# Outils logiques et algorithmiques – TP 2 – Arbres de syntaxe

**Exercise 1** (Formules logiques) On reprend le type caml vu en cours pour l'arbre de syntaxe des formules propositionnelles.

```
type binop = And | Or | Imp
type fmla =
  | True
  | False
  | Var of int
  | Not of fmla
  | Bin of binop * fmla * fmla
```

1. Écrire une fonction `contains: int -> fmla -> bool` qui prend en paramètres un numéro $i$ et une formule $f$, et qui renvoie `true` si la variable $X_i$ apparaît dans $f$.
2. Rappel de maths pour l'info : l'implication $f_1 \Longrightarrow f_2$ est équivalente à la disjonction $\neg f_1 \vee f_2$. En déduire une fonction `elim_imp: fmla -> fmla` qui prend en paramètre une formule $f$ et qui renvoie une formule $f'$ équivalente n'utilisant pas le connecteur $\Longrightarrow$.

Une formule en *forme normale négative* est une formule utilisant uniquement les connecteurs $\wedge$, $\vee$ et $\neg$, et dans laquelle le connecteur $\neg$ n'apparaît que directement appliqué à une variable. Nous allons voir que toutes formule $f$ peut être transformée en une formule équivalente $f'$ en forme normale négative.

3. Écrire une fonction `is_nnf: fmla -> bool` qui prend en paramètre une formule $f$, et qui renvoie `true` si et seulement si $f$ est en forme normale négative.
4. Écrire une fonction `neg_nnf: fmla -> fmla` qui prend en paramètre une formule $f$ supposée en forme normale négative, et qui renvoie une formule $f'$ équivalente à $\neg f$, en forme normale négative.
   *Rappel de maths pour l'info : on a les identités suivantes*

$$\neg(\neg f) \equiv f \qquad \neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2 \qquad \neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2$$

5. En déduire une fonction `nnf` qui prend en paramètre une formule $f$ arbitraire, et renvoie une formule $f'$ équivalente en forme normale négative.
6. Quelle est la complexité de la fonction `nnf` obtenue à la question précédente ?
7. *Bonus :* écrire deux fonctions mutuellement récursives `nnf: fmla -> fmla` et `neg_nnf: fmla -> fmla` telles que `nnf f` renvoie la forme normale négative de `f`, et `neg_nnf f` renvoie la forme normale négative de la négation de `f`, toutes deux *en temps linéaire en la taille de* `f`.

□

**Exercise 2** (Smart constructors) Observez cette fonction.

```
let s_not f = match f with
  | True   -> False
  | False  -> True
  | Not f' -> f'
  | _      -> Not f
```

Appliquée à une formule `f`, elle renvoie dans tous les cas une formule équivalente à `Not f`, mais en évitant de produire certains schémas trivialement simplifiables comme $\neg\top$, $\neg\bot$ ou $\neg\neg f'$. On pourrait imaginer avoir également des fonctions `s_and`, `s_or` et `s_imp` appliquant la même philosophie aux autres connecteurs (de telles fonctions de construction sont parfois appelées des *smart constructors*). Alors, plutôt que de définir la formule $\neg X_0 \Longrightarrow (X_1 \wedge \bot)$ par l'expression

```
let f = Bin(Imp, Not(Var 0), Bin(And, Var 1, False))
```

on pourrait à la place appeler les fonctions associées aux différents connecteurs.

```
let f' = s_imp (s_not (Var 0)) (s_and (Var 1) False)
```

La formule est alors simplifiée dès sa construction, et ici la formule `f'` serait directement `Var 0`.

1. Définir une fonction `s_and: fmla -> fmla -> fmla` telle que `s_and f1 f2` renvoie une formule équivalente à `Bin(And, f1, f2)` en évitant de produire des combinaisons trivialement simplifiables.
2. Définir de même des fonctions `s_or` et `s_imp` appliquant le même principe aux connecteurs de disjonction et d'implication.
3. Définir également des fonctions `s_iff` et `s_xor` pour les connecteurs logiques d'équivalence ($\Longleftrightarrow$) et de « ou exclusif ». *Vous pouvez utiliser les fonctions précédentes.*
4. Les *smart constructors* donnent aussi un moyen de simplifier une formule qui aurait déjà été construite : il suffit de la « reconstruire à l'identique », mais en utilisant les fonctions de construction plutôt que les constructeurs eux-mêmes. Écrire une fonction `simplify` utilisant ce principe.

□

**Exercise 3** (Expressions arithmétiques) On s'intéresse à des expression numériques construites avec les éléments suivants :

— des constantes entières,

— des variables numérotées $X_0$, $X_1$, $X_2$...

— des expressions construites avec les opérateurs `+`, `-` et `*`

Note : l'opérateur `-` peut apparaître aussi bien comme opérateur binaire (exemple : $X_0 - 1$) que comme opérateur unaire (exemple : $-(2 * X_1)$).

1. Définir un type caml `expr` pour représenter de l'arbre de syntaxe de telles expressions.
2. Définir une fonction `constant: expr -> bool` qui prend en paramètre une expression $e$ et qui renvoie `true` si et seulement si l'expression ne contient aucune variable.
3. Définir une fonction `eval: expr -> int array -> int` qui prend en paramètres une expression $e$ et un tableau $v$ donnant des valeurs pour les variables de $e$ (`v[i]` donne la valeur de $X_i$), et qui calcule la valeur de l'expression.

□

**Exercise 4** (Expression régulières) Rappel. Une expression régulière $e$ sur un alphabet $A$ est la description d'un ensemble de mots construits avec les lettres de $A$. Une *expression régulière* $e$ est :

— soit le mot vide $\varepsilon$,

— soit un caractère $a \in A$,

— soit l'alternative $e_1 \mid e_2$ entre deux expressions régulières $e_1$ et $e_2$,

— soit la concaténation $e_1 e_2$ de deux expressions régulières $e_1$ et $e_2$,

— soit l'étoile $e^*$ d'une expression régulière $e$.

Il s'agit à nouveau d'une définition intrinsèquement récursive que l'on peut représenter par un arbre de syntaxe.

```
type regexp =
  | Epsilon
  | Char   of char
  | Alt    of regexp * regexp
  | Concat of regexp * regexp
  | Star   of regexp
```

1. Écrire une fonction `epsilon: regexp -> bool` telle que `epsilon e` renvoie `true` si et seulement si le mot vide appartient à l'ensemble de mots représenté par `e`.
2. Écrire une fonction `first: char -> regexp -> bool` telle que `first a e` renvoie `true` si et seulement si au moins l'un des mots de l'ensemble représenté par `e` commence par la lettre `a`. Écrire de même une fonction `last`, détectant si l'un des mots représentés par `e` termine par la lettre `a`.
3. Écrire une fonction `follow: char -> char -> bool` telle que `follow a b e` renvoie `true` si et seulement si les lettres `a` et `b` apparaissent consécutivement dans au moins l'un des mots représentés par `e`.

La *dérivée* d'une expression régulière $e$ par rapport à un caractère $a$, que l'on notera ici $e/a$, est l'expression régulière décrivant l'ensemble des mots $m$ tels que $am$ est reconnu par $e$ (autrement dit : les possibilités de former un mot de $e$ une fois qu'on a déjà lu $a$). Par exemple : (`"abc" | "ad" | "efg" | "a"`)/`"a" = "bc" | "d"` | $\varepsilon$. Pour tenir compte des situations où il n'est plus possible de former un mot valide, on ajoute une nouvelle forme d'expression régulière : $\emptyset$, représentant l'ensemble vide (à ne pas confondre avec $\varepsilon$, qui désigne l'ensemble contenant exactement un mot : le mot vide). On donne les règles de calcul suivantes :

$$
\begin{aligned}
\varepsilon/a &= \emptyset \\
b/a &= \begin{cases} \varepsilon & \text{si } b = a \\ \emptyset & \text{sinon} \end{cases} \\
(e_1 \mid e_2)/a &= (e_1/a) \mid (e_2/a) \\
(e_1 e_2)/a &= \begin{cases} (e_1/a)e_2 & \text{si } e_1 \text{ ne reconnaît pas } \varepsilon \\ (e_1/a)e_2 \mid e_2/a & \text{si } e_1 \text{ reconnaît } \varepsilon \end{cases} \\
e^*/a &= (e/a)e^* \\
\emptyset/a &= \emptyset
\end{aligned}
$$

Pour tester si un mot $m$ est reconnu par une expression $e$, il suffit alors de calculer les dérivées successives de $e$ pour les lettres de $m$, et de tester si l'expression régulière obtenue à la fin reconnaît $\varepsilon$.

4. Ajouter un constructeur `Empty` au type `regexp` pour représenter l'expression régulière $\emptyset$.
5. Écrire une fonction `derive: regexp -> char -> regexp` telle que `derive e a` renvoie l'expression dérivée `e/a`.
6. En déduire une fonction `accept: regexp -> char list -> bool` telle que `accept e m` renvoie `true` si et seulement si l'expression régulière `e` reconnaît la séquence de lettres `m`.

□