositionnelle maximale

Les variables propositionnelles étant identifiées par des entiers naturels non nuls consécutifs, on aura besoin de déterminer le plus grand de ces entiers.

Question 2.
□ **2.1.** Écrire une fonction `varmax : fmla -> int` qui renvoie cet entier.
□ **2.2.** Tester votre solution sur des exemples pertinents.

1.4 Substitution

Le cours a défini la notion de substitution d'une variable propositionnelle par une formule logique.

Question 3.
□ **3.1.** Écrire une fonction `subst : int -> fmla -> fmla -> fmla` telle que `(subst i ps f)` remplace, dans la formule logique `f`, la variable propositionnelle identifiée par l'entier `i` par la formule logique `ps`.
□ **3.2.** Tester votre solution sur des exemples pertinents.

1.5 Raccourcis et constructions

Pour définir des formules logiques, les notations introduites par la définition des types sont parfois peu pratiques. On peut leurs préférer une syntaxe plus légère pour définir les formules. On propose les fonctions suivantes dont l'intérêt sera également de permettre d'effectuer des simplifications comme le verrons plus loin. Par exemple simplifier `Bin(Or, f, Top)` en `f`.

Programme 2

```
let f_top = Top
let f_bot = Bot
```

```
let f_var i = Var i
let f_not f = Not f
let f_or f1 f2 = Bin (Or, f1, f2)
let f_and f1 f2 = Bin (And, f1, f2)
let f_imp f1 f2 = Bin (Imp, f1, f2)
let f_xor f1 f2 = f_or (f_and f1 (Not f2)) (f_and (Not f1) f2)
let f_equiv f1 f2 = f_and (f_imp f1 f2) (f_imp f2 f1)
```

1.6 Smart-constructors

Un *smart constructor* est une fonction qui se comporte comme un constructeur du type `fmla` mais applique éventuellement des simplifications. Par exemple, le *smart constructor* de la conjonction reçoit deux formules `phi1` et `phi2` et renvoie `Bin(And, phi1, phi2)`, sauf s'il est certain que le résultat sera équivalent à $\top$, $\bot$ ou à l'une des deux sous-formules `phi1` ou `phi2`, auquel cas il renvoie directement la formule simplifiée à la place de la conjonction. La simplification consiste alors à appliquer le *smart constructor* correspondant à chaque constructeur de la formule, en partant des feuilles pour s'assurer que les simplifications puissent s'enchaîner en cascade.

Question 4.
□ **4.1.** Écrire des *smart constructors* associés aux connecteurs $\neg$, $\wedge$, $\vee$, $\rightarrow$.

```
smart_not : fmla -> fmla
smart_and : fmla -> fmla -> fmla
smart_or : fmla -> fmla -> fmla
smart_imp : fmla -> fmla -> fmla
```

□ **4.2.** Tester votre solution sur des exemples pertinents.

1.7 Simplication d'une formule

Certaines formules peuvent être rapidement simplifiées sans avoir à les évaluer entièrement. Par l'exemple, `Bin(Or, f, Top)` se simplifie en `f`. On définit ci-dessous une fonction qui effectue ces simplifications.

Question 5.
□ **5.1.** En utilisant les *smart constructors*, écrire une fonction `simplify : fmla -> fmla` qui effectue ces simplfications, quand elles sont possibles.
□ **5.2.** Tester votre solution sur des exemples pertinents.

2 Algorithme de Quine

L'algorithme de Quine cherche à déterminer la satisfiabilité d'une formule en construisant un *arbre de décision*. L'idée est de *substituer* chaque variable propositionnelle, par $\top$ d'une part, et $\bot$ d'autre part, chacune de ces options représentant une branche de l'arbre. Chaque substitution, en plus de décrire un choix d'une branche dans l'arbre de décision, donne une *évaluation partielle* de la formule. Si à un nœud de l'arbre, on constate que l'évaluation partielle est déjà fausse, il est inutile de poursuivre l'exploration de cette branche.

Illustrons sa mise en œuvre avec la formule $\varphi = (x \wedge \neg z) \rightarrow (\neg x \wedge \neg(y \wedge \neg z))$. On choisit de d'abord substituer x, puis, si nécessaire, de substituer y dans les formules obtenues et enfin, si nécessaire encore, de substituer z dans les dernières formules. Après chaque substitution, on simplifie la formule à l'aide des équations suivantes, qui peuvent être déduite des propriétés rappelées en annexe.

$$\begin{array}{lll} \varphi \wedge \bot \equiv \bot \wedge \varphi \equiv \bot & \varphi \wedge \top \equiv \top \wedge \varphi \equiv \varphi & \neg\top \equiv \bot \\ \varphi \vee \top \equiv \top \vee \varphi \equiv \top & \varphi \vee \bot \equiv \bot \vee \varphi \equiv \varphi & \neg\bot \equiv \top \\ \varphi \rightarrow \top \equiv \bot \rightarrow \varphi \equiv \top & \top \rightarrow \varphi \equiv \varphi & \varphi \rightarrow \bot \equiv \neg\varphi \end{array}$$

- On obtient tout d'abord avec x :

$$\varphi_1 = \varphi^{\{x \leftarrow \top\}} = \neg z \rightarrow \bot \qquad \varphi_2 = \varphi^{\{x \leftarrow \bot\}} = \top$$

  On peut déjà constater que la substitution $\{x \leftarrow \bot\}$ suffit à satisfaire la formule. Mais en pratique, selon le code qui met en œuvre l'algorithme de Quine, cette solution ne sera pas forcément découverte avant d'avoir exploré la branche correspondant à φ_1. Poursuivons donc les substitutions.

- On obtient alors, en substituant y dans φ_1 :

$$\varphi_{11} = \varphi_1^{\{y \leftarrow \top\}} = (\neg z \rightarrow \bot) \qquad \varphi_{12} = \varphi_1^{\{y \leftarrow \bot\}} = (\neg z \rightarrow \bot)$$

- Un dernière substitution, de z dans φ_{11} et φ_{12}, donne :

$$\varphi_{111} = \varphi_{11}^{\{z \leftarrow \top\}} = \top \qquad \varphi_{112} = \varphi_{11}^{\{z \leftarrow \bot\}} = \bot$$
$$\varphi_{121} = \varphi_{11}^{\{z \leftarrow \top\}} = \top \qquad \varphi_{122} = \varphi_{11}^{\{z \leftarrow \bot\}} = \bot$$

L'ensemble est résumé à la figure 1.

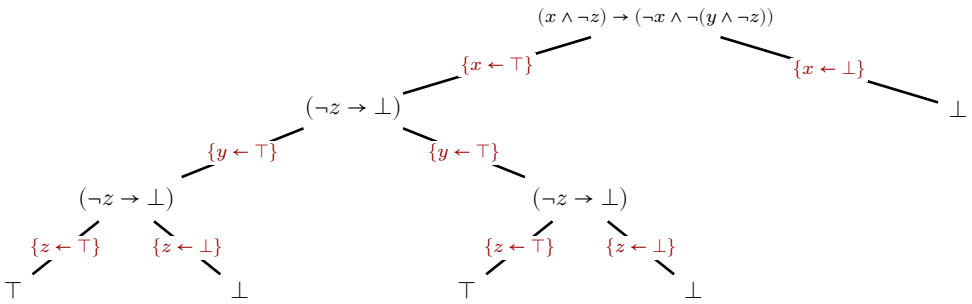

Figure 1 – Arbre de décision de $((x \wedge \neg z) \to (\neg x \wedge \neg(y \wedge \neg z)))$.

Question 6.
□ **6.1.** Écrire une fonction `quine_sat : fmla -> bool` qui met en œuvre l'algorithme de Quine.
□ **6.2.** Tester votre solution sur des exemples pertinents.

3 Fichier DIMACS

Les solveurs SAT sont des programmes qui résolvent un problème SAT. Les formules sous forme CNF ou DNF y sont décrites suivant des règles simples dans un fichier au format dit DIMACS, fichier texte dont les premières lignes sont des commentaires signalés par la présence d'un caractère `c` en début de chaque ligne. La ligne suivante commence par un `p` et décrit la nature du problème en précisant la forme normale codéee : `cnf` ou `dnf`. Deux entiers indiquent le nombre de variables propositionnelles et le nombre de clauses. Viennent ensuite les descriptions de chaque clause sous la forme de suites d'entiers. Le format adopte la convention suivante : une variable propositionnelle est représentée par un entier strictement positif ; sa négation par l'entier opposé. La fin d'une ligne est indiquée par la présence d'un `0`. Ci-dessous, un exemple de fichier DIMACS qui décrit la formule suivante.

$$(\neg x) \vee (\neg x \wedge \neg y) \vee (\neg x \wedge z) \vee (\neg y \wedge z) \vee (z)$$

fichier au format DIMACS

```
c x y z ~x ~y ~z
c 1 2 3 -1 -2 -3
p cnf 3 5
-1  0
-1 -2  0
-1  3  0
-2  3  0
 3  0
```

Question 7.
□ **7.1.** Écrire un programme qui lit un fichier DIMACS et qui construit une formule logique de type `fmla`.
□ **7.2.** Tester votre solution sur des exemples pertinents à l'aide de vos fonctions définies précédemment.

4 Annexe

4.1 Équivalences sémantiques propositionnelles

Sont rappelées ci-dessous quelques équivalences sémantiques de formules logiques propositionnelles. En particulier, les lois de de Morgan lient la conjonction à la négation d'une disjonction, et inversement.

$$\begin{aligned} \neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi && \text{(de Morgan)} \\ \neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi && \text{(de Morgan)} \end{aligned}$$

L'implication peut se décomposer en une disjonction et une négation. Elle est inversée par négation, et a également une interaction avec la conjonction.

$$\begin{aligned} \varphi \to \psi &\equiv \neg\varphi \vee \psi && \text{(implication)} \\ \varphi \to \psi &\equiv \neg\psi \to \neg\varphi && \text{(contraposition)} \\ (\varphi \wedge \psi) \to \theta &\equiv \varphi \to (\psi \to \theta) && \text{(curryfication)} \end{aligned}$$

Cette dernière équivalence est notamment à rapporcher de l'écriture curryfiée des fonctions en OCaml.

En *logique classique*, une variable propositionnelle x et sa négation $\neg x$ ne peuvent pas être toutes les deux vraies. Ce résultat constitue le principe de *non-contradiction*. En outre, si x est faux, alors $\neg x$ est vrai, et donc nécessairement un parmi x et $\neg x$ doit être vrai (principe du *tiers exclu*).
Si une négation inverse la signification d'une formule, une deuxième négation rétablit la sémantique d'origine.

$$\begin{aligned} \varphi \wedge \neg\varphi &\equiv \bot && \text{(non-contradiction)} \\ \varphi \vee \neg\varphi &\equiv \top && \text{(tiers exclu)} \\ \neg\neg\varphi &\equiv \varphi && \text{(double négation)} \end{aligned}$$

Enfin, on a les équivalences sémantiques suivantes dont la démonstration est laissée au soin du lecteur.

$$\begin{aligned} \varphi \wedge \top &\equiv \varphi && \text{(élément neutre)} \\ \varphi \vee \bot &\equiv \varphi && \text{(élément neutre)} \\ \varphi \wedge \bot &\equiv \bot && \text{(élément absorbant)} \\ \varphi \vee \top &\equiv \top && \text{(élément absorbant)} \\ \varphi \wedge \psi &\equiv \psi \wedge \varphi && \text{(commutativité)} \\ \varphi \vee \psi &\equiv \psi \vee \varphi && \text{(commutativité)} \\ (\varphi \wedge \psi) \wedge \theta &\equiv \varphi \wedge (\psi \wedge \theta) && \text{(associativité)} \\ (\varphi \vee \psi) \vee \theta &\equiv \varphi \vee (\psi \vee \theta) && \text{(associativité)} \\ \varphi \wedge (\psi \vee \theta) &\equiv (\varphi \wedge \psi) \vee (\varphi \wedge \theta) && \text{(distributivité)} \\ \varphi \vee (\psi \wedge \theta) &\equiv (\varphi \vee \psi) \wedge (\varphi \vee \theta) && \text{(distributivité)} \end{aligned}$$

4.2 Lecture dans un fichier

On rappelle ci-dessous quelques fonctions OCaml de lecture dans un fichier texte.

- `open_in : string -> in_channel` qui prend en argument un nom de fichier (c'est-à-dire un chemin vers un fichier) et renvoie une valeur de type `in_channel`.
- `input_line : in_channel -> string` qui lit des caractères dans le `in_channel` jusqu'à tomber sur un retour à la ligne, et renvoie la chaîne de caractères lue (sans le `"\n"`) ;
- `close_in : in_channel -> unit` pour fermer le fichier une fois qu'on a fini de l'utiliser.

Une valeur de type `in_channel` se comporte comme un *flux*, c'est-à-dire que les lignes sont *consommées* au fur et à mesure qu'elles sont lues. Si l'on appelle deux fois de suite `input_line` sur le même `in_channel`, on obtient une ligne puis la suivante. Par ailleurs, si on appelle `input_line` sur un `in_channel` *épuisé*, c'est-à-dire dans lequel il n'y a plus rien à lire, l'exception `End_of_file` est levée. C'est en fait le seul moyen de savoir (en utilisant ces fonctions) qu'on a terminé la lecture du fichier.