On définit une type `'a prop` comme suit :

```
type 'a prop =
  | Top
  | Bot
  | V of 'a
  | Et of 'a prop * 'a prop
  | Ou of 'a prop * 'a prop
  | Non of 'a prop
  | Implique of 'a prop * 'a prop
```

Le constructeur `V` permet de représenter des variables. Par exemple, une variable a pourra être représentée par la valeur `V "a"`. Les autres constructeurs permettent de représenter les constantes $\top$ et $\bot$ ainsi que les connecteurs logiques : $\wedge, \vee, \neg, \rightarrow$.

Un fichier `fenelon_tp21_eleve.ml` est à télécharger sur la page web du cours. Il contient le type ci-dessus, l'entête des fonctions à compléter, ainsi que des exemples sur lesquels tester vos fonctions.

1 Introduction

1. On définit la hauteur d'une expression logique comme la hauteur de son arbre syntaxique associé. Écrire une fonction `hauteur : 'a prop -> int` calculant la hauteur d'une expression logique.

   Exemple
   ```
   # hauteur e ;;
   - : int 3
   ```

2. la longueur (ou taille) d'une expression logique est le nombre de nœuds de son arbre syntaxique. Écrire une fonction `longueur : 'a prop -> int` calculant la longueur d'une expression logique.

   Exemple
   ```
   # longueur e ;;
   - : int 8
   ```

2 Un testeur de tautologies

3. Écrire une fonction `variables : 'a prop -> 'a list` renvoyant la liste des variables apparaissant dans une expression logique.
   Attention : chaque variable de l'expression doit apparaître une et une seule fois dans la liste.

   Exemple
   ```
   # variables e ;;
   - : string list = ["b", "a", "c"]
   ```

On va maintenant écrire une fonction d'évaluation d'une expression logique. Outre l'expression, cette fonction doit également prendre en entrée une valuation sur un ensemble de variables (contenant au moins toutes les variables de l'expression). Cette valuation est donnée sous la forme d'une liste de couples `(nom_de_variable, booleen)`, elle est donc de type `('a * bool) list`.

4. Écrire une fonction `association : 'a -> ('a * 'b) list -> 'b` prenant en entrée un élément x de type `'a` et une liste `l` de type `('a * 'b) list`, et retournant y tel que le couple (x, y) soit le premier couple apparaissant dans `l` avec comme première composante x.

5. Écrire une fonction `evalue : 'a prop -> ('a * bool) list -> bool` évaluant une expression logique avec une valuation.

Exemple

```
# evalue e ["a", true; "b", false; "c",true] ;;
- : bool = false
```

6. Écrire une fonction `valuations v : 'a list -> ('a * bool) list list` générant toutes les valuations sur un ensemble de variables donné en entrée.
Indication : On pourra utiliser `List.map`.

Exemple

```
# valuations ["a"; "b"] ;;
- : (string * bool) list list =
[[("a", true); ("b", true)]; [("a", true); ("b", false)];
 [("a", false); ("b", true)]; [("a", false); ("b", false)]]
```

7. En déduire `est_tautologie : 'a prop -> bool` renvoyant un booléen suivant si l'expression logique prise en entrée est une tautologie ou non.

Exemple

```
# est_tautologie f1 ;;
- : bool = false
# est_tautologie f2 ;;
- : bool = false
# est_tautologie f3 ;;
- : bool = true
```

8. Parmi les formules suivantes, laquelle est une tautologie ?
   - $(((P \to Q) \to P) \to P)$;
   - $(((P \to Q) \to P) \to Q)$.

   Cette tautologie s'appelle la loi de Pierce.

3 Amélioration : affichage d'un contre-exemple

En utilisant une exception, il est possible d'améliorer la fonction précédente.

9. Écrire une fonction `affiche : (string * bool) list -> unit` affichant à l'écran une valuation dans le cas où les variables sont de type `string`.
On utilisera les fonctions `print_int`, `print_string`, `print_newline`, ... Et on affichera 0 ou 1 plutôt que des booléens.

Exemple

```
# affiche ["a", true; "b", false; "c", true] ;;
a --> 1    b --> 0    c --> 1
- : unit = ()
```

On déclare l'exception suivante, de type `(string * bool) list` `exn` :

```
exception Contre_ex of (string * bool) list ;;
```

Rappel : le mécanisme pour lever une exception est le suivant :

- On **lève** une exception avec la fonction `raise`. Par exemple ici, avec `l` une `(string * bool) list`, on ferait :

  ```
  raise (Contre_ex l)
  ```

- Tant qu'elle n'est pas **rattrapée**, l'exception est **propagée** (on sort de toutes les fonctions appelées, jusqu'au programme principal, avec l'exception).
- Pour **rattraper** une exception, on encadre le code pouvant la produire avec `try ... with` :

  ```
  try
    code
  with Contre_ex l -> actions(l)
  ```

  Ce mécanisme permet d'effectuer `actions(l)` si l'exception `Contre_ex l` est levée durant l'évaluation de `code`. En fait, on peut filtrer sur plusieurs exceptions, mais on n'aura pas besoin de ça dans ce TP.

10. Écrire une fonction `est_tautologie' v : string prop -> bool` semblable à celle de la question 7, mais qui affiche un contre-exemple si la formule n'est pas une tautologie.
On lèvera une exception si on trouve une valuation qui ne satisfait pas l'expression logique, et on rattrapera l'exception pour afficher la valuation. On n'oubliera pas de renvoyer quand même `false` dans ce cas.

Exemple

```
# est_tautologie' f2 ;;
ça ne marche pas avec la valuation :
b --> 1    c --> 1    a --> 0
- : bool = false
```

4 Résolution naïve du problème SAT

11. Reprendre les questions précédentes avec une fonction `est_satisfiable : string prop -> bool` affichant à l'écran un **certificat** si la formule est satisfiable (i.e. une valuation satisfaisant la formule).

Remarque : la complexité de `est_tautologie` et `est_satisfiable` n'est pas "optimale" à cause de l'utilisation de listes pour représenter les valuations. Une utilisation de **tables de hachage** mènerait à une meilleure complexité.

5 Algorithme de Quine

On se propose maintenant d'implémenter l'algorithme de Quine vu en cours :

- on commence par simplifier la formule au maximum, en utilisant les **règles de simplifications de Quine**, jusqu'à ce qu'aucune règle ne puisse s'appliquer ;
- si la formule simplifiée φ obtenue est `Bot` ou `Top`, l'algorithme est terminé ;
- sinon, il reste au moins une variable v, et on effectue une opération de **branchement** qui produit les deux formules $\varphi[\bot/v]$ et $\varphi[\top/v]$, et on continue l'algorithme sur ces deux formules.

L'algorithme de Quine produit ainsi un **arbre de Quine**, dont toutes les feuilles sont soit `Top`, soit `Bot`. La formule de départ est valide (resp. satisfiable) si et seulement si toutes les feuilles sont des `Top` (resp. au moins une feuille est `Top`).

12. Écrire une fonction `subs : 'a prop -> 'a -> 'a prop -> 'a prop` telle que `subs phi v b` renvoie la formule $\varphi[\bot/v]$ ou $\varphi[\top/v]$ (selon si $b =$ `Bot` ou `Top`).
13. Écrire une fonction `simpl : 'a prop -> 'a prop` implémentant les **règles de simplification de Quine**.

On définit les arbres des Quine par le type suivant :

```
type 'a arbre_quine =
  | Valide
  | Invalide
  | Branchement of 'a * 'a arbre_quine * 'a arbre_quine
```

14. Écrire une fonction `quine : 'a prop -> 'a arbre_quine` prenant en entrée une formule logique et renvoyant un arbre de Quine pour cette formule.

Exemple

```
# quine f2 ;;
- : string arbre_quine =
Branchement ("b",
 Branchement ("c", Branchement ("a", Valide, Invalide), Valide),
Valide)
# quine f3 ;;
- : string arbre_quine =
Branchement ("b", Branchement ("a", Valide, Valide), Valide)
```