

## TP n°4

### Logique propositionnelle en OCAML

Dans les exercices ci-dessous, les fonctions du module `List` sont librement utilisables (`List.map`, `List.exists`, `List.for_all`...)

#### 1 Calcul propositionnel

On considère les formules de la logique propositionnelle construites avec des variables propositionnelles et les connecteurs  $\neg$ ,  $\wedge$  et  $\vee$ . Pour représenter ces formules en Ocaml, on se servira du type suivant :

```
type formule = Var of string
              | Neg of formule
              | Et  of formule * formule
              | Ou  of formule * formule;;
```

Par exemple, la formule  $p \wedge \neg(q \vee r)$  sera représentée par :

```
Et (Var "p", Neg(Ou(Var "q", Var "r")));;
```

**Exercice 1.** Écrire une fonction

```
string_of_formule : formule -> string
```

qui, étant donnée une formule, renvoie une chaîne de caractères qui représente cette formule écrite avec des parenthèses encadrant toutes les sous-formules de connecteur binaire. Par exemple, pour la formule ci-dessus, on obtiendra `"(p Et Neg (q Ou r))"`.

**Exercice 2.** Écrire une fonction

```
list_of_vars : formule -> string list
```

qui, étant donné une formule, renvoie une liste sans répétitions de tous ses noms de variables. Par exemple, `list_of_vars (Et (Ou(Var "p", Var "q"), Var "q"))` renverra `["p"; "q"]` à l'ordre des éléments près.

Pour construire l'union sans répétitions des noms de variables de deux sous-formules, vous pouvez vous servir de la fonction `union_sorted` du TP 2 – ou écrire toute autre fonction auxiliaire produisant cette union.

#### 2 Evaluation des formules

Appelons *affectation* toute liste d'association de type `(string * bool) list`. Une affectation permet de spécifier les valeurs de vérité des variables d'une formule. Chaque variable apparaît une fois au plus dans une affectation.

Par exemple, une affectation possible pour la formule `Et (Var "p", Var "q")` est `[("p", true), ("q", false)]`.

La valeur de vérité d'une formule  $f$  relativement à une affectation  $e$  est :

- indéfinie, si les variables apparaissant dans  $f$  n'apparaissent pas toutes dans  $e$ ,
- sinon, cette valeur de vérité est obtenue en remplaçant dans  $f$  :
  - les variables par les valeurs spécifiées par  $e$ ,
  - les connecteurs logiques par les opérations qui leur sont naturellement associées.

**Exercice 3.** Écrire une fonction

```
eval_formule : formule -> (string * bool) list -> bool
```

calculant la valeur de vérité d'une formule relativement à une affectation donnée. Par exemple,

`eval_formule (Et (Var "p", Var "q")) [("p", true); ("q", false)]` donnera **false**.

Vous pouvez utiliser `List.assoc v e` qui renvoie la partie droite  $b$  du premier couple de  $e$  de la forme  $(v, b)$  s'il existe, et déclenche une exception si ce couple est introuvable.

### 3 Formules satisfiables et tautologies

Une formule de la logique propositionnelle est appelée une *tautologie*, si elle s'évalue à **true** pour tout affectation spécifiant les valeurs de vérité de toutes ses variables. Une formule qui s'évalue à **true** pour *au moins* une affectation est dite *satisfiable*.

**Exercice 4.** Écrire une fonction polymorphe

```
add_to_all : 'a -> ('a list) list -> ('a list) list
```

telle que `add_to_all x [l1; ... ; ln]` renvoie la liste de listes `[x::l1; ... ; x::ln]`. Noter que cette fonction est très simple à écrire avec `List.map`.

**Exercice 5.** En vous servant de la fonction `add_to_all`, écrire

```
affectations_vars : string list -> ((string * bool) list) list
```

qui, étant donnée une liste de noms de variables, construit liste de tous les affectations possibles associés. Par exemple, `affectations_vars ["q"]` donnera

```
[["q", false]];
[["q", true] ]
```

et `affectations_vars ["p"; "q"]` donnera

```
[["p", false); ("q", false)];
[["p", false); ("q", true) ];
[["p", true); ("q", false)];
[["p", true); ("q", true) ]]
```

En déduire une fonction

```
affectations : formule -> (string * bool) list list
```

qui étant donnée une formule donne une liste de tous les affectations associés à ses variables.

**Exercice 6.** A l'aide de la fonction `affectations`, de `List.exists` et de `List.for_all`, écrire

```
satisfaisable: formule -> bool
```

qui, étant donnée une formule, détermine si elle est satisfaisable, et

tautologie : formule -> bool

qui, étant donnée une formule, détermine si elle est une tautologie. Par exemple,

- satisfaisable (Et (Var "p", Neg(Var "p"))) donnera **false**.
- tautologie (Ou(Var "p", Neg(Var "p"))) donnera **true**.

## 4 Conséquences logiques et équivalences.

Si  $f$  et  $g$  sont deux formules de la logique propositionnelle, on dit que  $g$  est *conséquence logique* de  $f$  si, pour toute affectation pour laquelle  $f$  s'évalue à **true**,  $g$  s'évalue aussi à **true**. Les affectations considérées doivent spécifier à la fois les valeurs des variables de  $f$  et de celles de  $g$  – ces deux ensembles de variables peuvent être joints par exemple à l'aide de la fonction `union_sorted`. Si  $g$  est conséquence logique de  $f$  et  $f$  est conséquence logique de  $g$ , on dit que ces deux formules sont *équivalentes*.

**Exercice 7.** Écrire une fonction

consequence: formule -> formule -> bool

qui, étant données une formule  $f$  et une formule  $g$ , détermine si  $g$  est conséquence logique de  $f$ .  
En déduire une fonction

equivalentes: formule -> formule -> bool

qui détermine si ses deux arguments sont des formules équivalentes.

## 5 Forme normale conjonctive (★)

Dans la logique propositionnelle les propriétés suivantes sont vérifiées :

1. *Descente des négations vers les variables.* Les formules suivantes sont équivalentes :
  - $\neg\neg F$  et  $F$
  - $\neg(F \vee G)$  et  $(\neg F \wedge \neg G)$ ,
  - $\neg(F \wedge G)$  et  $(\neg F \vee \neg G)$ .
2. *Descente des  $\vee$  sous les  $\wedge$  par factorisation.* Les formules suivantes sont équivalentes :
  - $(F \vee (G \wedge H))$  et  $(F \vee G) \wedge (F \vee H)$ ,
  - $((F \wedge G) \vee H)$  et  $(F \vee H) \wedge (G \vee H)$ .

Une formule sera dite sous *forme normale conjonctive* si elle est, au parenthésage et à permutation près, de la forme  $F_1 \wedge \dots \wedge F_n$  où chaque  $F_i$  est de la forme  $(\neg a_1 \vee \dots \vee \neg a_m \vee b_1 \vee \dots \vee b_p)$ , où les  $a_i$  et  $b_j$  sont des variables propositionnelles.

**Exercice 8.** A l'aide des propriétés ci-dessus, écrire les fonctions suivantes. Chacune, à partir d'une formule, doit produire une formule équivalente de la forme spécifiée.

- `desc_neg : formule -> formule`. Cette fonction doit faire descendre les négations d'une formule vers ses variables, en simplifiant les négations doubles.
- `desc_ou : formule -> formule`. Cette fonction doit faire descendre les  $\vee$  sous les  $\wedge$  dans toute formule dans laquelle les négations ne s'appliquent qu'à des variables.
- `fnc : formule -> formule`. Cette fonction doit transformer toute formule en sa forme normale conjonctive.

Par exemple, `fnc (Ou (Neg (Et (Var "p", Var "q")), Et (Var "q", Var "r")))` donnera :

`Et (Ou (Ou (Neg (Var "p"), Neg (Var "q")), Var "q"),  
Ou (Ou (Neg (Var "p"), Neg (Var "q")), Var "r"))`