Exercice 1 : Syntaxe et sémantique de la logique propositionnelle

1. Environnements propositionnels

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'un couple (a, b) dans une telle liste indique l'association de la valeur b à la clé a. La manipulation d'une telle liste d'association doit assurer l'invariant suivant : si (a, b) et (c, d) sont deux couples disjoints se trouvant dans la liste, alors $a \neq c$. Autrement dit les premières composantes des couples de la liste sont deux-à-deux disjointes. Aussi la liste d'association `[(p, true); (q, false)]` représente l'environnement propositionnel : $(p \mapsto \mathsf{V}, q \mapsto \mathsf{F})$.

Compagnon. On fournit dans le fichier compagnon :
- une définition d'un type `type env_prop` permettant la représentation de tels environnements propositionnels ;
- une fonction `print_env_prop` permettant l'affichage des environnements propositionnels.

De plus, à titre d'exemple, on y trouvera la définition d'une fonction `mem_assoc : env_prop -> string -> bool` prenant en arguments un environnement propositionnel et une variable et indiquant si la variable en question apparaît dans le domaine de définition de l'environnement propositionnel.

Q. 1 Définir une fonction `add_assoc : env_prop -> string -> bool -> env_propo` prenant en arguments un environnement propositionnel, une variable propositionnelle (une chaîne de caractères, *que l'on supposera ne pas déjà se trouver dans la liste d'association*) et un booléen et telle que l'appel `(add_assoc mu p b)` produit une liste d'association en tout point identique à `mu` mais contenant en plus l'association `(p, b)`. Remarquons que nous ne sommes pas contraints sur le placement du couple `(p, b)` dans la liste et pouvons donc l'insérer à l'endroit qui nous convient le mieux.

Q. 2 Définir une fonction `read_assoc : env_prop -> string -> bool` prenant en arguments un environnement propositionnel, une variable propositionnelle (que l'on supposera être la première composante d'un couple se trouvant dans l'environnement propositionnel) et calculant le booléen associé à cette variable propositionnelle dans la liste d'association. Si une telle valeur n'existe pas, on lèvera l'exception prédéfinie `Not_found` au moyen de l'appel `raise Not_found`.

Q. 3 Définir une fonction `set_assoc : env_prop -> string -> bool -> env_prop` prenant en arguments un environnement propositionnel `mu`, une variable propositionnelle `p` et un booléen `b` et ajoutant l'association $\mathtt{p} \mapsto \mathtt{b}$ à la liste, *si la variable propositionnelle est déjà présente dans la liste on modifiera la valeur qui lui est associée*.

Q. 4 ★ Définir une fonction `ae : string list -> env_prop list` générant tous les environnements propositionnels sur les variables propositionnelles passées en arguments. On pourra utiliser pour cela la relation de récurrence ci-dessous.

$$\begin{aligned} \texttt{(ae [])} &= \{()\} \\ \texttt{(ae (p :: l))} &= \{(p \mapsto \mathsf{V}) \uplus \mu \mid \mu \in \texttt{(ae l)}\} \\ &\quad \cup \{(p \mapsto \mathsf{F}) \uplus \mu \mid \mu \in \texttt{(ae l)}\} \end{aligned}$$

Où $(p \mapsto \mathsf{V}) \uplus \mu$ est une notation ♣ désignant l'ajout de l'association $(p \mapsto \mathsf{V})$ à l'environnement μ. Il pourra donc être opportun de définir une fonction auxiliaire `ajout_assoc_a_tous_env : string -> bool -> env_prop_list -> env_prop_list` prenant en arguments une variable propositionnelle `p`, un booléen `b` et une liste d'environnements propositionnels `l` et calculant la liste des environnements propositionnels de `l`, tous augmentés de l'association (p, b).

2. Syntaxe de la logique propositionnelle

Syntaxe de la logique propositionnelle. On rappelle, qu'étant donné un ensemble de variables propositionnelles $\mathscr{P}$, on définit les formules de la logique propositionnelle $\mathscr{F}$ comme étant le plus petit ensemble obtenu inductivement de la manière suivante :

- $\bot, \top \in \mathscr{F}$ [2]
- Si $p \in \mathscr{P}$, $p \in \mathscr{F}$
- Si $G \in \mathscr{F}$ et $H \in \mathscr{F}$ alors $G \vee H \in \mathscr{F}$, $G \wedge H \in \mathscr{F}$, $G \to H \in \mathscr{F}$, $G \leftrightarrow H \in \mathscr{F}$
- Si $G \in \mathscr{F}$ alors $\neg G \in \mathscr{F}$

Compagnon. On fournit dans le fichier compagnon :

- Une définition du type `formule` permettant la représentation de telles formules ;
- Une fonction `affiche: formule -> unit` d'affichage des formules de la logique propositionnelle.

Q. 5 Définir une fonction `empty_vars : formule -> bool` prenant en argument une formule G de la logique propositionnelle et retournant si cette formule est sans variable propositionnelle ou non. Par exemple pour la formule $(p \wedge q)$ la fonction devra s'évaluer à `false`, mais pour la formule $\top \to (\neg\bot)$ la fonction devra s'évaluer à `true`.

Q. 6 Définir une fonction `taille : formule -> int` prenant en argument une formule G de la logique propositionnelle et retournant la taille de cette formule.

3. Sémantique.

Q. 7 Définir une fonction `interprete_no_vars : formule -> bool` prenant en argument une formule de la logique propositionnelle *sans variables* et retournant l'interprétation de cette formule (dans un environnement quelconque, puisque l'on a vu en cours que si la formule ne contient pas de variables propositionnelles son interprétation ne dépend pas de l'environnement propositionnel).

Q. 8 Définir une fonction `interprete : formule -> env_prop -> bool` permettant l'interprétation d'une formule dans un environnement propositionnel. On lèvera une exception si l'environnement propositionnel ne contient pas toutes les variables apparaissant dans la formule.

Q. 9 Définir une fonction `vars : formule -> string list` prenant en argument une formule de la logique propositionnelle et retournant la liste *sans doublons* de ses variables propositionnelles.

Pour la suite il est nécessaire d'avoir traité la **Q. 4** pour continuer.

Q. 10 Définir une fonction `sat : formule -> env_prop option` ♡ permettant de résoudre le problème de la satisfiabilité d'une formule. On retournera donc `None` si la formule est insatisfiable et `Some(`μ`)` sinon, où μ est un modèle de la formule.

♣. classique

♡. Voir topo en fin d'énoncé pour type `'a option`

Q. 11 Définir une fonction `est_valide : formule -> true` permettant de tester si une formule est valide.

Q. 12 Définir une fonction `est_cons_semantique : formule -> formule -> bool` permettant de tester si une formule est conséquence sémantique d'une autre.

Q. 13 Définir une fonction `equiv : formule -> formule -> bool` permettant de tester si deux formules sont équivalentes.

Q. 14 Définir une fonction `models : formule -> env_prop set` donnant l'ensemble des modèles d'une formule.

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

Cet exercice nécessite d'avoir fait la **Q. 4** de l'exercice précédent. On définit le type ci-dessous.

```
type fct_bool = env_prop -> bool
```

Q. 1 Donner une fonction `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. On pourra donc travailler en plusieurs temps :

- Écriture d'une fonction prenant en argument un couple (p, b) et retournant le littéral p si $b =$ `true` et $\neg p$ sinon.
- Écriture d'une fonction prenant en argument un environnement propositionnel μ et retournant une clause conjonctive dont μ est le seul modèle (notons H_μ une telle formule).
- Écriture de la fonction `formule_of_fct_bool` construisant la disjonction de toutes les clauses H_μ pour μ dans l'image réciproque par la fonction booléenne de V (autrement dit : les environnements pour lesquels la fonction booléenne en question vaut vrai).

Q. 2 Donner une fonction `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* ♣ 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 alors 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 ;

♣. C'est une définition par l'exemple, la notion de règle de réécriture n'est pas à votre programme

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 $\wedge$. En effet en utilisant cette règle la formule $(p \wedge (q \wedge r))$ est transformée en $\neg(\neg p \vee \neg(q \wedge r))$, elle-même transformée en $\neg(\neg p \vee \neg\neg(\neg q \vee \neg r))$.
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 Donner les arguments justifiant de la terminaison et de la correction de votre système de réécriture.

♣. Voir topo sur le type `'a option` en fin d'énoncé

Le type `'a option` en OCaml

Il n'est pas rare que l'on souhaite étendre un type OCaml `t` pour y ajouter une valeur. Considérons par exemple l'ensemble des entiers `int` et la fonction de division :

```
let ma_division (x: int) (y: int) : int =
  print_string "je fais une division";
  (x / y)
```

Cette fonction est bien définie mais lève une erreur quand on l'appelle avec une valeur nulle pour l'entier `y`. Aussi on souhaite ajouter une valeur spéciale, qui ne soit pas un entier, aux entiers, on pourra alors retourner cette valeur dans le cas d'une division par zéro.
OCaml fournit pour cela le type prédéfini `'a option`, défini de la manière suivante :

```
type 'a option =
  | None
  | Some of 'a
```

Aussi un élément de type `'a option` est : ou bien la valeur `None`, ou bien la valeur `Some(x)` où `x` est de type `'a`. On peut alors redéfinir notre division de la manière suivante :

```
let ma_division (x: int) (y: int) : int option =
  print_string "je fais une division";
  if y = 0 then None
  else Some(x / y)
```

Noter le type de retour de la fonction.

```
# ma_division 4 2 ;;
je fais une division- : int option = Some 2
# ma_division 4 0 ;;
je fais une division- : int option = None
```

Bien sûr, en tant que type défini par énumération, on peut raisonner par disjonction sur une valeur de type `'a option` au moyen de la construction `match ... with ...`. Ainsi on peut définir la fonction ci-dessous effectuant une division par 2 au moyen de la fonction `ma_division`.

```
let division_par_2 (x: int) : int =
  match ma_division x 2 with
  | None -> failwith "ma_division : division par zéro"
  | Some(r) -> r
```

Noter le type de retour de la fonction.

```
# division_par_2 3 ;;
je fais une division- : int = 1
```

Une utilisation classique du type `'a option` est en substitution au type `bool`. Considérons par exemple une fonction de recherche : *existe-t-il un élément dans le tableau vérifiant telle propriété ?* Une telle fonction aurait un type de retour booléen : oui ou non. On pourrait toutefois enrichir une telle fonction en : *existe-t-il un élément dans le tableau vérifiant telle propriété ? Si oui le retourner.* Une telle fonction aurait un type de retour optionnel : non (`None`) ou oui et voici l'élément (`Some(élément)`).