	OCaml	Python
test d'égalité	`=`	`==`
test de différence	`<>`	`!=`
division euclidienne	`/`	`//`
modulo	`mod`	%
et	`&&`	`and`
ou	`||`	`or`

Opérateurs en OCaml/Python

	OCaml
Définition	`let r = ref ...`
Accéder à la valeur	`!r`
Modifier la valeur	`r := ...`

Références

- `()` est une valeur de type `unit`, qui signifie rien.
- Si `f : 'a -> 'b -> 'c` alors `f` prend deux arguments de type `'a` et `'b` et renvoie un résultat de type `'c`.
  De plus, `f x` (application partielle) est la fonction de type `'b -> 'c` qui à `y` associe `f x y`.
- On créé souvent une liste avec une fonction récursive :

```
let rec range n = (* renvoie [n - 1; ...; 1; 0] *)
    if n = 0 then []
    else (n-1)::range (n - 1)
```

On peut éventuellement utiliser une référence sur une liste :

```
let range n =
    let l = ref [] in (* moins idiomatique *)
    for i = n - 1 downto 0 do
        l := i::!l
    done; !l
```

- **Impossible de modifier une liste** `l`.
  `e::l` renvoie une **nouvelle liste** mais ne modifie pas `l`.
  `let l = [] in ...` ne sert à rien.
- **Impossible d'écrire** `l.(i)` pour une liste `l` : il faut utiliser une fonction récursive pour parcourir une liste.

```
let rec appartient e l = match l with
    | [] -> false
    | t::q -> e = t || appartient e q
```

- Chaque cas d'un `match` sert à **définir de nouvelles variables**, et ne permet pas de comparer des valeurs.

Exemple : Dans `appartient`, il ne faut pas écrire `| e::q -> ...` (ceci remplacerait le `e` en argument).
Pour tester l'égalité : `| t::q -> if t = e then ...` ou `| t::q when t = e -> ...`.

- Les indices d'un tableau à n éléments vont de 0 à $n-1$:

```
let appartient e t =
    let r = ref false in
    for i = 0 to  n - 1 do
        if t.(i) = e then r := true
    done; !r
```

- **Impossible de renvoyer une valeur à l'intérieur d'une boucle** (pas de `return` en OCaml) :

```
let appartient e t =
    for i = 0 to n - 1 do
        if t.(i) = e then true (* NE MARCHE PAS !!! *)
    done; false
```

- Types union et enregistrement :

```
(* type union : l'un OU l'autre *)
type 'a option = None | Some of 'a
(* on utilise match (et fonction récursive)
   sur les types union *)
let add x y = match x, y with
    | None, _ | _, None -> None
    | Some x1, Some y1 -> Some (x1 + y1)

(* type enregistrement : l'un ET l'autre *)
type fraction = { num : int; denum : int }
let f = { num = 1; denum = 4 }
f.denum (* donne 4 *)
```

Remarque : l'égalité (avec `=`) est automatiquement définie avec un nouveau type. Exemple : si `t1` et `t2` sont des arbres binaires, alors `t1 = t2` vaut `true` si `t1 = V` et `t2 = V` ou si `t1 = N(r1, g1, d1)`, `t2 = N(r2, g2, d2)` avec `r1 = r2`, `g1 = g2` et `d1 = d2`.

- Quelques fonctions utiles (avec les équivalents sur `Array`) :
  - `Array.make_matrix n p e` renvoie une matrice $n \times p$ dont chaque élément est `e`
  - `List.iter f l` appelle `f` sur chaque élément de `l`
  - `List.map f [a1; ...; an]` renvoie `[f a1; ...; f an]`
  - `List.filter f l` renvoie la liste des `e` tels que `f e`
  - `List.exists f l` et `List.for_all f l`.

Type	Exemple	Obtenir valeur	Modification	Taille	Création
`array`	`[|1; 2|] : int array`	`t.(i)`	`t.(i) <- ...`	`Array.length` (en O(1))	`Array.make n e` (en O(n))
`string`	`"abc" : string`	`s.[i]`		`String.length` (en O(1))	`String.make n e`
`list`	`[1; 2] : int list`	Fonction récursive		`List.length` (en O(n))	
`tuple`	`(1, "abc", [1; 2]) : int*string*int list`	`let a, b, c = t`			

Remarque : une « liste » Python est en réalité un tableau.

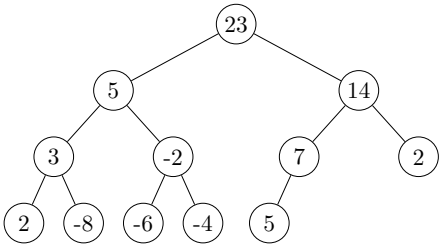
File de priorité

- Une **file de priorité max** (FP max) est une structure de données possédant les opérations suivantes :
  - Extraire maximum : supprime et renvoie le maximum
  - Ajouter élément
  - Tester si la FP est vide

  On définit une FP min en remplaçant maximum par minimum.
- Il est possible d'utiliser un arbre binaire de recherche pour implémenter une FP max, en utilisant le fait que le maximum est tout à droite de l'arbre. Les opérations d'ajout et d'extraction sont alors linéaires en la hauteur de l'arbre. On peut mettre à jour un élément (changer sa valeur) en le supprimant puis en le réajoutant (avec la nouvelle valeur).

Tas

- Un **tas max** est un arbre binaire presque complet (tous les niveaux, sauf le dernier, sont complets) où chaque noeud est plus grand que ses fils.
  Remarque : la racine est le maximum du tas.



Exemple de tas max

  Un tas min est comme un tas max, sauf que chaque noeud doit être plus petit que ses fils.
- Soit T un arbre binaire presque complet de hauteur h et avec n noeuds. Alors $h = \lfloor \log_2(n) \rfloor$ (donc $\boxed{h = \mathrm{O}(\log(n))}$).

  > Preuve : T contient plus de sommets qu'un arbre binaire complet de hauteur $h-1$ et moins de sommets qu'un arbre binaire complet de hauteur h, donc :
  >
  > $$\begin{aligned} & 2^h - 1 < n \leq 2^{h+1} - 1 \\ \implies & 2^h \leq n < 2^{h+1} \\ \implies & h \leq \log_2(n) < h+1 \\ \implies & h = \lfloor \log_2(n) \rfloor \end{aligned}$$

- On stocke les noeuds du tas dans un tableau `t` tel que :
  - `t.(0)` est la racine de `t`.
  - `t.(i)` a pour fils `t.(2*i + 1)` et `t.(2*i + 2)`, si ceux-ci sont définis.

  Le père de `t.(j)` est donc `t.((j - 1)/2)` (si `j` $\neq 0$).
  Exemple : le tas en exemple ci-dessus est représenté par `t = [|23; 5; 14; 3; -2; 7; 2; 2; -8; -6; -4; 5|]`.
- En pratique, comme on ne peut pas ajouter d'élément à un tableau, on utilise un tableau `t` plus grand que le nombre d'éléments du tas pour pouvoir y ajouter des éléments. On stocke le nombre d'éléments dans une variable `n`.

```
type 'a tas = {t : 'a array; mutable n : int}

let swap tas i j = (* échange tas.t.(i) et tas.t.(j) *)
    let tmp = tas.t.(i) in
    tas.t.(i) <- tas.t.(j);
    tas.t.(j) <- tmp
```

- On utilise deux fonctions auxiliaires pour rétablir la propriété de tas après modification :
  - `up tas i` : suppose que `tas` est un tas max sauf `tas.t.(i)` qui peut être supérieur à son père. Fait monter (on dit aussi percoler) `tas.t.(i)` de façon à obtenir un tas max.
  - `down tas i` : suppose que `tas` est un tas max sauf `tas.t.(i)` qui peut être inférieur à un fils. Fait descendre `tas.t.(i)` de façon à obtenir un tas max.

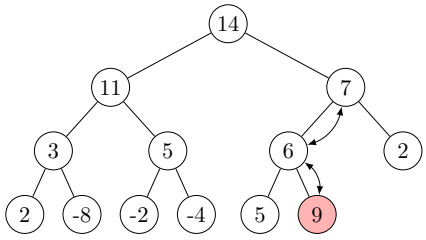
```
let rec up h i =
    let p = (i - 1)/2 in
    if i <> 0 && tas.t.(p) < tas.t.(i) then (
        swap h i p;
        up h p
    )

let rec down tas i =
  let m = ref i in (* maximum parmi i et ses fils *)
  if 2*i + 1 < tas.n && tas.t.(2*i + 1) > tas.t.(!m)
  then m := 2*i + 1;
  if 2*i + 2 < tas.n && tas.t.(2*i + 2) > tas.t.(!m)
  then m := 2*i + 2;
  if !m <> i then (
    swap h i !m;
    down h !m
  )
```

- Pour ajouter un élément à un tas : l'ajouter comme dernière feuille (dernier indice du tableau) puis faire remonter.
  Complexité : $\mathrm{O}(h) = \boxed{\mathrm{O}(\log(n))}$.

```
let ajoute tas e =
    tas.t.(tas.n) <- e;
    up h tas.n;
    tas.n <- tas.n + 1
```

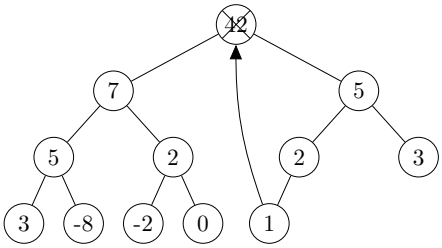
  Exemple : ajout de 9 dans un tas.



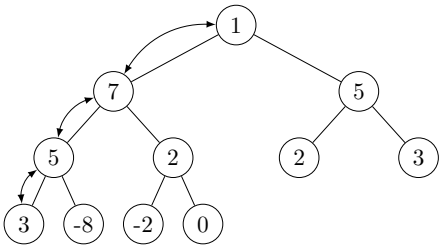
- Pour extraire le maximum d'un tas : échanger la racine avec la dernière feuille puis descendre la nouvelle racine.
  Complexité : $\mathrm{O}(h) = \boxed{\mathrm{O}(\log(n))}$.

```
let rec extract_max h =
    swap h 0 (h.n - 1);
    h.n <- h.n - 1;
    down h 0;
    h.a.(h.n)
```

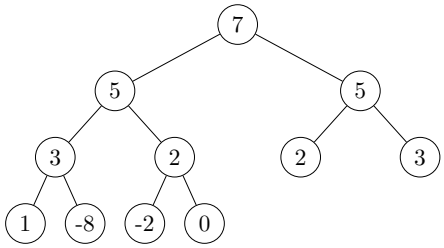
Exemple :



Suppression de la racine qu'on remplace par la dernière feuille



Percolation



Tas obtenu après extraction du maximum

Applications

- **Tri avec une file de priorité** : on ajoute tous les éléments dans un tas puis on les extrait un par un.
  On obtient ainsi le **tri par tas** en $O(n \log(n))$ avec un tas. On peut même utiliser le tableau en entrée comme tableau `tas.t` du tas, ce qui permet d'obtenir un tri en place, c'est-à-dire sans utiliser de tableau supplémentaire (O(1) en mémoire).
- **Algorithme de Dijkstra**, pour extraire le sommet de distance estimée minimum à chaque itération. Dans le pseudocode suivant, on peut utiliser une file de priorité pour `q` :

```
Initialement : q contient tous les sommets
               dist.(s) <- 0
               dist.(v) <- infini, si v <> s

Tant que q est non vide:
    Extraire u de q tel que dist.(u) soit minimum
    Pour tout voisin v de u:
        Si dist.(u) + w(u, v) < dist.(v):
            dist.(v) <- dist.(u) + w(u, v)
```

- **Algorithme de Kruskal**, pour obtenir l'arête de poids minimum à chaque itération.

- Un **couplage** de $G = (V, E)$ est un ensemble d'arêtes $M \subseteq E$ tel qu'aucun sommet ne soit adjacent à 2 arêtes de M :

$$\forall e_1, e_2 \in M,\ e_1 \neq e_2 \implies e_1 \cap e_2 = \emptyset$$

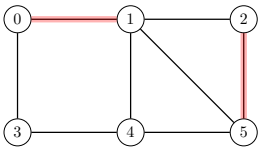
  Un sommet $v \in V$ est **couvert** par M s'il appartient à une arête de M. Sinon, v est **libre** pour M.

- Exercice : Écrire une fonction `est_couplage : (int*int) list -> int > bool` déterminant si une liste d'arêtes (chaque arête étant un couple) forme un couplage.

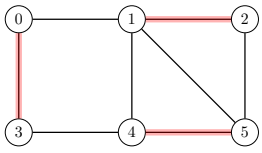
```
let est_couplage m n = (* n = nb de sommets *)
    let couvert = Array.make n false in
    let rec aux l = match l with
        | [] -> true
        | (u, v)::q ->
            if couvert.(u) || couvert.(v) then false
            else begin
                couvert.(u) <- true;
                couvert.(v) <- true;
                aux q
            end
    in aux m
```

- La **taille** de M, notée $|M|$, est son nombre d'arêtes.
  M est un couplage **maximum** s'il n'existe pas d'autre couplage de taille strictement supérieure.
  M est un couplage **maximal** s'Il n'existe pas de couplage M' tel que $M \subsetneq M'$.
  M est un couplage **parfait** si tout sommet de G appartient à une arête de M.
  Un chemin est **élémentaire** s'il ne passe pas deux fois par le même sommet.
  Un chemin élémentaire de G est M-**alternant** si ses arêtes sont alternativement dans M et dans $E \setminus M$.
  Un chemin de G est M-**augmentant** s'il est M-alternant et si ses extrémités sont libres pour M.

- Soit M un couplage de G et P un chemin M-augmentant. Alors $M \Delta P$ est un couplage de G et $|M \Delta P| = |M| + 1$.



Un couplage M



$M \Delta P$, où
$P = 3 - 0 - 1 - 2 - 5 - 4$

- M est un couplage maximum de G $\iff$ Il n'existe pas de chemin M-augmentant dans G.
  Preuve :

$\implies$ Soit M un couplage maximum. Supposons qu'il existe un chemin M-augmentant P. Alors $M \Delta P$ est un couplage de G et $|M \Delta P| > |M|$: absurde.

$\impliedby$ Supposons qu'il existe un couplage M^* vérifiant $|M^*| > |M|$. Considérons $G^* = (V, M \Delta M^*)$.
Les degrés des sommets de G^* sont au plus 2, donc G^* est composé de cycles et de chemins uniquement.
Chacun de ces cycles et chemins alternent entre des arêtes de M et des arêtes de M^*.
Comme $|M^*| > |M|$, un de ces chemins contient plus d'arêtes de M^* que de M : c'est un chemin M-augmentant.

- On en déduit l'algorithme :

Couplage maximum par chemin augmentant

Entrée : Graphe $G = (V, E)$
Sortie : Couplage maximum M de G

$M \leftarrow \emptyset$
Tant que il existe un chemin M-augmentant P dans G **:**
  $M \leftarrow M \Delta P$

Remarques :
- On peut aussi partir d'un couplage initialement non vide, et on obtiendra quand même un couplage maximum à la fin.
- Il est difficile de trouver un chemin M-augmentant dans un graphe quelconque : c'est pour cela qu'on s'intéresse aux graphes bipartis dans la suite.

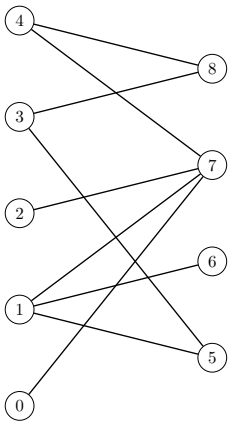
- Un graphe $G = (V, E)$ est **biparti** s'il existe V_1 et V_2 tels que $V = V_1 \sqcup V_2$ et toute arête de E ait une extremité dans V_1 et l'autre dans V_2.
  Remarque : cela revient à donner une couleur à chaque sommet de façon à ce que les extrémités de chaque arête soient de couleurs différentes.

- Exercice : Écrire une fonction `est_biparti : int list array -> bool` pour déterminer si un graphe (représenté par liste d'adjacence) est biparti, en complexité linéaire.
  Solution : On fait un parcours en profondeur depuis un sommet quelconque en alternant les couleurs 0 et 1.

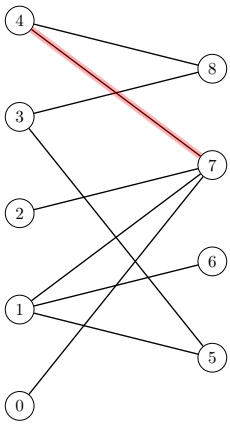
```
let est_biparti g =
    let n = Array.length g in
    let couleurs = Array.make n (-1) in
    let rec aux u c = (* on donne la couleur c à u *)
        if couleurs.(u) = -1 then begin
            couleurs.(u) <- c;
            List.for_all (fun v -> aux v (1 - c)) g.(u)
        end else couleurs.(u) = c
    in aux 0 0
```

- Il est facile de trouver un chemin M-augmentant dans un graphe biparti $G = V_1 \sqcup V_2$:
  - Partir d'un sommet $v \in V_1$ libre.
  - Se déplacer (DFS) en alternant entre des arêtes de M et des arêtes de $G \setminus M$, sans revenir sur un sommet visité.
  - Si on arrive à un sommet libre de V_2, alors on a trouvé un chemin M-augmentant.

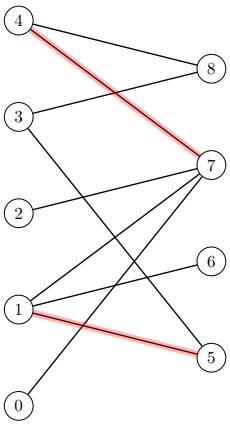
Exemple de recherche d'un couplage maximum par chemin augmentant dans un graphe biparti :



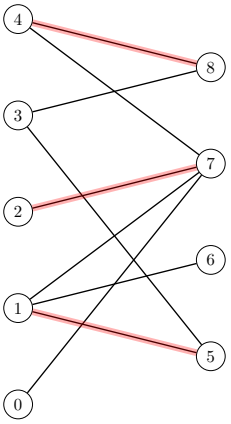
Graphe biparti G et
couplage $M = \emptyset$



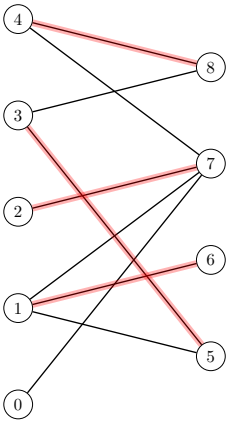
$M \leftarrow M \Delta P$, où
$P = 4 - 7$



$M \leftarrow M \Delta P$, où
$P = 1 - 5$



$M \leftarrow M \Delta P$, où
$P = 2 - 7 - 4 - 8$



$M \leftarrow M \Delta P$, où
$P = 3 - 5 - 1 - 6$

- Définitions :

	Signification	Exemple
Alphabet	Ensemble fini de lettres	$\Sigma = \{a, b\}$
Mot	Suite finie de lettre	$m = abaa$
ε	Mot vide (sans lettre)	
Langage	Ensemble de mots	$L = \{\varepsilon, a, baa\}$

ε est un mot, pas une lettre.

- Opérations sur des mots $u = u_1...u_n$ et $v = v_1...v_p$:

	Définition	Exemple avec $u = ab$ et $v = cbc$
Concaténation	$uv = u_1...u_nv_1...v_p$	$uv = abcbc$
Puissance	$u^n = u...u$	$u^3 = ababab$
Taille	$\|u\| = n$	$\|u\| = 2$

Deux mots sont égaux s'ils ont la même taille et les mêmes lettres.

- Opérations sur des langages $L_1 = \{\varepsilon, ab\}$ et $L_2 = \{b, ab\}$:

	Définition	Exemple
Concaténation	$L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}$	$L_1L_2 = \{b, ab, abb, abab\}$
Puissance	$L^n = \{u^n \mid u \in L\}$	$L_1^2 = \{\varepsilon, ab, abab\}$
Etoile	$L^* = \bigcup_{k\in\mathbb{N}} L^k$	$L_1^* = \{\varepsilon, ab, abab...\}$

Comme L_1 et L_2 sont des ensembles, on peut aussi considérer $L_1 \cup L_2$, $L_1 \cap L_2$...

- Les langages réguliers sont tous ceux qu'on peut obtenir avec les règles suivantes :
  - Un langage fini est régulier
  - L_1 et L_2 réguliers $\implies$ $L_1 \cup L_2$ régulier
  - L_1 et L_2 réguliers $\implies$ L_1L_2 régulier
  - L régulier $\implies$ L^* régulier

  Exemples : Un alphabet Σ est toujours régulier car fini. Σ^* est régulier car est l'étoile du langage régulier Σ.

- Une expression régulière est une suite de symboles contenant : lettres, $\emptyset$, ε, $|$ (union, parfois notée $+$), *.
  À chaque expression régulière e on associe un langage $L(e)$.

  Exemple : Le langage de l'expression régulière $e = a^*b \mid \varepsilon$ est $L(e) = (\{a\}^*\{b\}) \cup \{\varepsilon\}$.

- L régulier $\iff$ $\exists$ une expression régulière de langage L.

Exemples de langages réguliers sur $\Sigma = \{a, b\}$:

1. Mots contenant au plus un a : $b^*(a|\varepsilon)b^*$.
2. Mots de taille $n \equiv 1 \mod 3$: $((a|b)^3)^*(a|b)$.
3. Mots contenant un nombre pair de a : $(ab^*a|b)^*$.
4. Mots contenant un nombre impair de a : $b^*a(ab^*a|b)^*$.

- Définition possible d'expression régulière en OCaml :

```
type 'a regexp =
    | Vide | Epsilon | L of 'a
    | Union of 'a regexp * 'a regexp
    | Concat of 'a regexp * 'a regexp
    | Etoile of 'a regexp

(* définition de e ci-dessus *)
let e = Union(Concat(Etoile(a), b), Epsilon)
```

Exemple : déterminer si ε appartient au langage de e.

```
let rec has_eps e = match e with
    | Vide | L _ -> false
    | Epsilon | Etoile _ -> true
    | Union(e1, e2) -> has_eps e1 || has_eps e2
    | Concat(e1, e2) -> has_eps e1 && has_eps e2
```

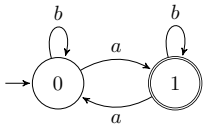
- Quelques techniques de preuve :
  - Sur des mots : récurrence sur la taille du mot.
  - Pour montrer l'égalité de deux langages : double inclusion ou suite d'équivalences.
  - Pour montrer $P(L)$ pour un langage régulier L : par induction ($\approx$ récurrence), en montrant le cas de base (si L est un langage fini) et les cas d'hérédité ($P(L_1) \wedge P(L_2) \implies P(L_1L_2)$, $P(L_1) \wedge P(L_2) \implies P(L_1 \cup L_2)$, $P(L) \implies P(L^*)$).
  - Pour montrer $P(e)$ pour une expression régulière e : par induction, en montrant les cas de base ($P(\emptyset)$, $P(\varepsilon)$, $P(a)$, $\forall a \in \Sigma$) et les cas d'hérédité ($P(e_1)$ et $P(e_2) \implies P(e_1e_2)$ et $P(e_1|e_2)$, $P(e) \implies P(e^*)$).

  Exemple : Le miroir d'un mot $u = u_1...u_n$ est $\widetilde{u} = u_n...u_1$ et le miroir d'un langage L est $\widetilde{L} = \{\widetilde{u} \mid u \in L\}$.
  Montrons : L régulier $\implies$ $\widetilde{L}$ régulier.
  On pourrait le montrer par récurrence, mais il est peut-être plus simple de définir une fonction $f(e)$ qui à une expression régulière e associe une expression régulière pour le miroir de $L(e)$:
  - $f(\emptyset) = \emptyset$, $f(\varepsilon) = \varepsilon$ et $\forall a \in \Sigma, f(a) = a$.
  - $f(e_1e_2) = f(e_2)f(e_1)$ (le miroir de uv est $\widetilde{v}\widetilde{u}$).
  - $f(e_1|e_2) = f(e_1)|f(e_2)$.
  - $f(e_1^*) = f(e_1)^*$.

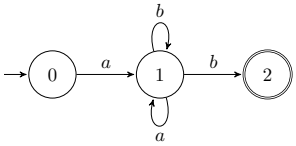
  On a bien défini une fonction f telle que, pour toute expression régulière e, $f(e)$ est une expression régulière de $\widetilde{L(e)}$. Donc le miroir d'un langage régulier est régulier.

- Un **automate** est un 5-uplet $A = (\Sigma, Q, I, F, E)$ où :
  - Σ est un alphabet.
  - Q est un ensemble fini d'**états**.
  - $I \in Q$ est un ensemble d'**états initiaux**.
  - $F \subseteq Q$ est un ensemble d'**états acceptants** (ou **finaux**)
  - $E \subseteq Q \times \Sigma \times Q$ est un ensemble de **transitions**.
    On peut remplacer l'ensemble E de transitions par une **fonction de transition** $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$
- – Un chemin dans A est **acceptant** s'il part d'un état initial pour aller dans un état final.
  - Un mot est **accepté** par A s'il est l'étiquette d'un chemin acceptant.
  - Le **langage** $L(A)$ **accepté** (ou **reconnu**) par A est l'ensemble des mots acceptés par A.

  Exemple :
  - Soit A l'automate suivant :

    Alors $L(A) = \{$ mot avec un nombre impair de a $\}$ $= L(b^*a(b^*ab^*ab^*)^*)$.
  - Le langage $a(a+b)^*b$ est reconnaissable, car reconnu par l'automate ci-dessous.

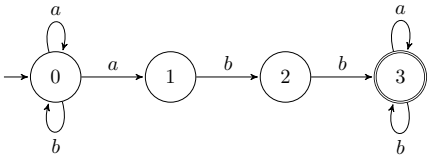
- Pour déterminer algorithmiquement si un automate A accepte un mot $u = u_1...u_n$, on peut calculer de proche en proche $Q_0 = I$, $Q_1 =$ états accessibles depuis Q_0 avec la lettre u_1, $Q_2 =$ états accessibles depuis Q_0 avec la lettre u_2... et regarder si Q_n contient un état final.
- Soit $A = (\Sigma, Q, I, F, E)$ un automate.
  - A est **complet** si : $\forall q \in Q,\ \forall a \in \Sigma,\ \exists (q, a, q') \in E$
  - Un automate $(\Sigma, Q, \{q_i\}, F, E)$ est **déterministe** si :
    1. Il n'y a qu'un seul état initial q_i.
    2. $(q, a, q_1) \in E \wedge (q, a, q_2) \in E \implies q_1 = q_2$: il y a au plus une transition possible en lisant une lettre depuis un état
  - Un automate déterministe et complet possède une unique transition possible depuis un état en lisant une lettre.
    On a alors une fonction de transition de la forme $\delta : Q \times \Sigma \longrightarrow Q$ qu'on peut étendre en $\delta^* : Q \times \Sigma^* \longrightarrow Q$ définie par :
    * $\delta^*(q, \varepsilon) = q$
    * Si $u = av$, $\delta^*(q, av) = \delta^*(\delta(q, a), v)$

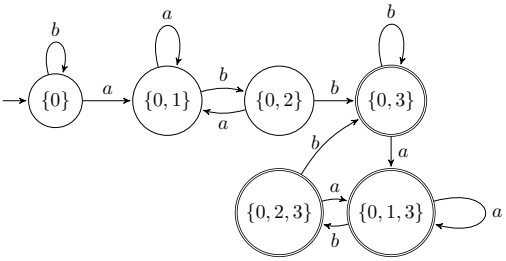
    Ainsi, $\delta^*(q, u)$ est l'état atteint en lisant le mot u depuis l'état q.
    On a alors $u \in L(A) \iff \delta^*(q_i, u) \in F$.
- Deux automates sont **équivalents** s'ils ont le même langage.
- Soit A un automate. Alors A est équivalent à un automate déterministe complet.

  Preuve : Utilise l'automate des parties $A' = (\Sigma, \mathcal{P}(Q), \{I\}, F', \delta')$ où $F' = \{X \subseteq Q \mid X \cap F \neq \emptyset\}$

  Remarque : Si on veut juste un automate complet (pas forcément déterministe), on peut ajouter un état poubelle vers lequel on redirige toutes les transitions manquantes. Dans l'automate des parties, cet état poubelle est $\emptyset$.

  Exemple : Un automate A avec son déterminisé A'.

- Soit L un langage reconnaissable. Alors $\overline{L}$ $(= \Sigma^* \backslash L)$ est reconnaissable.

  Preuve : Soit $A = (\Sigma, Q, q_i, F, \delta)$ un automate déterministe complet reconnaissant L. Alors $A' = (\Sigma, Q, q_i, Q \backslash F, \delta)$ (on inverse états finaux et non-finaux) reconnaît $\overline{L}$.
- Soient L_1 et L_2 des langages reconnaissables. Alors :
  - $L_1 \cap L_2$ est reconnaissable.
  - $L_1 \cup L_2$ est reconnaissable.
  - $L_1 \backslash L_2$ est reconnaissable.

  Preuve : Soient $A_1 = (Q_1, q_1, F_1, \delta_1)$ et $A_2 = (Q_2, q_2, F_2, \delta_2)$ des automates finis déterministes complets reconnaissants L_1 et L_2. Soit $A = (Q_1 \times Q_2, (q_1, q_2), F, \delta)$ (**automate produit**) où :
  - $F = F_1 \times F_2$: A reconnait $L_1 \cap L_2$.
  - $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ ou } q_2 \in F_2\}$: A reconnait $L_1 \cup L_2$.
  - $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ et } q_2 \notin F_2\}$: A reconnait $L_1 \setminus L_2$.

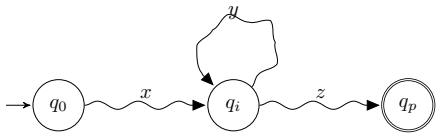
  Remarque : Comme l'ensemble des langages reconnaissables est égal à l'ensemble des langages rationnels, l'ensemble des langages rationnels est aussi stable par complémentaire, intersection et différence.
  Il n'y a pas de stabilité par inclusion (L rationnel et $L' \subseteq L$ n'implique pas forcément L' rationnel).

- (**Lemme de l'étoile** ♡) Soit L un langage reconnaissable par un automate à n états.
  Si $u \in L$ et $|u| \geq n$ alors il existe des mots x, y, z tels que :
  - $u = xyz$
  - $|xy| \leq n$
  - $y \neq \varepsilon$
  - $xy^*z \subseteq L$ (c'est-à-dire : $\forall k \in \mathbb{N},\ xy^kz \in L$)

  Preuve : Soit $A = (\Sigma, Q, I, F, \delta)$ un automate reconnaissant L et $n = |Q|$.
  Soit $u \in L$ tel que $|u| \geq n$.
  u est donc l'étiquette d'un chemin acceptant C :

  $$q_0 \in I \xrightarrow{u_0} q_1 \xrightarrow{u_1} ... \xrightarrow{u_{p-1}} q_p \in F$$

  C a $p + 1 > n$ sommets donc passe deux fois par un même état $q_i = q_j$ avec $i < n$. La partie de C entre q_i et q_j forme donc un cycle.

  Soit $x = u_0u_1...u_{i-1}$, $y = u_i...u_j$ et $z = u_{j+1}...u_{p-1}$.
  xy^kz est l'étiquette du chemin acceptant obtenu à partir de C en passant k fois dans le cycle.

  Application : $L_1 = \{a^nb^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable.
  Preuve : Supposons L_1 reconnaissable par un automate à n états. Soit $u = a^nb^n$. Clairement, $u \in L_1$ et $|u| \geq n$.
  D'après le lemme de l'étoile : il existe x, y, z tels que $u = xyz$, $|xy| \leq n$, $y \neq \varepsilon$ et $xy^*z \subseteq L$.
  Comme $|xy| \leq n$, x et y ne contiennent que des a : $x = a^i$ et $y = a^j$. Comme $y \neq \varepsilon$, $j > 0$.
  En prenant $k = 0$: $xy^0z = xz = a^{n-j}b^n$.
  Or $j > 0$ donc $a^{n-j}b^n \notin L_1$: absurde.

Automate de Glushkov : régulier $\implies$ reconnaissable

- Une expression régulière est **linéaire** si chaque lettre y apparaît au plus une fois : $a(d+c)^*b$ est linéaire mais pas $ac(a+b)$.

- Soit L un langage. On définit :
  - $P(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$ (premières lettres des mots de L)
  - $S(L) = \{a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset\}$ (dernières lettres des mots de L)
  - $F(L) = \{u \in \Sigma^2 \mid \Sigma^* u \Sigma^* \cap L \neq \emptyset\}$ (facteurs de longueur 2 des mots de L)
  - L est **local** si, pour tout mot $u = u_1u_2...u_n \neq \varepsilon$:

    $$u \in L \iff u_1 \in P(L) \wedge u_n \in S(L) \wedge \forall k, u_k u_{k+1} \in F(L)$$

    Il suffit donc de regarder la première lettre lettre, la dernière lettre et les facteurs de taille 2 pour savoir si un mot appartient à un langage local.
    Remarques :
    * $\implies$ est toujours vrai donc il suffit de prouver $\impliedby$.
    * Définition équivalente :

      $$L \text{ local} \iff L \setminus \{\varepsilon\} = (P(L) \cap S(L)) \setminus N(L)$$

      où $N(L) = \Sigma^2 \setminus F(L)$.

    Exemples :
    - Si $L_2 = (ab)^*$ alors $P(L_2) = \{a\}$, $S(L_2) = \{b\}$ et $F(L_2) = \{ab, ba\}$. De plus si $u = u_1u_2...u_n \neq \varepsilon$ avec $u_1 \in P(L), u_n \in S(L)$, et $\forall k, u_k u_{k+1} \in F(L)$ alors $u_1 = a$, $u_n = b$ et on montre (par récurrence) que $u = abab...ab \in {}_2$. Donc L_2 est local.
    - Si $L_3 = a^* + (ab)^*$ alors $P(L_3) = \{a\}$, $S(L_3) = \{a, b\}$, $F(L_3) = \{aa, ab, ba\}$. Soit $u = aab$. La première lettre de u est dans $P(L_3)$, la dernière dans $S(L_3)$ et les facteurs de u sont aa et ba qui appartiennent à $F(L_3)$. Mais $u \notin L_3$, ce qui montre que L_3 n'est pas local.

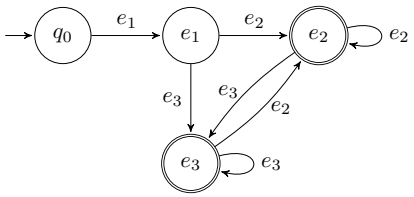
- Un automate déterministe (Σ, Q, q_0, F, E) est **local** si toutes les transitions étiquetées par une même lettre aboutissent au même état : $(q_1, a, q_2) \in E \wedge (q_3, a, q_4) \in E \implies q_2 = q_4$

- Un langage local L est reconnu par un automate local.

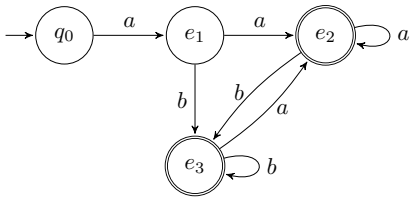
  Preuve : L est reconnu par (Σ, Q, q_0, F, E) où :
  - $Q = \Sigma \cup \{q_0\}$: un état correspond à la dernière lettre lue
  - $F = S(L)$ si $\varepsilon \notin L$, sinon $F = S(L) \cup \{q_0\}$.
  - $E = \{(q_0, a, a) \mid a \in P(L)\} \cup \{(a, b, b) \mid ab \in F(L)\}$

- L'**algorithme de Berry-Sethi** permet de construire un automate à partir d'une expression régulière e.

  Exemple avec $e = a(a+b)^*$:
  1. On linéarise e en e', en remplaçant chaque occurrence de lettre dans e par une nouvelle lettre : $e' = e_1(e_2 + e_3)^*$
  2. On peut montrer que $L(e')$ est un langage local.
  3. Un langage local est reconnu par l'automate local $A = (\Sigma, Q, q_0, F, E)$

  4. On fait le remplacement inverse de 1. sur les transitions de A pour obtenir un automate reconnaissant $L(e)$:

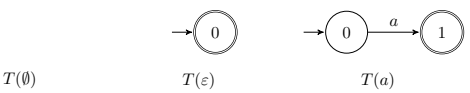
Automate de Thompson : régulier $\implies$ reconnaissable

- Une ε-transition est une transition étiquetée par ε.

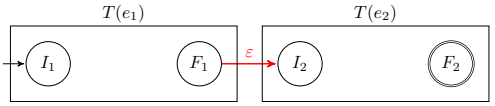
- Un automate avec ε-transitions est équivalent à un automate sans ε-transitions.

  Preuve : Si $A = (\Sigma, Q, I, F, \delta)$ est un automate avec ε-transitions, on définit $A' = (\Sigma, Q, I', F, \delta')$ où :
  - I' est l'ensemble des états atteignables depuis un état de I en utilisant uniquement des ε-transitions.
  - $\delta'(q, a)$ est l'ensemble des états q' tel qu'il existe un chemin de q à q' dans A étiqueté par un a et un nombre quelconque de ε (ce qui peut être trouvé par un parcours de graphe).

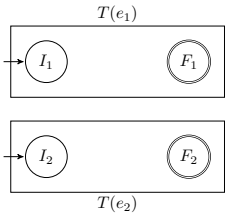
- L'**automate de Thompson** est construit récursivement à partir d'une expression régulière e :
  - Cas de base :

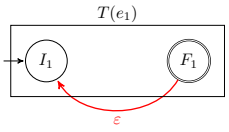
  - $T(e_1e_2)$: ajout d'une ε-transition depuis chaque état final de $T(e_1)$ vers chaque état initial de $T(e_2)$.

  - $T(e_1|e_2)$: union des états initiaux et des états finaux.



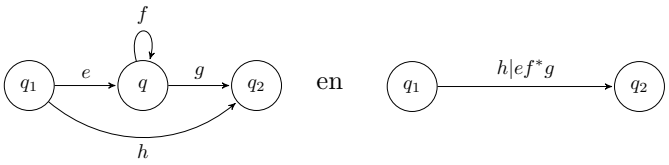
– $T(e_1^*)$: ajout d'une ε-transition depuis chaque état final vers chaque état initial.



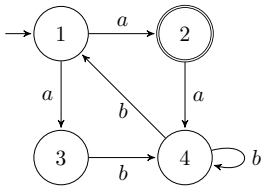
Élimination des états : reconnaissable $\Longrightarrow$ régulier

- Tout automate est équivalent à un automate avec un unique état initial sans transition entrante et un unique état final sans transition sortante.
  Preuve : On ajoute un état initial q_i et un état final q_f et des transitions ε depuis q_i vers les états initiaux et depuis les états finaux vers q_f.

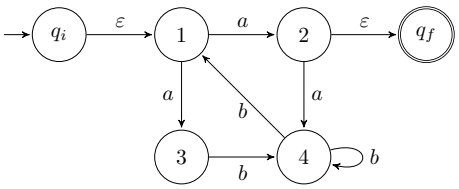
- **Méthode d'élimination des états** : On considère un automate A comme dans le point précédent. Tant que A possède au moins 3 états, on choisit un état $q \notin \{q_i, q_f\}$ et on supprime q en modifiant les transitions :



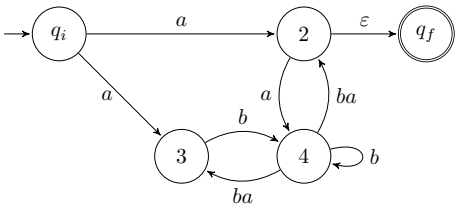
Exemple :



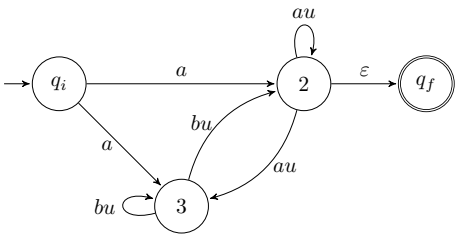
1. On commence par se ramener à un automate avec un état initial sans transition entrante et un état final sans transition sortante :



2. Suppression de l'état 1 :

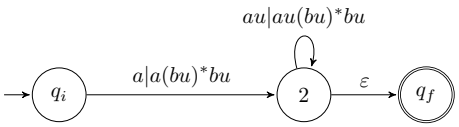


3. Suppression de l'état 4 :

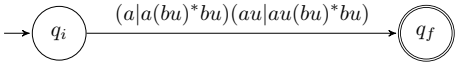


Avec $u = b^*ba$.

4. Suppression de l'état 3 :



5. Suppression de l'état 2 :



On obtient l'expression régulière $a|a(bu)^*bu(au|au(bu)^*bu)$, où $u = b^*ba$.

régulier $\Longleftrightarrow$ reconnaissable

- **Théorème de Kleene** : un langage est régulier si et seulement si il est reconnaissable par un automate.

- Les théorèmes sur les automates s'appliquent aussi aux langages réguliers, et inversement. Notamment, les langages réguliers sont stables par union, concaténation, étoile, intersection, complémentaire, différence.

- La logique propositionnelle définit la notion de formule vraie (si elle est vraie pour toute valuation).
  La déduction naturelle permet de formaliser la notion de preuve mathématique.
- Un **séquent** est noté $\Gamma \vdash A$ où Γ est un ensemble de formules logiques et A une formule logique.
  $\Gamma \vdash A$ signifie Intuitivement que sous les hypothèses Γ, on peut déduire A.
- Règles de déduction naturelle classique, où A, B, C sont des formules quelconques :

	Introduction	**Élimination**
Conjonction	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_e^g \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_e^d$
Disjonction	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_i^g \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_i^d$	$\dfrac{\Gamma, A \vdash C \quad \Gamma, B \vdash C \quad \Gamma \vdash A \vee B}{\Gamma \vdash C}\ \vee_e$
Implication	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i$	$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$
Négation	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg_e$
Vrai $\top$	$\dfrac{}{\Gamma \vdash \top}\ \top_i$	
Faux $\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e$

Axiome	**Affaiblissement**	**Réduction à l'absurde**
$\dfrac{}{\Gamma, A \vdash A}\ \text{ax}$	$\dfrac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{aff}$	$\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{raa}$

Il n'est pas nécessaire d'apprendre ces règles par coeur (elles seront rappelées), mais il faut les comprendre et savoir les utiliser.

- Une **preuve** d'un séquent $\Gamma \vdash A$ est un arbre dont les nœuds sont des séquents, les arcs des règles et la racine est $\Gamma \vdash A$.
  Exemples :

1. Preuve de $(A \wedge B) \to C \vdash A \to (B \to C)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \to C \vdash A \wedge B \to C}\ \text{ax} \quad \dfrac{\dfrac{}{A \vdash A}\ \text{ax} \quad \dfrac{}{B \vdash B}\ \text{ax}}{A, B \vdash A \wedge B}\ \wedge_i}{(A \wedge B) \to C, A, B \vdash C}\ \to_e}{(A \wedge B) \to C, A \vdash B \to C}\ \to_i}{(A \wedge B) \to C \vdash A \to (B \to C)}\ \to_i$$

2. Preuve de $A \vdash \neg\neg A$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \text{ax} \quad \dfrac{}{\neg A \vdash \neg A}\ \text{ax}}{A, \neg A \vdash \bot}\ \neg_e}{A \vdash \neg\neg A}\ \neg_i$$

3. On peut décomposer une preuve longue en plusieurs parties, pour plus de lisibilité. Par exemple pour prouver $\vdash A \vee (B \wedge C) \longrightarrow (A \vee B) \wedge (A \vee C)$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{ax}}{A \vdash A \vee B}\,\vee_i^g \qquad \dfrac{\dfrac{\dfrac{}{B \wedge C \vdash B \wedge C}\,\text{ax}}{B \wedge C \vdash B}\,\wedge_e^g}{B \wedge C \vdash A \vee B}\,\vee_i^d \qquad \dfrac{}{A \vee (B \wedge C) \vdash A \vee (B \wedge C)}\,\text{ax}}{A \vee (B \wedge C) \vdash A \vee B\ (*)}\,\vee_e$$

On montre de même $A \vee (B \wedge C) \vdash A \vee C$ $(**)$ et finalement :

$$\dfrac{\dfrac{\dfrac{}{A \vee (B \wedge C) \vdash A \vee B}\,* \qquad \dfrac{}{A \vee (B \wedge C) \vdash A \vee C}\,**}{A \vee (B \wedge C) \vdash (A \vee B) \wedge (A \vee C)}\,\wedge_i}{\vdash A \vee (B \wedge C) \longrightarrow (A \vee B) \wedge (A \vee C)}\,\longrightarrow_i$$

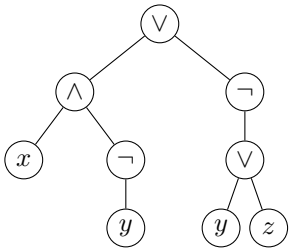
- (Correction de la déduction naturelle) Si $\Gamma \vdash A$ est prouvable alors $\Gamma \models A$.

> Preuve : Soit $P(h)$: « si T est un arbre de preuve de hauteur h pour $\Gamma \vdash A$ alors $\Gamma \models A$ ».
> $P(0)$ est vraie : Si T est un arbre de hauteur 0 pour $\Gamma \models A$ alors il est constitué uniquement d'une application de ax, ce qui signifie que $A \in \Gamma$ et implique $\Gamma \models A$.
> Soit T un arbre de preuve pour pour $\Gamma \vdash A$ de hauteur $h+1$. Considérons la règle appliquée à la racine de T.
>
> – $(\wedge_i)$ Supposons T de la forme : $\dfrac{\dfrac{T_1}{\Gamma \vdash A} \qquad \dfrac{T_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\,\wedge_i$
> Par hypothèse de récurrence sur T_1 et T_2, on obtient $\Gamma \models A$ et $\Gamma \models B$.
> Une valuation v satisfaisant toutes les formules de Γ satisfait donc à la fois A et B, et donc $A \wedge B$. On a bien $\Gamma \models A \wedge B$.
>
> – $(\wedge_e)$ Supposons T de la forme : $\dfrac{\dfrac{T_1}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A}\,(\wedge_e^g)$ Par récurrence sur T_1, $\Gamma \models A \wedge B$ et donc $\Gamma \models A \wedge B$.
>
> – Les autres cas sont similaires...

- Soit V un ensemble (de **variables**). L'ensemble des **formules logiques** sur V est défini inductivement :
  - T et F sont des formules (Vrai et Faux)
  - Toute variable $x \in V$ est une formule
  - Si φ est une formule alors $\neg\varphi$ est une formule
  - Si φ, ψ sont des formules alors $\varphi \wedge \psi$ (conjonction) et $\varphi \vee \psi$ (disjonction) sont des formules

```
type 'a formula =
    | T | F (* true, false *)
    | Var of 'a (* variable *)
    | Not of 'a formula
    | And of 'a formula * 'a formula
    | Or of 'a formula * 'a formula
```

- On peut représenter une formule logique par un arbre.
  Exemple : $(x \wedge \neg y) \vee \neg(y \vee z)$ est représenté par



  - L'**arité** d'un connecteur logique est son nombre d'arguments (= nombre de fils dans l'arbre).
    $\neg$ est d'arité 1 (unaire) et $\wedge, \vee$ sont d'arités 2 (binaire).
  - La **taille** d'une formule est le nombre de symboles qu'elle contient (= nombre de noeuds de l'arbre).
  - La **hauteur** d'une formule est la hauteur de l'arbre associé.

- (Exemple de démonstration par induction sur les formules) Soit φ une formule ayant $n(\varphi)$ symboles de négation et $b(\varphi)$ connecteurs binaire. Alors la taille $t(\varphi)$ de φ est : $t(\varphi) = 1 + n(\varphi) + 2b(\varphi)$.

  Preuve : Montrons $P(\varphi) : t(\varphi) = 1 + n(\varphi) + 2b(\varphi)$ par induction.
  Cas de base : $t(T) = 1 = 1 + 0 + 0$ donc $P(T)$ est vraie.
  De même pour $P(F)$ et $P(x)$ où x est une variable.
  Hérédité : Soit φ une formule.
  - Si $\varphi = \neg\psi$ alors $t(\psi) = 1 + n(\psi) + 2b(\psi)$ par induction et $t(\varphi) = t(\neg\psi) = 1 + t(\psi) = 1 + \underbrace{1 + n(\psi)}_{n(\varphi)} + 2\underbrace{b(\psi)}_{b(\varphi)} =$ $1 + n(\varphi) + 2b(\varphi)$ donc $P(\varphi)$ est vraie.
  - Si $\varphi = \psi_1 \wedge \psi_2$ alors, par induction, $t(\psi_1) = 1 + n(\psi_1) + 2b(\psi_1)$ et $t(\psi_2) = 1 + n(\psi_2) + 2b(\psi_2)$. Donc $t(\varphi) = 1 + t(\psi_1) + t(\psi_2) = 1 + \underbrace{n(\psi_1) + n(\psi_2)}_{n(\varphi)} + 2\underbrace{(1 + b(\psi_1) + b(\psi_2))}_{b(\varphi)}$.
    Donc $P(\varphi)$ est vraie.
  - De même si $\varphi = \psi_1 \vee \psi_2$.

  Par induction structurelle, $P(\varphi)$ est donc vraie pour toute formule φ.

- $\varphi \longrightarrow \psi$ est défini par $\neg\varphi \vee \psi$.
  $\varphi \longleftrightarrow \psi$ est défini par $\varphi \longrightarrow \psi \wedge \psi \longrightarrow \varphi$.

- Une **valuation** sur un ensemble V de variables est une fonction $v : V \longrightarrow \{0, 1\}$. 0 est aussi noté Faux ou $\bot$. 1 est aussi noté Vrai ou $\top$. L'**évaluation** $[\![\varphi]\!]_v$ d'une formule φ sur v est définie inductivement :
  - $[\![T]\!]_v = 1$, $[\![F]\!]_v = 0$
  - $[\![x]\!]_v = v(x)$ si $x \in V$
  - $[\![\neg\varphi]\!]_v = 1 - [\![\varphi]\!]_v$
  - $[\![\varphi \wedge \psi]\!]_v = \min([\![\varphi]\!]_v, [\![\psi]\!]_v)$
  - $[\![\varphi \vee \psi]\!]_v = \max([\![\varphi]\!]_v, [\![\psi]\!]_v)$

  Si $[\![\varphi]\!]_v = 1$, on dit que v est un **modèle** pour φ.

```
let rec eval d = function
    | T -> true
    | F -> false
    | Var(x) -> d x
    | Not(p) -> not (eval p)
    | And(p, q) -> (eval p) && (eval q)
    | Or(p, q) -> (eval p) || (eval q)
```

- Une formule toujours évaluée à 1 est une **tautologie**. Une formule toujours évaluée à 0 est une **antilogie**. Une formule qui possède au moins une évaluation à 1 est **satisfiable**.

- Deux formules φ et ψ sur V sont **équivalentes** (et on note $\varphi \equiv \psi$) si, pour toute valuation $v : V \to \{0, 1\}$: $[\![\varphi]\!]_v = [\![\psi]\!]_v$.
  - $\neg\neg\varphi \equiv \varphi$
  - $\varphi \vee \neg\varphi \equiv T$ (toujours vrai)
  - $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ (de Morgan)
  - $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ (de Morgan)
  - $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
  - $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

- La table de vérité permet de voir rapidement quelles sont les évaluations possibles d'une formule. Une formule à n variables possède 2^n évaluations possibles, et donc 2^n lignes dans sa table de vérité.

x	y	$(x \wedge y) \vee (\neg x \wedge \neg y)$
0	0	1
0	1	0
1	0	0
1	1	1

Table de vérité de
$(x \wedge y) \vee (\neg x \wedge \neg y)$