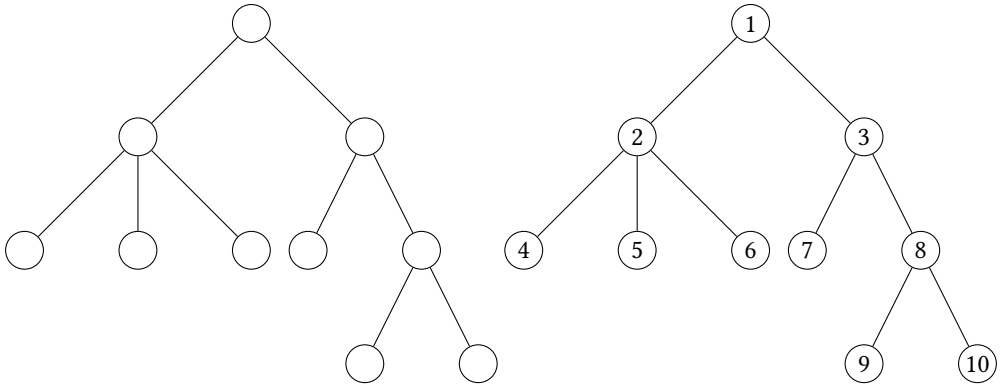
Arbres

Table des matières

I) Arbres

Un arbre enraciné est une structure hiérarchique avec un élément distingué, appelée racine, et où chaque élément a un certain nombre de «fils». On peut considérer différentes variations de cette notion, selon si les éléments de l'arbre sont étiquettés ou non, et selon le nombre de «fils» que l'on autorise (exactement 2, au plus 2, un nombre quelconque).



Terminologie

- Les éléments de l'arbre sont appelés des **nœuds**.
- Un nœud de l'arbre à des **enfants** (ou **fils**).
- Les nœuds de l'arbre sans fils sont appelés des **feuilles**.
- Les nœuds qui ne sont pas des feuilles sont des **nœuds internes**.
- Mis à part la racine, tout élément de l'arbre a un **parent**.
- Le nombre d'enfants d'un nœud est **l'arité** de ce nœud.
- Les **descendants** d'un nœud e sont les nœuds qui sont hiérarchiquement sous le nœud e.

Remarque En théorie des graphes, un arbre est un graphe connexe acyclique. Un arbre enraciné est alors un arbre dans lequel on a choisi un sommet particulier, qui devient la racine.

1) Arbres binaires stricts (ou entiers) étiquettés

Ce sont des arbres où chaque nœud interne a exactement deux fils. Autrement dit, chaque nœud est soit une feuille (sans fils) soit admet un fils gauche et un fils droit.

On définit en `OCaml` le type récursif suivant pour représenter des arbres binaires stricts étiquettés, avec des étiquettes de type `'a`. Les fonctions prenant un tel arbre en argument s'écrivent typiquement avec le modèle par filtrage illustré à droite.

```
type 'a arbre =  Feuille of 'a
               | Noeud of 'a * 'a arbre * 'a arbre
```

```
let rec f = function
  | Feuille x -> ...
  | Noeud(x, g, d) -> ...
```

Remarque Par convention, les feuilles de l'arbre sont également des nœuds. Le constructeur `Noeud` serait mieux nommé `NoeudInterne`.

On peut définir un arbre manuellement, en utilisant les constructeurs :

```
let arb = Noeud (5,
                 Feuille 3,
                 Noeud (4,
                        Feuille 1,
                        Feuille 6))
```

Fonctions élémentaires

Définition

- La **profondeur** d'un nœud est «l'étage» auquel il se trouve, ou sa distance à la racine.
  La racine a une profondeur de 0.
- La **hauteur** d'un arbre est la profondeur maximale d'un nœud de l'arbre.
  Un arbre réduit à une feuille a une hauteur de 0.
- La **taille** d'un arbre est le nombre de nœuds de l'arbre (en comptant les feuilles).

Exercice 1 Écrire des fonctions `taille : 'a arbre -> int` et `hauteur : 'a arbre -> int`.

Exercice 2

1. Écrire une fonction `rech_etiquette : 'a arbre -> 'a -> bool` qui prend en argument un arbre et une étiquette et cherche si un des nœuds de l'arbre a cette étiquette.
2. Écrire une fonction `profondeur_etiquette : 'a arbre -> 'a -> int` qui prend en argument un arbre et une étiquette étant présente au moins une fois dans l'arbre, et renvoie la profondeur d'un nœud ayant cette étiquette.

Exercice 3 Écrire une fonction `val etiquettes : 'a arbre -> 'a list` qui prend en argument un arbre, et renvoie la liste de ses étiquettes.

Construction d'un arbre

Exercice 4 Écrire une fonction `miroir` qui prend en argument un arbre, et renvoie un nouvel arbre avec la même structure, mais où les deux enfants de chaque nœuds ont été échangés.

Exercice 5 Écrire une fonction `numerote : 'a arbre -> int arbre` qui prend en argument un `'a arbre`, et renvoie un `int arbre` ayant la même structure que l'arbre en argument, où les nœuds sont numérotés, de sorte que la racine ait le numéro 1, et que si un nœud a le numéro k, son fils gauche ait le numéro $2k$, et son fils droit le numéro $2k+1$.

Exercice 6

1. Écrire une fonction `affecte_descendants` qui étant donné un arbre `a`, renvoie un arbre de même structure, tel que chaque sommet soit étiqueté par la somme des étiquettes de ses descendants (dont lui même) dans `a`.
2. Écrire une fonction `affecte_ancetres` qui étant donné un arbre étiqueté `a`, renvoie un arbre étiqueté `b` de même structure de sorte que l'étiquette de la racine ait été préservée et que chaque sommet soit étiqueté par la somme des étiquettes de ses ancêtres (dont lui-même) dans `a`.

Aspects théoriques

Définition mathématique

D'un point de vue mathématique, on peut définir un arbre binaire strict non vide comme un ensemble fini non vide A muni d'une relation binaire $\preceq$ (la relation de parenté) telle que

- il existe un élément $r \in A$ tel que $\forall x \in A,\ r \not\preceq x$.
- pour tout $x \in A$ différent de r, il existe un unique $y \in A,\ x \preceq y$.
- pour tout $x \in A$ différent de r, il existe une suite d'éléments deux à deux en relation, de l'élément x à la racine r :

$$x_0 = x \preceq x_1 \preceq \cdots \preceq x_n = r$$

Définition inductive

En pratique il est plus utile d'admettre une définition *inductive* des arbres, correspondant à la définition du type en `OCaml` : un arbre est soit une feuille, soit un nœud avec deux sous-arbres.

```
type 'a arbre =  Feuille of 'a | Noeud of 'a * 'a arbre * 'a arbre
```

Plus précisément, les arbres sont tous les objets que l'on peut construire à partir des feuilles en appliquant un nombre fini de fois la construction $(\mathcal{A}_1, \mathcal{A}_2) \to \text{Noeud}(\mathcal{A}_1, \mathcal{A}_2)$.
Cette définition donne lieu à une méthode de démonstration, dite par induction structurelle.

Propriété – Principe d'induction structurelle. Soit $\mathcal{P}(A)$ une assertion dépendant d'un arbre A. Pour montrer que $\mathcal{P}$ est valable pour tous les arbres, on montre que

- elle est valable pour les feuilles.
- si elle est valable pour deux arbres A_1 et A_2, elle est valable pour l'arbre formé d'un nœud dont les fils sont A_1 et A_2.

Remarque On peut voir le principe de récurrence comme un cas particulier du principe d'induction structurelle :

- l'ensemble $\mathbb{N}$ peut être construit par induction, un entier étant soit l'entier 0, soit le successeur d'un entier n.
- une preuve par induction d'une assertion $\mathcal{P}(n)$ consiste à vérifier que la propriété est vraie pour $n = 0$, et que si elle est vraie pour n, elle est vraie par son successeur.

Un exemple

Proposition Un arbre binaire strict avec n_i nœuds internes a $n_f = n_i + 1$ feuilles.

Démonstration.

□

2) Arbres binaires généraux

Ce sont des arbres dont chaque nœud a au plus deux enfants.
On représente un arbre binaire général (étiqueté) par le type suivant.

```
type 'a arbre =  Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Remarque Quelle est la représentation d'une feuille ?

Définition Par convention, la taille de l'arbre vide vaut 0 et sa hauteur vaut -1.

Exercice 7 Écrire une fonction `nb_feuilles` qui compte le nombre de feuilles d'un arbre binaire.

Exercice 8 Définir une fonction OCaml qui prend en arguments un entier p et un arbre binaire et qui renvoie la liste des sous-arbres non vides dont la racine est à la profondeur p dans A.

Exercice 9 Conversion entre les types Dans cet exercice, on suppose défini un type `type 'a arbre_strict = F of 'a | N of 'a * 'a arbre_strict * 'a arbre_strict`.

1. Écrire une fonction `est_strict` qui prend en argument un `'a arbre` et renvoie `true` s'il est strict, ou vide.
2. Écrire une fonction `strict_to_gen` qui prend en argument un `'a arbre_strict` et renvoie un `'a arbre` représentant le même arbre.
3. Écrire une fonction `gen_to_strict` qui prend en argument un `'a arbre` qui soit non vide et strict, et qui renvoie un `'a arbre_strict`. On renverra une erreur avec `failwith` si les conditions ne sont pas vérifiées.

Lien entre la taille et la hauteur

Proposition La taille n et la hauteur h d'un arbre binaire $\mathcal{A}$ vérifient

$$h + 1 \leq n \leq 2^{h+1} - 1.$$

Démonstration.

□

Corollaire On a $\log_2(n+1) - 1 \leq h \leq n-1$.

Remarque Pour un arbre général dont l'arité de chaque nœud est inférieur à $a > 1$, on a $h + 1 \leq n \leq \dfrac{a^{h+1}-1}{a-1}$.

a) Arbres complets

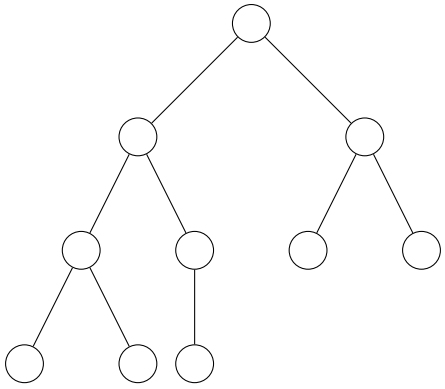
Définition Un arbre binaire est dit **complet** s'il est strict et si toutes ses feuilles ont la même hauteur.

Exercice 10
1. La fonction ci-dessous est-elle correct ?
2. Si A est un arbre complet, quel est, en fonction de la taille $|A|$ de A, la complexité de cette implémentation ?
3. Peut-on faire mieux ?

```
let rec est_complet = function
    Vide -> true
  | Noeud(a, g, d) -> est_complet g && est_complet d && hauteur g = hauteur d
```

Exercice 11 Un arbre binaire de hauteur h est dit presque complet à gauche s'il admet 2^{h-1} nœuds à la profondeur $h-1$ (le maximum possible), et si le niveau à la profondeur h est rempli éventuellement partiellement, mais nécessairement de gauche à droite.

1. Écrire une fonction `est_pcg`.
2. Montrer que la hauteur d'un tel arbre à n nœuds vérifie $h \leq \ln_2(n)$.



3) Arbres quelconques

On représente un arbre dont les nœuds ont une arité quelconque par le type suivant.

```
type 'a arbre = N of 'a * 'a arbre list
```

On rappelle les signatures des fonctions

- `List.map : ('a -> 'b) -> 'a list -> 'b list` qui prend en argument une fonction `f: 'a -> 'b` et une `'a list` et renvoie une `'b list` obtenue en appliquant `f` à chaque élément de la liste.
- `List.iter : ('a -> unit) -> 'a list -> unit` qui prend en argument une fonction `f: 'a -> unit` et une `'a list` et applique la fonction à chaque élément de la liste.
- `List.fold : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc` qui prend en argument une fonction `f: 'acc -> 'a -> 'acc`, une valeur initiale de l'accumulateur x_0, et une liste `[`$a_1; a_2; \dots, a_n$`]` et renvoie $f(\dots f(f(f(x_0, a_1), a_2), a_3)\dots)$.
- `List.exists : ('a -> bool) -> 'a list -> bool` qui prend en argument une fonction `f: 'a -> bool` et une `'a list` et renvoie `true` si et seulement si il existe un élément `e` de la liste pour lequel `f e` renvoie `true`.
- `List.concat : 'a list list -> 'a list` qui prend en argument une liste de liste, et concatène chaque sous-liste en une seule grande liste.

Exercice 12 En utilisant éventuellement les fonctions précédentes. Implémenter des fonctions

1. `taille` et `hauteur`.
2. `recherche` qui prend en argument un élément `e` et un arbre et renvoie `true` si et seulement si l'élément est l'étiquette d'un nœud de l'arbre.
3. `etiquettes` qui prend en argument un arbre et renvoie la liste des étiquettes de ses nœuds.
4. `descendant_gauche : 'a arbre -> 'a` qui renvoie l'étiquette la plus "à gauche" de l'arbre (on s'arrête quand un nœud n'a plus d'enfants). Idem pour `descendant_droite`.

Exercice 13 Écrire une fonction `to_binary : 'a arbre -> 'a barbre` qui prend en argument un arbre général $\mathcal{A}_g$ et renvoie un arbre binaire général, avec les mêmes sommets que l'arbre initial, dans lequel chaque sommet ait comme fils gauche son fils aîné dans $\mathcal{A}_g$, et comme fils droit son frère suivant dans $\mathcal{A}_g$.

```
type 'a barbre = Vide | Noeud of 'a * 'a barbre * 'a barbre
```

Exercice 14 Écrire une fonction `reracine` qui prend en argument un arbre et une étiquette présente dans l'arbre, et qui renvoie un nouvel arbre, obtenu en prenant pour nouvelle racine le nœud de l'étiquette donnée.

II) Arbres binaires de recherche

Il s'agit d'arbres binaires à étiquettes entières tel que pour chaque nœud n, les étiquettes des descendants de gauche de n aient des étiquettes strictement inférieures à celle de n, et les étiquettes des descendants de droite de n y soit strictement supérieures.

Propriété Les étiquettes d'un ABR sont distinctes.

Démonstration.

□

On utilise le type suivant.

```
type arbre = Vide | Noeud of int * arbre * arbre
```

1) Recherche et insertion

Exercice 15 Écrire une fonction `recherche` qui prend en argument un élément `x` et un arbre binaire de recherche et renvoie `true` si et seulement si `x` est l'étiquette d'un nœud de l'arbre.

Exercice 16 Écrire une fonction `insere` qui prend en argument un élément `x` et un arbre binaire de recherche ne contenant pas l'étiquette `x`, et renvoie un nouvel arbre binaire de recherche, qui contient également l'étiquette `x`, ajouté comme feuille de l'arbre.

2) Suppression d'un élément

La suppression d'un élément se ramène à la suppression de la racine (d'un sous-arbre).

Pour supprimer la racine, on la remplace par le plus grand élément du sous-arbre gauche (s'il est non vide), c'est-à-dire l'élément du sous-arbre gauche situé sur la branche la plus à droite ; en particulier, cet élément n'admet pas de fils droit. Et on met à son ancienne place la racine de son sous-arbre gauche, si celui-ci n'est pas vide.

Pour mettre en œuvre cet algorithme, le plus simple est d'écrire d'abord une fonction qui étant donné un ABR non vide renvoie le couple formé de l'ABR obtenu en supprimant le plus grand élément et du plus grand élément.

Exercice 17 Écrire de telles fonctions `supprime_plus_grand` et `supprime_racine`.

3) Représentation d'un ensemble par un ABR

On peut imaginer représenter un ensemble d'entier, auquel on souhaite pouvoir ajouter/retirer des éléments, par un ABR.

Les fonctions précédentes ont des complexité linéaire en la hauteur h de l'arbre. Si on suppose que l'arbre est suffisamment équilibré, on peut espérer que la taille $n = |A|$ de l'arbre soit de l'ordre de 2^h, autrement dit que $h = O(\ln n)$.

En pratique, il existe des variantes des ABR, comme les arbres rouges et noirs ou les arbres AVL, qui permettent de garantir que l'arbre reste relativement équilibré au fil des insertions/délétions. Une telle structure de donnée permet de représenter un ensemble d'entiers, auquel on peut ajouter/supprimer des éléments, avec des opérations élémentaires de complexité en $O(\ln n)$.

4) Vérification de la structure

Il s'agit d'écrire une fonction `validation` qui prend un arbre en argument et renvoie `true` si et seulement si c'est bien un arbre binaire de recherche.

Exercice 18 Écrire une telle fonction. †

III) Parcours d'un arbre

On travaille avec des arbres binaires généraux :

```
type 'a arbre =  Vide | Noeud of 'a * 'a arbre * 'a arbre
```

1) Parcours en profondeur

On veut parcourir un arbre binaire en profondeur, et appeler une fonction `traite` sur chaque étiquette.

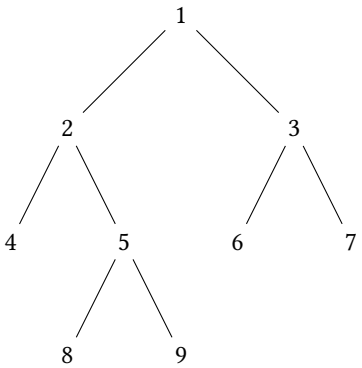
On distingue trois types de parcours, selon si on traite un nœud avant de traiter ses enfants (parcours préfixe), entre le traitement de ses deux enfants (parcours infixe), ou après le traitement de ses deux enfants (parcours suffixe).

```
(* Parcours préfixe *)
let rec parcours traite = function
    Vide -> ()
  | Noeud(a,g,d) ->
     traite a;
     parcours traite g;
     parcours traite d
```

```
(* Parcours infixe *)
let rec parcours traite = function
    Vide -> ()
  | Noeud(a,g,d) ->
     parcours traite g;
     traite a;
     parcours traite d;
```

```
(* Parcours suffixe *)
let rec parcours traite = function
    Vide -> ()
  | Noeud(a,g,d) ->
     parcours traite g;
     parcours traite d;
     traite a;
```

Exercice 19 Pour l'arbre ci-contre, décrire ce qu'imprime l'appel `parcours print_int arbre`, selon si le parcours est préfixe, infixe, ou suffixe.



Exercice 20 Renvoyer la liste des étiquettes dans un parcours en profondeur, selon l'ordre préfixe.

1. En utilisant la concaténation de listes.
2. sans utiliser la concaténation de listes. ‡

Chemins

Exercice 21 Écrire une fonction `chemin` qui prend un arbre `a` et un nœud `x` et qui renvoie la liste des nœuds de `a` parcourus de la racine jusqu'au nœud `x`. On suppose qu'il existe bien un tel chemin.

1. Pour un arbre binaire. §
2. Pour un arbre d'arité quelconque `type 'a arbre = N of 'a * 'a arbre list`.

Pour écrire des versions fonctionnelles, on pourra utiliser, au choix

- la fonction `List.find : ('a -> bool) -> 'a list -> 'a` qui lève une exception si elle ne trouve rien, à rattraper avec la syntaxe, `try expr with _ -> res`, qui évalue l'expression `expr` et renvoie `res` à la place si l'évaluation a levé une exception.
- la fonction `List.find_opt : ('a -> bool) -> 'a list -> 'a option`, le type `'a option` étant défini comme `type 'a option = None | Some of 'a`.

Reconstruction de l'arbre à partir d'un parcours

Exercice 22 On considère dans cet exercice des arbres binaires stricts, d'étiquettes distinctes. À tout parcours d'un arbre, on associe la liste des étiquettes dans l'ordre du parcours.

1. Dessiner deux arbres distincts dont les parcours préfixes donnent la même liste.
2. Écrire une fonction `reconstruire` qui prend en argument une liste de type `int * bool`, les entiers étant les étiquettes dans un parcours préfixes, et les booléens spécifiant si c'était l'étiquette d'une feuille (`true`) ou d'un nœud interne, et qui renvoie l'arbre.

   Si possible, on veut une complexité linéaire.

Exercice 23 ★ Un arbre binaire est uniquement déterminé par la donnée des deux listes des étiquettes de ses nœuds dans un parcours préfixe, et dans un parcours infixe. Écrire une fonction `reconstruire : int list -> int list -> int arbre` qui prend en argument ces deux listes et renvoie un arbre binaire général :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

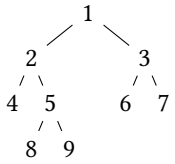
On supposera que les étiquettes sont des entiers positifs.

†. Indication : utiliser une fonction auxiliaire `max_min` qui étant donné un arbre non vide `a` renvoie un triplet `(v, mi, ma)`, où `v` vaut `true` si `a` est un ABR et `false` sinon, et où `mi` et `ma` sont respectivement la plus petite et la plus grande étiquette de l'arbre `a`. ‡. Utiliser une fonction auxiliaire récursive avec un accumulateur.

§. On pourra utiliser une fonction auxiliaire qui renvoie un couple `(bool, 'a list)` où le booléen décide si on a trouvé un chemin. Ou on peut simplement convenir de renvoyer la liste vide si on n'a pas trouvé de chemin.

2) Parcours en largeur

Un parcours en largeur consiste à visiter les nœuds dans l'ordre de leur profondeur (et typiquement, de gauche à droite) : dans l'arbre ci-contre, les sommets sont numérotés selon un parcours en largeur.




```
type 'a arbre =  Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Exercice 24 Écrire une fonction `sommets_parcours_largeur` qui renvoie la liste des sommets d'un arbre, selon l'ordre d'un parcours en largeur. ¶

IV) Structures arborescentes diverses

1) Formules arithmétiques, formules booléennes

a) Représentation d'expressions arithmétiques

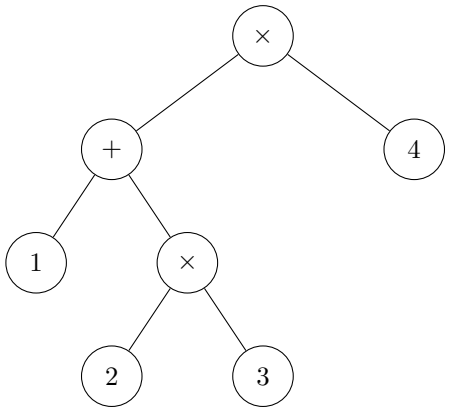
On peut représenter une expression arithmétique sur des entiers par un type arborescent dont les feuilles sont entières, et les nœuds sont étiquetés par des opérateurs.

L'arbre ci-contre correspond à l'expression $(1 + 2 \times 3) \times 4$ (représentation infixe).

On peut noter cette expression en notation préfixe :

$$\times\ (+\ 1\ (\times\ 2\ 3))\ 4$$

Dans cette dernière, les parenthèses ne sont pas utiles (puisqu'on peut reconstruire l'arbre binaire strict à partir d'un parcours préfixe), on peut donc l'écrire simplement $\times + 1 \times 2\ 3\ 4$.



On pourrait utiliser le type suivant, où on utilise un caractère (`'+'`, `'x'`, etc) pour l'opérateur.

```
type expr =  F of int | N of char * expr * expr
```

En pratique, il vaut mieux définir clairement les opérateurs avec

```
type op = Plus | Moins | Fois | Div
type expr = F of int | N of op * expr * expr
```

On peut enrichir cette définition en autorisant des variables, pour représenter des expressions algébriques

```
type expr = F of int | Var of int | N of op * expr * expr
```

Où la construction `Var 1` représente une variable x_1.

On peut ajouter des opérateurs unaires (des fonctions).

```
type op1 = Exp | Sin
type expr = F of int | Var of int | N2 of op * expr * expr | N1 of op1 * expr
```

b) Formules booléennes

On s'intéresse à des formules booléennes, construites à partir de variables booléennes $x_1, \dots, x_n$ (qui prennent les valeurs `true` ou `false`), et des opérateurs logiques $\wedge$ (et), $\vee$ (ou) et $\neg$ (négation).

Par exemple : $x_1 \wedge \big(x_2 \vee (x_3 \wedge \neg x_1)\big)$ est une formule booléenne. Pour chaque choix de valeur de vérité des variables x_1, x_2, x_3, on peut évaluer cette formule, en un booléen.

On utilise le type récursif suivant

```
type prop = X of int | Non of prop | Ou of prop * prop | Et of prop * prop
```

```
let x1 = X 1 and x2 = X 2 and x3 = X 3
let p1 = Et (x1, Ou (x2, Non x3)) (* Représente x1 ∧ (x2 ∨ ¬x3) *)
```

```
let conjonction x y = Et (x,y)
```
```
conjonction : prop -> prop -> prop
```

1. Écrire deux fonctions `implique` et `equivalent` qui prennent en argument deux propositions p et q et renvoient une proposition booléenne représentant respectivement $p \Rightarrow q$ et $p \Leftrightarrow q$.
2. Écrire une fonction `max_ind` prenant en argument une proposition et renvoyant le maximum des indices des variables apparaissant dans cette proposition.

¶. Utiliser une fonction auxiliaire, avec un accumulateur des sommets visités, une liste des prochains sommets à visiter à la profondeur actuelle, et d'une liste des sommets à visiter à la profondeur suivante

3. Écrire des fonctions
   a) `toute` qui prend en argument une liste non vide de propositions et qui renvoie une proposition qui est vraie si et seulement si toutes les propositions de la liste sont vraies.
   b) `aucune` qui prend en argument une liste non vide de propositions et qui renvoie une proposition qui est vraie si et seulement si aucune des propositions de la liste n'est vraie.
   c) `une_seule` qui prend en argument une liste non vide de propositions et qui renvoient une proposition qui est vraie si et seulement si une seule des propositions de la liste est vraie.

Évaluation et satisfiabilité

4. Écrire une fonction `evalue : bool array -> prop -> bool`, qui prend en argument un tableau donnant les valeurs de vérités des variables et une formule booléenne et renvoie l'évaluation de cette formule.
5. Écrire une fonction `tautologie` qui prend en argument une proposition et décide si c'est une tautologie (c'est-à-dire si elle est vraie quelles que soient les valeurs de ses variables).
   Elle utilisera une fonction récursive auxilière `taut_aux` prenant en argument un entier `k` qui permet de construire en partant de la fin toutes les valeurs possibles d'un tableau d'évaluation (dans le but de les tester) :

```
let tautologie prop =
  let n = max_ind prop in
  let tab = Array.make (n+1) false in
  let rec taut_aux = function
    | 0 -> evalue tab prop
    | k -> begin
        (* On modifie la k-ième valeur du tableau tab *)
        (* Et on effectue des appels récursifs *)
      end
  in
  taut_aux n;;
```

6. En utilisant la question précédente, écrire une fonction `satisfiable` qui étant donné une proposition, décide si il existe des valeurs de ses variables pour lesquelles elle est vraie.

NP-complétude de SAT (HP)

Définition – Classes de complexité.

- On dit qu'un problème algorithmique est dans la classe P s'il admet une résolution par un algorithme avec une complexité polynomiale en la taille de ses arguments.
- On dit qu'un problème algorithmique est dans la classe NP s'il existe un algorithme qui peut vérifier qu'une solution est correcte, en temps polynomial.

Exemple Le problème de satisfiabilité, noté SAT, est dans la classe NP, puisque, étant donné un tableau de valeurs à attribuer aux variables, on peut vérifier que pour ces valeurs une formule s'évalue bien en `true`, en temps linéaire en la taille de la formule.

Définition On dit qu'un problème algorithmique $\mathcal{P}$ est NP-complet s'il est NP, et si tout problème NP se ramène à celui-ci par une réduction polynomiale.
Autrement dit, quel que soit $\mathcal{P}'$ de classe NP, il existe une construction φ telle que quelle que soit l'instance $\mathcal{I}$ de $\mathcal{P}'$, $\varphi(\mathcal{I})$ soit une instance de $\mathcal{P}$, de taille au plus polynomiale en la taille de $\mathcal{I}$, dont la résolution permette la résolution de $\mathcal{I}$.

Remarque En particulier, s'il existant un algorithme de complexité polynomiale permettant de résoudre un problème NP-complet, alors il existerais des algorithmes polynomiaux permettant de résoudre tous les problème de la classe NP.

Théorème Le problème SAT de satisfiabilité de formules logiques est NP-complet.

Algorithme de Quine

On s'intéresse à un autre algorithme pour la satisfiabilité d'une formule booléenne.
Pour cela, on étend la définition d'une formule logique, en autorisant des feuilles représentant les valeurs `true` et `false` :

```
type prop = V | F | X of int | Non of prop | Ou of prop * prop | Et of prop * prop
```

7. Écrire une fonction `simplifie : prop -> prop` qui prend en argument une proposition, et la simplifie (en une formule qui a le même sens logique), en retirant les feuilles `V` et `F` (à moins que la formule ne soit réduite à `V` et `F`).
8. Écrire une fonction `satisfiable : prop -> bool` utilisant le principe suivant :
   - à partir d'une formule f contenant au moins une variable, on choisit une variable x_n apparaissant dans f et on construit deux formules $f_{|x_n=V}$ et $f_{|x_n=F}$ obtenue en remplaçant dans f les occurrences de x_n par V et F respectivement.
   - la formule f est alors satisfiable si et seulement si la formule $f_{|x_n=V} \vee f_{|x_n=F}$ est satisfiable.

Remarque En pratique, il serait plus rapide d'éliminer en priorité les variables apparaissant le plus souvent.

2) Mots de Dyck, nombre d'arbres binaires

On s'intéresse à des mots formés de parenthèses ouvrantes et fermantes, comme `(()())()(`. On représente un tel mot par les deux types suivants

```
type par = O | F
type mot = par list
```

1. Écrire une fonction `compte_diff : mot -> int` qui prend en argument un mot u et renvoie la différence entre le nombre de parenthèses ouvrantes $n_O(u) - n_F(u)$.

On s'intéresse à l'ensemble des mots «bien parenthésés», appelés mots de Dyck. On admet qu'un mot u est bien parenthésé si et seulement si $n_O(u) = n_F(u)$ et tous les préfixes v de u vérifient $n_O(v) \geq n_F(v)$.

2. Écrire une fonction `dyck : mot -> bool` qui teste si le mot est bien parenthésé. On veut une complexité linéaire.

Remarque On pourrait donner une définition inductive de l'ensemble des mots bien parenthésés : le mot vide est bien parenthésé, et si u, v sont bien parenthésés, le mot $OuFv$ est bien parenthésé.

3. On note C_n le nombre de mots bien parenthésés de taille $2n$.
   a) Déterminer C_0, C_1, C_2, C_3.
   b) Déterminer une relation de récurrence vérifiée par C_n.

Remarque Les nombres C_n sont les nombres de Catalan. On peut montrer que $C_n = \frac{1}{n+1}\binom{2n}{n}$.

Lien avec les arbres binaires

On travaille avec des arbres binaires généraux non étiquettés, de type

```
type arbre = Vide | N of arbre * arbre
```

On rappel qu'un arbre binaire est strict si chaque nœud a ou bien 0 ou bien 2 enfants (non vides).

4. Montrer qu'un arbre binaire stricte est de taille impaire.
5. Montrer que le nombre de mots de Dyck de taille $2n$ est égal au nombre d'arbres stricts de taille $2n + 1$.
6. Écrire une fonction `arbre_vers_dyck : arbre -> mot`.
7. Écrire la réciproque `dyck_vers_arbre : mot -> arbre`.