

1 Logique

► **Question 1** Écrivez des fonctions *non*, *et*, *ou*, et *implique* qui prennent comme arguments deux booléens et effectuent les opérations booléennes éponymes. Vous n'avez le droit d'utiliser que **if ... then ... else ...** et les constantes **true** et **false**.

val non : bool → bool

val et : bool → bool → bool

val ou : bool → bool → bool

val implique : bool → bool → bool

On note \mathcal{B} l'ensemble des booléens : une fonction booléenne est une application $\mathcal{B}^n \rightarrow \mathcal{B}$, pour un certain n . Par exemple g est la fonction booléenne définie par $g(x, y) = x \vee y$. Par la suite, on nommera toujours les variables d'une fonction booléenne x_0, \dots, x_{n-1} , donc on pourra simplement noter la fonction g par $x_0 \vee x_1$. En Caml, on représentera simplement une variable x_i par l'entier i :

```
type variable = int;;
```

Si vous construisez une fonction booléenne avec les primitives précédentes, il est très facile de connaître sa valeur pour des x_0, \dots, x_{n-1} donnés. Cependant les problèmes intéressants sont plus compliqués que cela. Étant donnée une fonction booléenne g , on se demandera plutôt :

- Est-ce que $g(x_0, \dots, x_{n-1})$ s'évalue en "vrai" pour toutes valeurs de x_0, \dots, x_{n-1} ? On dit alors que g est une tautologie.
- Existe-t-il un n -uplet de booléens x_0, \dots, x_n tel que $g(x_0, \dots, x_{n-1})$ s'évalue en "vrai"? On dit alors que g est satisfiable.

► **Question 2** Donnez des exemples de tautologies et de fonctions non satisfiables.

Pouvez-vous donner des idées expliquant intuitivement pourquoi ces deux problèmes sont difficiles à résoudre en général? Connaissez-vous un exemple de forme particulière de fonction booléenne pour laquelle ils deviennent plus simples?

2 Arbres de décision

2.1 Définition

Nous allons utiliser une structure de données particulière, les *arbres de décision*, pour représenter les fonctions booléennes, en espérant que cette structure simplifie les calculs dans les exemples qui nous intéressent.

Étant donné un ensemble ordonné de n variables notées x_0, \dots, x_{n-1} , on définit les *arbres de décision* sur ces variables de la façon suivante :

1. les symboles "vrai" et "faux" sont des arbres de décision ;
2. $\text{Test}_{x_i}(v, f)$ est un arbre de décision, si v et f sont des arbres de décision tels que :
 - (a) les arbres v et f sont différents,
 - (b) toutes les variables de v et f sont des x_j avec $j > i$.

Dans un nœud interne $\text{Test}_{x_i}(v, f)$, x_i est la *variable testée*, v est le *fil positif* et f le *fil négatif*.

```
type abd =
  | Bool of bool
  | Test of variable * abd * abd
;;
```

Pour formaliser la manière dont un arbre de décision représente une fonction booléenne, on introduit une fonction *test* composant trois applications g, g', g'' de $\mathcal{B}^n \rightarrow \mathcal{B}$ de la manière suivante : $\text{test}(g, g', g'')$ est une fonction de $\mathcal{B}^n \rightarrow \mathcal{B}$ définie par

- $\text{test}(g, g', g'')(x_0, \dots, x_{n-1}) = g'(x_0, \dots, x_{n-1})$ si $g(x_0, \dots, x_{n-1})$ s'évalue en "vrai".
- $\text{test}(g, g', g'')(x_0, \dots, x_{n-1}) = g''(x_0, \dots, x_{n-1})$ si $g(x_0, \dots, x_{n-1})$ s'évalue en "faux".

Finalement :

- Les arbres "vrai" et "faux" représentent respectivement les applications constantes vraie et fausse.
- L'arbre $\text{Test}_{x_i}(v, f)$ représente l'application $\text{test}(x_i, g', g'')$, où les arbres v et f représentent respectivement les applications g' et g'' .

► **Question 3** On représentera dans cette question un n -uplet de valeurs booléennes par un tableau x de n cases au moins. La valeur x_i sera alors $x.(i)$. Écrivez une fonction *evaluation* qui prend en argument un arbre de décision a et un tableau x et qui évalue la fonction représentée par a en x .

val evaluation : abd → bool array → bool

2.2 Manipulations

► **Question 4** Une projection selon une variable x_i est une application qui à un n -uplet (x_0, \dots, x_{n-1}) associe la valeur de x_i . Écrivez une fonction *abd.proj* qui prend en argument une variable, et renvoie un arbre de décision représentant la projection selon cette variable.

val abd.proj : variable → abd

▷ **Question 5** Écrivez une fonction `abd_neg` qui prend en argument un arbre de décision représentant une fonction g , et renvoie un arbre de décision représentant $\neg g$. Trouvez un argument simple pour prouver que votre fonction est correcte.

`val abd_neg : abd → abd`

Si g est une fonction booléenne de $\mathcal{B}^n \rightarrow \mathcal{B}$, et b un booléen, on note $g_{i \leftarrow b}$ la fonction booléenne de $\mathcal{B}^n \rightarrow \mathcal{B}$ telle que :

$$g_{i \leftarrow b}(x_0, \dots, x_i, \dots, x_{n-1}) = g(x_0, \dots, b, \dots, x_{n-1})$$

▷ **Question 6** Écrivez une fonction `abd_partiel` qui prend en arguments une variable x_i , un booléen b et un arbre représentant g , et renvoie un arbre représentant $g_{i \leftarrow b}$.

`val abd_partiel : variable → bool → abd → abd`

2.3 Composition

▷ **Question 7** Si ce n'est déjà fait, sortez une feuille de papier et un stylo. Exprimez alors la fonction `test` à l'aide de la projection x_i , et des applications partielles $g_{i \leftarrow faux}$, $g_{i \leftarrow vrai}$, $g'_{i \leftarrow faux}$, $g'_{i \leftarrow vrai}$, $g''_{i \leftarrow faux}$, $g''_{i \leftarrow vrai}$.

▷ **Question 8** Dédisez une fonction `abd_test` qui prend en arguments les arbres de trois fonctions c , v et f , et qui renvoie l'arbre représentant la composition `test(c, v, f)`. Attention à bien respecter toutes les contraintes sur la forme des arbres de décision.

`val abd_test : abd → abd → abd → abd`

▷ **Question 9** Utilisez la fonction `abd_test` pour écrire trois fonctions `abd_et`, `abd_ou` et `abd_implique` réalisant les opérations $f \wedge g$, $f \vee g$ et $f \Rightarrow g$ sur des arbres de décision. N'hésitez pas à vous inspirer de vos réponses à la première question de ce TP.

`val et : abd → abd → abd`

`val ou : abd → abd → abd`

`val implique : abd → abd → abd`

3 Jouons

La direction d'une crèche très stricte décide de contrôler les jouets qu'apportent les enfants. Elle définit donc une politique d'admission des jouets, qui doivent se conformer à l'ensemble des six règles suivantes :

1. les jouets doivent être de petite taille, sauf les peluches ;
2. un jouet est soit vert, soit grand, soit grand et vert ;
3. les jouets électriques sont obligatoirement accompagnés de leurs piles ;
4. un enfant ne peut à la fois apporter des piles et une peluche ;
5. tous les jouets verts sont des peluches et sont électriques ;

6. toutes les peluches sont vertes.

Un jouet est caractérisé par cinq variables booléennes, à savoir `peluche`, `petit`, `vert`, `électrique` et `piles`. Ainsi, si Oscar se présente avec un train électrique rouge, muni de piles et que ce jouet est jugé grand, `peluche` vaut `false`, `petit` vaut `false`, `vert` vaut `false`, `électrique` vaut `true` et `piles` vaut `true`.

▷ **Question 10** Expliquez comment écrire la politique de la crèche comme une fonction booléenne. Oscar peut-il entrer ?

On ordonne les variables dans l'ordre suivant :

`peluche < petit < vert < électrique < piles`

▷ **Question 11** Donnez l'arbre correspondant à la politique de la crèche. Vous pouvez bien sûr le faire calculer par Caml. Reformulez en français la politique d'une façon plus simple que les six règles.

▷ **Question 12** Face aux protestations des parents, il est décidé d'annuler la sixième règle. Calculez le nouvel arbre et résumez en français la situation. Vous pouvez en profiter pour tester sur Oscar votre fonction `evaluation`.

Pour assouplir encore cette discipline, on propose une autre sixième règle :

6'. Indépendamment des autres règles, tous les jouets verts sont autorisés.

▷ **Question 13** Expliquez comment obtenir la fonction booléenne correspondant à cette nouvelle politique. Calculez le nouvel arbre, et amusez-vous à changer l'ordre des variables pour obtenir un arbre le plus petit possible.

4 Bonus

Quelques question subsidiaires plus difficiles pour ceux qui en veulent.

▷ **Question 14** Écrivez une fonction `abd_solutions` qui prend une politique p représentée par son arbre de décision en argument et compose l'affiche des jouets autorisés. Quel est le nombre de lignes des affiches produites par votre fonction pour les politiques précédemment évoquées, ainsi que pour la politique laxiste qui autorise tous les jouets ?

`val abd_solutions : abd → unit`

▷ **Question 15** Le type `abd` permet de représenter des arbres qui ne sont pas des arbres de décision : avez-vous un exemple ? Écrivez une fonction `abd_correct` qui teste si un arbre est bien un arbre de décision : on peut la programmer pour qu'elle n'effectue qu'un seul parcours de l'arbre.

`val abd_correct : abd → bool`

On peut commencer par écrire une fonction `abd_egal` qui teste si deux arbres sont égaux, et ce sans utiliser l'opérateur `=` sur les arbres, puis s'en inspirer.

► **Question 1**

```
let et x y = if x then y else x;;
let ou x y = if x then x else y;;
let implique x y = if x then y else true;;
```

► **Question 2**

- Deux exemples de tautologies : $x_0 \vee \neg x_0$ et $x_0 \Rightarrow x_0$.
- Un exemple de formule non satisfiable : $x_0 \wedge \neg x_0$.

Pour résoudre nos deux problèmes, on peut regarder la *table de vérité* de la fonction : si toutes les lignes valent vrai, la fonction est une tautologie (premier problème), ou si l'une des lignes est vraie, on a trouvé les x_0, \dots, x_{n-1} cherchés (second problème). Cependant, si la fonction attend n arguments, la table de vérité possède 2^n lignes. Et personne ne sait vraiment faire mieux qu'un algorithme exponentiel.

En revanche, si l'on arrive à écrire la fonction sous une forme normale disjonctive, on peut s'en sortir en temps polynomial.

► **Question 3**

```
let rec evaluation a x =
  match a with
  | Bool b      -> b
  | Test(i, v, f) -> if x.(i)
                      then evaluation v x
                      else evaluation f x
;;
```

► **Question 4**

```
let abd_proj x =
  Test(x, Bool true, Bool false)
;;
```

► **Question 5**

```
let rec abd_neg a =
  match a with
  | Bool b      -> Bool (not b)
  | Test(x, a1, a2) -> Test(x,
                              abd_neg a1,
                              abd_neg a2)
;;
```

Quant à la correction, il suffit de remarquer que la seule opération réalisée par la fonction est une transposition de **true** et **false**, donc si l'arbre en entrée est bien formé, l'arbre renvoyé l'est aussi.

► **Question 6**

```
let rec abd_partiel x b a =
  match a with
  | Test (y, a1, a2) when x = y ->
    if b then a1 else a2
  | Test (y, a1, a2) when x > y ->
    let a1' = abd_partiel x b a1
    and a2' = abd_partiel x b a2
    in
    if a1' = a2'
    then a1'
    else Test(y, a1', a2')
  | - as a -> a
;;
```

► **Question 7**

$$\text{test}(g, g', g'') = \text{test}(x_i, \text{test}(g_{i \leftarrow \text{vrai}}, g'_{i \leftarrow \text{vrai}}, g''_{i \leftarrow \text{vrai}}), \text{test}(g_{i \leftarrow \text{faux}}, g'_{i \leftarrow \text{faux}}, g''_{i \leftarrow \text{faux}}))$$

► **Question 8**

```
let rec abd_test c v f =
  match c with
  | Bool true -> v
  | Bool false -> f
  | Test(x, -, -) ->
    (* En prenant la plus petite parmi les
       racines de c, v et f, on s'assure que
       résultat sera bien formé. *)
    let racine_min x a =
      match a with
      | Test(y, -, -) -> min x y
      | - -> x
    in
    let xi = racine_min (racine_min x v) f in
    let xip = abd_partiel xi
    in
    let a1 = abd_test (xip true c)
                (xip true v)
                (xip true f)
    and a2 = abd_test (xip false c)
```

```

                (xip false v)
                (xip false f)
    in
        if a1 = a2
        then a1
        else Test(xi, a1, a2)
;;

```

► **Question 9**

```

let abd_et x y = abd_test x y x;;
let abd_ou x y = abd_test x x y;;
let abd_implique x y =
    abd_test x y (Bool true);;

```

► **Question 10** La fonction booléenne est la conjonction des six formules suivantes :

1. $\neg \text{peluche} \Rightarrow \text{petit}$
2. $\text{vert} \vee \text{grand}$
3. $\text{électrique} \Rightarrow \text{pile}$
4. $\neg(\text{piles} \wedge \text{peluche})$
5. $\text{vert} \Rightarrow (\text{peluche} \wedge \text{électrique})$
6. $\text{peluche} \Rightarrow \text{vert}$

Le pauvre petit Oscar devra passer la journée dehors, renoncer à son jouet ou se trouver une nouvelle crèche.

► **Question 11** Avec cette politique, la crèche n'accepte aucun jouet ! Le calcul avec *Caml* peut s'effectuer ainsi :

```

let peluche = abd_proj 0
and petit = abd_proj 1
and vert = abd_proj 2
and électrique = abd_proj 3
and piles = abd_proj 4;;

let p =
    let regles = [abd_implique (abd_neg peluche) petit;
                abd_ou vert (abd_neg petit);
                abd_implique électrique piles;
                abd_neg (abd_et piles peluche);
                abd_implique vert (abd_et peluche électrique);
                abd_implique peluche vert]
    in
        List.fold_left abd_et (Bool true) regles;;

```

► **Question 12** La crèche accepte cette fois les grandes peluches sans piles, qui ne sont ni vertes ni électriques. Oscar ne peut toujours pas entrer.

► **Question 13** On obtient la nouvelle fonction en prenant la disjonction de `vert` et de la formule précédente.

Avec cette nouvelle politique, les arbres sont globalement plus petits (nombre de nœuds) quand la variable `vert` a un petit indice.

► Question 14

```
let nom = [| "peluche"; "petit"; "vert"; "electrique"; "piles" |];;

let rec affiche_chaine =
  (* Affichage d'une caractéristique d'un jouet,
   niée ou non. *)
  let affiche_criterie (carac, neg) =
    if neg then print_string "non_"; print_string nom.(carac)

  in function
    | [] -> print_string "Tous_les_jouets_sont_acceptés\n"
    | [c] -> affiche_criterie c; print_newline()
    | c::r ->
      affiche_criterie c; print_string ",_"; affiche_chaine r;;

(* Une solution, qui parcourt l'arbre et affiche
   tous les chemins menant à une feuille "Bool true". *)
let abd_solutions politique =
  (*
   La fonction abd_affiche est de type
   (int * bool) list -> abd -> int.
   Le premier paramètre stocke les variables rencontrées
   sur les noeuds Test(x, -, _) : on ajoute (x, false)
   lorsque l'on explore le fils de gauche (ce qui
   signifie que x n'est pas nié), alors que l'on ajoute
   (x, true) pour explorer le fils de droite
   (c'est-à-dire quand x est nié).

   La valeur renvoyée correspond au nombre de règles
   qui ont été affichées, ce qui permet de distinguer le
   cas où tous les jouets sont refusés.
  *)
  let rec abd_affiche chaine = function
    Bool true -> affiche_chaine chaine; 1
  | Bool false -> 0
  | Test(x, ag, ad) ->
    abd_affiche ((x, false)::chaine) ag
    + abd_affiche ((x, true)::chaine) ad

  in if abd_affiche [] politique = 0
    then print_string "Aucun_jouet_n'est_accepté\n";;

abd_solutions p;;
abd_solutions (Bool true);;
```

► Question 15

```
let rec abd_egal a a' = match (a, a') with
| (Bool b, Bool b') -> b = b'
| (Test (x, ag, ad), Test (x', ag', ad'))
  when x = x' -> (abd_egal ag ag') && (abd_egal ad ad')
| _ -> false;;
```

La fonction auxiliaire `egal_top` suit le même schéma que `abd_egal`, mais elle renvoie plus d'informations : elle retourne un triplet composé de :

1. l'indice de la plus petite variable apparaissant dans les sous-arbres, -1 si elle est indéfinie ;
2. un booléen indiquant si à un moment, l'ordre sur les variables n'a pas été respecté ;
3. un booléen indiquant si les sous-arbres sont égaux.

```
let abd_correct =
  let rec echec_top = function
  | (Bool b, Bool b') -> (-1, false, b = b')
  | (Test (x, ag, ad), Bool _) -> (x, false, false)
  | (Bool _, Test(x, ag, ad)) -> (x, false, false)
  | (Test (x, ag, ad), Test (x', ag', ad')) ->
    let (top1, ordre, egal) = echec_top (ag, ag')
    and (top2, ordre', egal') = echec_top (ad, ad')
    in let top =
      if top1 <= 0 || top2 <= 0
      then max top1 top2
      else min top1 top2
      in ((if top >= 0 then min top x else x),
        ordre || ordre' || (x > top && top <> -1),
        x = x' && egal && egal')
  in function
  | Bool _ -> true
  | Test (x, ag, ad) ->
    let (top, ordre, egal) = echec_top (ag, ad)
    in (top = -1 || top > x) && (not ordre) && (not egal);;

abd_correct (Test (0, (Test (1, Bool true, Bool false)), Bool false));;
abd_correct (Test (1, (Test (0, Bool true, Bool false)), Bool false));;
abd_correct (Test (0, (Test (1, Bool true, Bool false)),
  (Test (1, Bool false, Bool false))));;
abd_correct (Test (0, (Test (1, Bool true, Bool false)),
  (Test (1, Bool true, Bool false))));;
```