L2/LD – TP 5 – Formules propositionnelles avec variables

Tous les exercices de ce TP doivent être finies avant la prochaine séance de TP, éventuellement sur votre temps de travail personnel.

Vous devez rendre sous la forme d'un unique fichier source compressé au format .zip avant 23h30 la veille de votre prochain TP :

- **la fonction** `affichePF`
- **la fonction** `interpretation`

1 Introduction

L'objectif de ce TP est d'implanter en Caml la notion de formule de la logique des propositions.

La fonction `string_of_bool` Caml convertit un booléen en chaîne de caractères, ce qui permet de l'afficher, avec la fonction `print_string`.

```
À taper :   let p= false;;
Réponse :   val p : bool = false
À taper :   string_of_bool p;;
Réponse :   - : string = "false"
À taper :   print_string (string_of_bool p);;
Réponse :   false- : unit = ()
```

2 Représentation de formules propositionnelles avec variables

Il a été vu dans le TP précédent comment définir un type pour représenter des formules propositionnelles sans variables.

Nous souhaitons ici élargir ce type pour pouvoir représenter des formules propositionnelles avec variables. Il s'agit de formules construites à partir de symboles de propositions atomiques, appelées aussi *variables propositionnelles*.

L'ensemble des formules propositionnelles est alors défini comme le plus petit ensemble tel que :

- Vrai et Faux sont des formules propositionnelles ;
- si P est un symbole de proposition atomique alors P est une formule propositionnelle ;
- si P est une formule propositionnelle alors $\neg P$ est une formule propositionnelle ;
- si P et Q sont des formules propositionnelles alors $P \wedge Q$ et $P \vee Q$ sont des formules propositionnelles.

On admet qu'un symbole de proposition atomique est n'importe quelle chaîne de caractères. On définit donc le type Caml `ap` ("Atomic Proposition") suivant :

```
type ap = string;;
```

A partir de ce type `ap`, on définit un deuxième type Caml, le type `pf` des formules propositionnelles selon la définition précédente, de la façon suivante :

```
type pf = Vrai
        | Faux
        | Atome of ap
        | Neg of pf
        | Et of pf * pf
        | Ou of pf * pf
;;
```

Ensuite, on peut, par exemple, représenter la formule propositionnelle $f = P \wedge (\text{Faux} \vee (\text{Vrai} \wedge Q))$ par l'expression Caml :

```
Et(Atome("P"), Ou(Faux, Et(Vrai, Atome("Q"))))
```

1. Déclarez dans un fichier `formule2.ml`, les types `ap` et pf.
2. Définissez la formule propositionnelle $f = P \wedge (\text{Faux} \vee (\text{Vrai} \wedge Q))$, vous devez obtenir le résultat suivant :

   ```
   val f : pf = Et(Atome "P", Ou(Faux,Et(Vrai,Atome "Q")))
   ```

3. Ecrivez une fonction récursive `affichePF` qui affiche une formule de type `pf` de manière infixée, sans afficher les parenthèses les plus extérieures.

   On doit obtenir :

   À taper : `affichePF f;;`
   Réponse : `P ^ (Faux v (Vrai ^ Q))- : unit = ()`

4. Définir les formules propositionnelles suivantes :
   – $f1 = (\text{Vrai} \wedge \text{Faux}) \vee (\text{Vrai} \wedge (\text{Faux} \wedge ((\text{Vrai} \wedge \text{Vrai}) \wedge \text{Faux})))$
   – $f2 = (((\text{P} \wedge \text{Faux}) \wedge (\text{Faux} \vee \text{Vrai})) \wedge ((\text{P} \vee \text{Vrai}) \vee (\text{Vrai} \wedge \text{Q}))) \wedge \text{Vrai}$

   Vérifiez que vous obtenez :

   À taper : `affichePF f1;;`
   Réponse : `(Vrai ^ Faux) v (Vrai ^ (Faux ^ ((Vrai ^ Vrai) ^ Faux)))- : unit = ()`
   À taper : `affichePF f2;;`
   Réponse : `(((P ^ Faux) ^ (Faux v Vrai)) ^ ((P v Vrai) v (Vrai ^ Q))) ^ Vrai- : unit = ()`

3 Déclencher une exception

On considère la fonction `test` suivante :

```
let test = function
  1 -> 1
| 2 -> 2
```

Elle n'est définie que pour 1 et 2, ce que remarque l'interpréteur en donnant même un exemple pour lequel la fonction n'est pas définie : 0

À taper :
```
let test = function
  1 -> 1
| 2 -> 2
```
Réponse :
```
File "...", line ..., characters 11-37:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 0
val test : int -> int = <fun>
```

Quand on utilise cette fonction pour une valeur non définie on obtient un message d'erreur :

À taper :
```
test 4;;
```
Réponse :
```
Exception: Match_failure ("formule2.ml", 49, 11)
```

Il est possible de rendre ce message plus clair en indiquant lors de la déclaration de la fonction le message retourné si elle est mal utilisée :

À taper :
```
let test = function
  1 -> 1
| 2 -> 2
| _ -> failwith "mauvaise usage de la fonction test"
```
Réponse :
```
val test : int -> int = <fun>
```
À taper :
```
test 1;;
```
Réponse :
```
- : int = 1
```
À taper :
```
test 4;;
```
Réponse :
```
Exception: Failure "mauvaise usage de la fonction test".
```

On dit que l'on a déclenché une exception

4 Valuation

Pour interpréter les formules propositionnelles définies par les types Caml `ap` et `pf`, il est nécessaire de valuer les propositions atomiques utilisées dans cette formule.

1. Définir une fonction `valuation1 : ap -> bool` retournant respectivement les valeurs `true` et `false` pour les propositions atomiques `P` et `Q` en gérant à l'aide d'une exception le cas où la proposition atomique passée en paramètre n'est pas définie.
2. Définir une fonction `valuation2` retournant la valeur `true` pour les propositions atomiques `P` et `Q` en gérant à l'aide d'une exception le cas où la proposition atomique passée en paramètre n'est pas définie.

5 Fonction ayant en argument une fonction

Une fonction peut être l'argument d'une autre fonction. On peut ainsi définir la fonction `ffois2` qui prend en argument une valeur `x` et une fonction `f` et retourne `2*f(x)` :

À taper :
```
let ffois2 = function
(x,f) -> f(x)*2
```
Réponse :
```
val ffois2 : 'a * ('a -> int) -> int = <fun>
```
À taper :
```
ffois2 (4,(function x -> x));;
```
Réponse :
```
- : int = 8
```

On considère la fonction `ff` qui prend en argument une variable `x` et une fonction `f` et qui retourne `f(x)`.

```
À taper :   let ff = function
            (x,f) -> f(x)
Réponse :   val ff : 'a * ('a -> 'b) -> 'b = <fun>
À taper :   ff (2,(function x -> x));;
Réponse :   - : int = 2
À taper :   ff (4,(function x -> 2*x));;
Réponse :   - : int = 8
À taper :   ff (true,(function x -> true && false));;
Réponse :   - : bool = false
```

6 Interprétation de formules propositionnelles

1. Définissez une fonction `interpretation : pf * (ap -> bool) -> bool` permettant de réaliser l'interprétation booléenne d'une formule propositionnelle à partir d'une valuation donnée en paramètre.
2. Testez la fonction `interpretation` avec
   - la formule `f` et la valuation `valuation1`
   - la formule `f` et la valuation `valuation2`
   - la formule `f1` et la valuation `valuation1`
   - la formule `f1` et la valuation `valuation2`
   - la formule `f2` et la valuation `valuation1`
   - la formule `f2` et la valuation `valuation2`

   ```
   À taper :   interpretation (f,valuation1);;
   Réponse :   - : bool = false
   À taper :   # interpretation (f,valuation2);;
   Réponse :   - : bool = true
   À taper :   # interpretation (f1,valuation1);;
   Réponse :   - : bool = false
   À taper :   # interpretation (f2,valuation1);;
   Réponse :   - : bool = false
   À taper :   # interpretation (f1,valuation2);;
   Réponse :   - : bool = false
   À taper :   # interpretation (f2,valuation2);;
   Réponse :   - : bool = false
   ```

3. Définissez la formule propositionnelle $f3 = \text{Faux} \vee ((\text{R} \wedge \text{Vrai}) \wedge \text{P})$. Vérifiez que l'application de la fonction `interpretation` à la formule `f3` et à la valuation `valuation1` lève bien une exception.