Logique propositionnelle, forme normale conjonctive, résolution propositionnelle : théorème de Robinson (1965)

Antoine MOTEAU
antoine.moteau@wanadoo.fr

On se propose d'établir, à partir de sa représentation arborescente, si une formule de la logique propositionnelle est ou non une tautologie.

On sait que la logique propositionnelle finie est décidable ; on peut, par exemple, examiner les tables de vérité des propositions. On se propose ici d'examiner un autre algorithme de décision, la résolution propositionnelle, pour (tenter de) décider si une formule propositionnelle est ou non une tautologie.

Table des matières

1 Logique propositionnelle

Soit $\mathscr{T}$ un ensemble fini de symboles de *propositions atomiques* (ou atomes ou variables propositionnelles).
On considère l'ensemble $\mathscr{A}(\mathscr{T})$ des *formules propositionnelles* (ou formules ou propositions) définies sur l'ensemble $\mathscr{T}$ comme étant le plus petit ensemble tel que (avec les notations usuelles) :

- les constantes `Vrai` et `Faux` sont des formules propositionnelles,
- si p est un symbole de proposition atomique (atome) alors p est une formule propositionnelle,
- si A est une une formule propositionnelle alors $\neg A$ est une formule propositionnelle,
- si A et B sont des formules propositionnelles, alors $(A \wedge B)$, $(A \vee Q)$ et $(A \Rightarrow B)$, sont des formules propositionnelles,

où ces règles sont appliquées un nombre fini de fois.

Une formule propositionnelle définie sur $\mathscr{T}$ sera représentée par (assimilée à) un arbre dont les nœuds sont les connecteurs logiques ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) et les feuilles des symboles de proposition atomiques ou les constantes `Vrai`, `Faux`. La *taille* de la formule est le nombre de nœuds de l'arbre et sa *hauteur* est la hauteur de l'arbre.

On appelle *littéral* un symbole de proposition atomique p (littéral *positif*), la négation d'un symbole de proposition atomique $\neg p$ (littéral *négatif*), `Faux` ou `¬Faux` (`Vrai` n'est pas un littéral).

Une *valuation* (ou *interprétation*) est une fonction $v : \mathscr{T} \longrightarrow \{\texttt{V},\texttt{F}\}$ (ou $\{1,0\}$), qui assigne une *valeur de vérité* à chaque symbole de proposition atomique. Cette valuation s'étend en une unique fonction $v : \mathscr{A}(\mathscr{T}) \longrightarrow \{\texttt{V},\texttt{F}\}$ (ou $\{1,0\}$), selon les conventions et règles de calcul usuelles : $v(\texttt{Vrai}) = \texttt{V}$ (ou 1) , $v(\texttt{Faux}) = \texttt{F}$ (ou 0), $v(A \wedge B) = v(A) \wedge v(B)$ (ou $v(A) \times v(B)$), etc ...

Une formule propositionnelle est

- *satisfiable* s'il existe une valuation pour laquelle elle prend la valeur `V`,
- *falsifiable* s'il existe une valuation pour laquelle elle prend la valeur `F`,
- *contingente* si elle est à la fois satisfiable et falsifiable,
- une *tautologie* si elle prend la valeur `V` pour toute valuation,
- une *contradiction* (ou *antilogie*) si elle prend la valeur `F` pour toute valuation.

Remarque. Ici, on ajoute le connecteur $\Leftrightarrow$ et les littéraux `Vrai`, `¬Vrai` (en réduisant `Vrai` à `¬Faux` et `¬Vrai` à `Faux`).

1.1 Formules représentées par une chaîne de caractères

Une formule peut être écrite comme une chaîne de caractères :

- ELTP : Expression Logique Totalement Parenthésée ou quasi Totalement Parenthésée, sans règles de priorité.
  Par exemple `"(((e) Et (t)) => ((a) Ou (Non (b))))"` est une ELTP
  et `"((e Et t) => (a Ou Non b))"` est une formule quasi ELTP (c'est plus léger).
- ELNTP : Expression Logique Non Totalement Parenthésée, qui utilise des règles de priorité ("usuelles" ?).
  Par exemple, `"e Ou t Ou u Et v => a Ou Non b"; "a => b => c"` seraient interprétées comme les quasi-ELTP `"((e Ou (t Ou (u Et v))) => (a Ou Non b))"; "(a => (b =>c))"`.

1.1.1 Formules représentées par des chaînes de caractères quasi ELTP

Les formules ELQTP (Expression Logique Quasi Totalement Parenthésée) peuvent s'écrire

- **avec ou sans parenthèses autour d'une variable a, d'un Non a** :
  `"a", "(a)", "Non a", "(Non a)", "Non ((a)+b)", "(Non((a)+b))"` sont acceptées,
- **avec parenthèses obligatoires autour d'une sous-expression binaire** `g op d` :
  `"(a+b)", "(a + (b+c))", "Non(a+b)` sont acceptées ; `"a+b", "(a+b+c)", "Non a + b"` sont rejetées.
- **avec ou sans parenthèses superflues** : `"((( a )))", ((a+b))", "((( ((a))+b )))"` sont acceptées.

En se réduisant à des **noms de variables constitués d'une seule lettre minuscule a ... z, non accentuée**, les formules ELQTP (quasi complètement parenthésées), seront conformes à la grammaire :

$$A ::= \underbrace{a \mid \ldots \mid z}_{\texttt{variable}} \mid \underbrace{0}_{\texttt{faux}} \mid \underbrace{1}_{\texttt{vrai}} \mid \underbrace{\neg A}_{\substack{\texttt{Non } A \\ \textit{avec N} \\ \sim A}} \mid \underbrace{(A \wedge A)}_{\substack{(A\ \texttt{Et}\ A) \\ (A\ \texttt{et}\ A) \\ (A \cdot A),(A * A)}} \mid \underbrace{(A \vee A)}_{\substack{(A\ \texttt{Ou}\ A) \\ (A\ \texttt{ou}\ A) \\ (A + A)}} \mid \underbrace{(A \Rightarrow A)}_{(A\ \texttt{=>}\ A)} \mid \underbrace{(A \Leftrightarrow A)}_{(A\ \texttt{<=>}\ A)} \mid (A)$$

les espaces entre constituants étant non significatifs :

`"(Non((aEtNonb)=>a)=>b)" , "  (~((a.~b)=>  a  ) =>b )  "` sont acceptées,
`"N on a" , "(Non((a Et Non b) => a) = >b)" , "(~((a.~b)=>a)= >b)"` sont refusées.

1.1.2 Formules représentées par des chaînes de caractères ELNTP

Les formules ELNTP (Expression Logique Non Totalement Parenthésée) peuvent s'écrire

- **avec ou sans parenthèses autour d'une variable a, d'une constante a, d'un Non a** :
  `"a"`, `"(a)"`, `"Non a"`, `"(Non a)"`, `"Non ((a)+b)"`, `"(Non((a)+b))"` sont acceptées,
- **avec ou sans parenthèses autour d'une sous-expression binaire** `g op d` :
  `"(a+b)"`, `"(a + (b+c))"`, `"Non(a+b)"`,`"a+b"`, `"a+b+c"`, `"Non a + b` sont acceptées.
- **avec ou sans parenthèses superflues** : `"((( a )))"`, `((a+b))"`, `"((( ((a))+b )))"` sont acceptées.

et utilisent les règles de priorités usuelles[a] , avec associativité systématique à droite[b] :

- Vrai, Faux, Variable : 6 ; Non : 5 ; Et : 2 ; Ou : 1 ; <=>, => : 0

  Par exemple, formules ELNTP et leurs équivalents quasi-ELTP :

  associativité à droite : `"a*b*c"` ≡ `"(a*(b*c))"` ; `"a+b+c"` ≡ `"(a+(b+c))"` ;
  `"a=>b=>c"` ≡ `"(a=>(b=>c))"` ; `"a<=>b<=>c"` ≡ `"(a<=>(b<=>c))"`

  règles de priorité : `"Non a * b"` ≡ `"(Non a * b)"` ; `"a * b + c"` ≡ `"((a * b) + c)"` ;
  `"a + b => c"` ≡ `"((a + b) => c)"` ; `"a => b + c"` ≡ `"(a => (b + c))"` ;
  `"a => b <=> c"` ≡ `"(a => (b <=> c))"` ...

---

a. celles qui sont utilisées le plus souvent, mais cela varie selon les auteurs ...
b. => est non associatif : `"(a=>b)=>c "`≢ `"a=>(b=>c)"` et on choisit l'associativité à droite : `"a=>b=>c"` ≡ `"a=>(b=>c)"`
Et, Ou, <=> sont associatifs ; `"(a*b)*c"` ≡ `"a*(b*c)"` ; `"(a+b)+c"` ≡ `"a+(b+c)"` ; `"(a<=>b)<=>c"` ≡ `"a<=>(b<=>c)"`,
et, par souci d'homogénéité, on impose l'associativité à droite.

Les formules ELNTP (non totalement parenthésées), seront conformes à la grammaire :

$$A ::= \underbrace{\texttt{nom}}_{\text{variable sauf noms réservés}} \mid \underbrace{0}_{\texttt{Faux}} \mid \underbrace{1}_{\texttt{Vrai}} \mid \underbrace{\neg A}_{\texttt{Non } A,\ \texttt{non } A,\ \sim A} \mid \underbrace{A \wedge A}_{A \texttt{ Et } A,\ A \texttt{ et } A,\ A \cdot A, A * A} \mid \underbrace{A \vee A}_{A \texttt{ Ou } A,\ A \texttt{ ou } A,\ A+A} \mid \underbrace{A \Rightarrow A}_{A \texttt{ => } A,\ A \texttt{ Imp } A,\ A \texttt{ imp } A} \mid \underbrace{A \Leftrightarrow A}_{A \texttt{ <=> } A,\ A \texttt{ Equ } A,\ A \texttt{ equ } A} \mid (A)$$

$$\texttt{nom} ::= \texttt{lettre} \mid \texttt{lettre suite}$$

$$\texttt{suite} ::= \texttt{lettre} \mid \texttt{chiffre} \mid \texttt{lettre suite} \mid \texttt{chiffre suite}$$

$$\texttt{lettre} ::= \texttt{a} \mid \dots \mid \texttt{z} \mid \texttt{A} \mid \dots \mid \texttt{Z} \quad ; \quad \texttt{chiffre} ::= 0 \mid \dots \mid 9$$

Ici, **les noms de variables sont composés de lettres, minuscules a ... z, ou majuscules A ... Z, non accentuées, de chiffres** 0...9 **et doivent obligatoirement commencer par une lettre.**
Les noms, en minuscules et/ou en majuscules, "Vrai", "Faux", "Non", "Et", "Ou", "Imp", "Equ" sont réservés.

Les espaces entre constituants peuvent être significatifs :

`"  (  Non((aEtNonb)     =>a   )=>       b)   "` est acceptée, mais `"aEtNonb"` est une variable !
`"  (  Non((a Et non b)    =>a   )=>       b)   "` est acceptée, avec le sens `"(~((a*(~b))=>a)=>b)"`
`"   (~   ((a    .~b)=>       a   )   =>b )   "` est acceptée,
`"N on a"` , `"(Non((a Et Non b) => a) = >b)"` , `"(~((a.~b)=>a)= >b)"`, `"ba ba"` sont refusées,
`"baba"`, `Nona"`, `"aEtNonb"` sont acceptées comme des variables.

Lors de la construction d'une chaîne ELNTP depuis un arbre formule, les parenthèses seront

- réduites, selon les règles de priorité usuelles, avec certains opérateurs : `"a+b+c"`, `"a+b*c"`, `"a=>b+c"`, ..
- explicites avec => ou <=> en cascade : `"a=>(b=>c)"`, `"a<=>(b=>c)"`, ..

1.1.3 ELQuasiTP ou ELNonTP ?

Avec la représentation ELQuasiTP,

- la conversion chaîne ELQuasiTP vers arbre formule est (relativement) facile à programmer,
- la conversion arbre formule vers chaîne ELQuasiTP est facile à programmer (peut être laissée à titre d'exercice).

Avec la représentation ELNonTP,

- la conversion d'une chaîne ELQuasiTP vers un arbre formule est complexe à programmer,
- la conversion d'un arbre formule vers un chaîne ELNonTP est complexe à programmer.

L'écriture d'une forme normale conjonctive sous forme de chaîne est plus lourde en ELQuasiTP qu'en ELNonTP :
L'ELNonTP `(a+b+c+d+e)*(x+y+z+t+u)` s'écrit `((a+(b+(c+(d+a))))*(x+(y+(z+(t+a)))))` comme ELQuasiTP.

On choisit ici la représentation par chaine ELNonTP,
avec des fonctions de conversion (chaîne vers arbre, arbre vers chaîne) pré-programmées.

2 Environnement de programmation (OCaml)

2.1 Eléments utiles

2.1.1 Fonctions usuelles de OCaml

`List.mem : 'a -> 'a list -> bool`

`mem a l` is true if and only if a is structurally equal (see module eq) to an element of l.

```
List.mem 6 [1;2;3;4;5;6;7;5];;  (* true *).
```

`List.exists : ('a -> bool) -> 'a list -> bool.`

`exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an). *Rem :* exists p [] is false.

2.1.2 Fonctions complémentaires

`tri_uniq_nat : 'a list -> 'a list`

`tri_uniq_nat u` sort list u in increasing order according to natural comparison function, with removing duplicates.

```
let tri_uniq_nat w  = List.sort_uniq compare w;; (* tri, avec unicité, selon l'ordre "naturel" *)
tri_uniq_nat ["z"; "a"; "z"; "w"; "d"; "b"; "z"; "c"; "a"];; (* ["a"; "b"; "c"; "d"; "w"; "z"] *)!
```

▶ `except : 'a -> 'a list -> 'a list` A ECRIRE

`except a l` returns the list l where the first element structurally equal to a has been removed. The list l is returned unchanged if it does not contain a. `except 5 [1;2;3;4;5;6;7;5];; (* [1; 2; 3; 4; 6; 7; 5] *)`

▶ `except_all : 'a -> 'a list -> 'a list` A ECRIRE

`except_all a l` returns the list l where all elements structurally equal to a have been removed. The list l is returned unchanged if it does not contain a. `except_all 5 [1;2;3;4;5;6;7;5];; (* [1; 2; 3; 4; 6; 7] *)`

2.2 Arbre formule propositionnelle

On utilise une représentation arborescente pour décrire les formules :

```
type formule =      (* arbre formule *)
    Var of string   (* nom de variable, ici réduit à 1 caractère, de a à z *)
  | Vrai | Faux
  | Non of formule
  | Et of formule * formule  | Ou of formule * formule
  | Imp of formule * formule | Equ of formule * formule
;;
```

2.3 Interface (construction, lecture d'un arbre formule)

2.3.1 Arbre formule déduit d'une chaîne ELNonTP

`formule_of_elnontp : string -> formule`

formule_of_elnontp s renvoie la formule construite à partir de l'expression ELNonTP contenue dans la chaîne de caractères s. Provoque les exceptions Stream.Failure ou Stream.Error en cas d'erreur d'analyse du flux (formule vide, contenant des caractères non reconnus, incorrecte, mal parenthésée).

2.3.2 Chaîne ELNonTP déduite d'un arbre formule

`elnontp_of_formule: formule -> string`

elnontp_of_formule p renvoie la chaîne ELNonTP de la formule représentée par l'arbre p.

2.3.3 Chaîne ELQuasiTP déduite d'un arbre formule

▶ `elquasitp_of_formule: formule -> string` A ECRIRE

elquasitp_of_formule p renvoie la chaîne ELQuasiTP de la formule représentée par l'arbre p.

2.3.4 Représentation graphique d'un arbre formule

`dessine_formule : formule -> unit = <fun>`

dessine_formule p représente (tant bien que mal) l'arbre formule p (pas trop compliqué) sur la fenêtre graphique de Caml.

3 Forme normale conjonctive

3.1 Définitions

Définition 3.1. (Clauses et formes clausales)

Une *clause* est une disjonction [a] $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$ où les ℓ_i sont des littéraux.

Une *forme clausale* est une conjonction [b] $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ où les C_i sont des clauses.

Une *forme normale conjonctive* est une forme clausale où dans chacune des clauses les littéraux sont distincts.

a. Une clause vide est interprétée comme `Faux` ; à rapprocher de $0 = \sum_{i \in \emptyset} i$.

b. Une forme clausale vide est interprétée comme `Vrai` (ou `NonFaux`) ; à rapprocher de $1 = \prod_{i \in \emptyset} i$.

Bien que les clauses et formes clausales soient des formules propositionnelles, donc normalement représentables de façon arborescente, pour des raisons pratiques, **on représente ici les clauses et les formes clausales par des listes** :

```
type clause = formule list;;                (* où formule est réduit à un littéral *)
type forme_clausale = clause list;;
```

Par exemple, `"(a+c+x+y)*(a+d)*(b+c)*(b+d)"` sera représentée par la liste :
`x00 = [[Var "a"; Var "c"; Var "x"; Var "y"]; [Var "a"; Var "d"]; [Var "b"; Var "c"]; [Var "b"; Var "d"]]`

Remarque. D'un coté il y a la théorie et de l'autre la pratique :

La représentation, par liste de listes, d'une forme normale conjonctive pourrait nous amener à y admettre des clauses distinctes mais identiques, comportant exactement les mêmes littéraux, à l'ordre près.

3.2 Chaînes déduites d'une forme clausale

3.2.1 Chaîne ELQuasiTP déduite d'une forme clausale

▶ `elquasitp_of_fc : formule list list -> string` A ECRIRE

elquasitp_of_fc u renvoie la chaîne ELQuasiTP de la formule représentée par la forme clausale u.

```
elquasip_of_fc [];;    (* "1" *)
elquasitp_of_fc x00;;  (* "((a+(c+(x+y)))*((a+d)*((b+c)*(b+d))))" *)
```

3.2.2 Chaîne ELNonTP déduite d'une forme clausale

▶ `elnontp_of_fc : formule list list -> string` A ECRIRE

elnontp_of_fc u renvoie la chaîne ELNonTP de la formule représentée par la forme clausale u.

```
elnontp_of_fc [];;   (* "1" *)
elnontp_of_fc x00;   (* "(a+c+x+y)*(a+d)*(b+c)*(b+d)" *)
```

3.3 Formule (équilibrée) déduite d'une clause, d'une forme clausale

```
let rec partage = function
     []  -> [], []
   | [a] -> [a], []
   | a::b::q -> let u,v = partage q in a::u, b::v
;;
(* partage : 'a list -> 'a list * 'a list = <fun> *)
let rec formule_of_clause = function
  | []     -> Faux  (* failwith "erreur" *)
  | [a]    -> a
  | [g; d] -> Ou (g, d)
  | q      -> let g, d = partage q in Ou (formule_of_clause g, formule_of_clause d)
and     formule_of_forme_clausale = function
  | []     -> Non Faux   (* failwith "erreur" *)
  | [a]    -> formule_of_clause a
  | [g; d] -> Et (formule_of_clause g, formule_of_clause d)
  | q      -> let g, d = partage q in Et (formule_of_forme_clausale g, formule_of_forme_clausale d)
;;
(* formule_of_clause : formule list -> formule = <fun> *)
(* val formule_of_forme_clausale : formule list list -> formule = <fun> *)
  elquasitp_of_formule (formule_of_forme_clausale x00);; (* "((((a+x)+(c+y))*(b+c))*((a+d)*(b+d)))" *)
```

3.4 Unions

1. *Union disjonctive* de deux clauses $C_1 = \ell_1^1 \vee \ldots \vee \ell_1^n$ et $C_2 = \ell_2^1 \vee \ldots \vee \ell_2^m$:
   On note $C_1 \veebar C_2$ la disjonction $\ell_1^1 \vee \ldots \vee \ell_1^n \vee \ell_2^1 \vee \ldots \vee \ell_2^m$ dans laquelle on a supprimé toute répétition.
2. *Union conjonctive* de deux formes clausales F_1 et F_2 :
   On note $F_1 \barwedge F_2$ la conjonction des clauses de F_1 et F_2, parmi lesquelles on a supprimé toute répétition.
   Remarque. Mais, dans cette conjonction, pour des raison pratiques (voir encadré ci-dessus) on pourrait être amené à admettre des clauses comportant exactement les mêmes littéraux (mais dans un ordre différent) ....

Les clauses et les formes clausales étant des listes, les deux opérations $\veebar$ et $\barwedge$ sont réalisées par la même fonction `union` :

▶ `union : 'a list -> 'a list - 'a list` A ECRIRE

union l1 l2 return a list which contains all elements of lists l1 and l2, with no duplicates.

```
union ["a";"b";"a"] ["u";"v";"a";"u"];;      (* ["a"; "b"; "u"; "v"] *)
union [Var "a"; Var "b"; Var "a"] [Var "u"; Var "v"; Var "a"; Var "u"];;
   (* [Var "a"; Var "b"; Var "u"; Var "v"] *)

union [["a"; "b"]; ["x"; "y"; "z"]; ["u"; "v"; "u"; "w"]]  [["x"; "y"; "z"]; ["f"; "g";"h"]];;
   (* [["a"; "b"]; ["x"; "y"; "z"]; ["u"; "v"; "u"; "w"]; ["f"; "g"; "h"]] *)

union [["a"; "b"]; ["x"; "y"; "z"]; ["u"; "v"; "u"; "w"]]  [[["z"; "y"; "x"]; ["f"; "g";"h"]];;
   (* [["a"; "b"]; ["x"; "y"; "z"]; ["u"; "v"; "u"; "w"]; [["z"; "y"; "x"]; ["f"; "g"; "h"]] *)
```

Remarque. `["z"; "y"; "x"]` n'est pas égal à `["x"; "y"; "z"]`

▷ **Question** : Quelle est la complexité de `union` en fonction de la taille des arguments ?

3.5 Produit de formes clausales

Soit $A = A_1 \wedge \ldots \wedge A_n$ et $B = B_1 \wedge \ldots \wedge B_m$ deux formes clausales (les A_i et B_i étant des clauses).
le *produit* de A et B, noté $A \otimes B$, est la forme clausale définie par

$$A \otimes B = \barbigwedge_{i,j} A_i \veebar B_j$$

Le produit $\otimes$ sera réalisé par la fonction :

▶ `produit : 'a list list -> 'a list list -> 'a list list` A ECRIRE

Par exemple,

```
produit [["a"; "b"]; ["x"; "y"; "z"]; ["u"; "v"; "u"; "w"]] [["x"; "y"; "z"]; ["f"; "g"; "h"]];;
(* [["a"; "b"; "x"; "y"; "z"]; ["a"; "b"; "f"; "g"; "h"];
    ["x"; "y"; "z"]; ["x"; "y"; "z"; "f"; "g"; "h"];
    ["u"; "v"; "w"; "x"; "y"; "z"]; ["u"; "v"; "w"; "f"; "g"; "h"]] *)

produit [[Var "a"; Var "b"]; [Var "x"; Var "y"; Var "z"]; [Var "u"; Var "v"; Var "u"; Var "w"]]
        [[Var "x"; Var "y"; Var "z"]; [Var "f"; Var "g"; Var "h"]];;
(* [[Var "a"; Var "b"; Var "x"; Var "y"; Var "z"]; [Var "a"; Var "b"; Var "f"; Var "g"; Var "h"];
    [Var "x"; Var "y"; Var "z"]; [Var "x"; Var "y"; Var "z"; Var "f"; Var "g"; Var "h"];
    [Var "u"; Var "v"; Var "w"; Var "x"; Var "y"; Var "z"];
    [Var "u"; Var "v"; Var "w"; Var "f"; Var "g"; Var "h"]] *)
```

$$\begin{aligned}&\big((a+b)*(x+y+z)*(v+u+v+w)\big) \otimes \big((x+y+z)*(g+f+h)\big)\\ &= (a+b+x+y+z)*(a+b+g+f+h)*(x+y+z)*(x+y+z+f+g+h)\\ &\quad *(u+v+w+x+y+z)*(u+v+w+g+f+h)\end{aligned}$$

Remarque. la répétition de u et une répétition de $x+y+z$ ont été supprimées (lors d'un appel de `union`).

▷ **Question** : Quelle est la complexité de `produit` en fonction de la taille des formes clausales passées en arguments ?

3.6 Mise d'une formule sous forme normale conjonctive

Définition 3.2. (forme normale conjonctive d'une formule propositionnelle)

La *forme normale conjonctive* $\mathscr{C}(P)$ d'une formule propositionnelle P est définie ainsi :

Si P est un littéral, alors $\mathscr{C}(P) = P$.

$\mathscr{C}(\texttt{Vrai}) = \neg\texttt{Faux}$ et $\mathscr{C}(\neg\texttt{Vrai}) = \texttt{Faux}$	$\mathscr{C}(\neg(\neg A)) = \mathscr{C}(A)$
$\mathscr{C}(A \wedge B) = \mathscr{C}(A) \barwedge \mathscr{C}(B)$	$\mathscr{C}(\neg(A \wedge B)) = \mathscr{C}(\neg A) \otimes \mathscr{C}(\neg B)$
$\mathscr{C}(A \vee B) = \mathscr{C}(A) \otimes \mathscr{C}(B)$	$\mathscr{C}(\neg(A \vee B)) = \mathscr{C}(\neg A) \barwedge \mathscr{C}(\neg B)$
$\mathscr{C}(A \Rightarrow B) = \mathscr{C}(\neg A) \otimes \mathscr{C}(B)$	$\mathscr{C}(\neg(A \Rightarrow B)) = \mathscr{C}(A) \barwedge \mathscr{C}(\neg B)$
$\mathscr{C}(A \Leftrightarrow B) = \cdots$	$\mathscr{C}(\neg(A \Leftrightarrow B)) = \cdots$

▷ **Question** : Justifier la terminaison de cette définition et prouver que $\mathscr{C}(P)$ est une forme normale conjonctive.

Proposition 3.1.

Une proposition P et sa forme normale conjonctive, $\mathscr{C}(P)$, sont équivalentes (prennent les mêmes valeurs de vérité pour les mêmes valuations).

En particulier, P est une tautologie si et seulement si $\mathscr{C}(P)$ est une tautologie.

Le calcul de la forme normale conjonctive, $\mathscr{C}(P)$, d'une proposition P sera réalisé par la fonction :

▶
```
val fnc_of_formule : formule -> formule list list = <fun>                    A ECRIRE
```

```
elnontp_of_fc ( fnc_of_formule (formule_of_elnontp "(a * b * a) + (c * d)") );;
(* (a+c)*(a+d)*(b+c)*(b+d) *)
```

▷ **Question** : Calculer la forme normale conjonctive des propositions $\big((a \wedge b) \Rightarrow a\big)$ et $\big(a \Rightarrow (a \wedge b)\big)$.

3.7 Standardisation (réduction) des clauses et des formes clausales

3.7.1 Standardisation (réduction) des clauses

La réduction des séquences `"X+a+Y+~a+Z"` à `"Non Faux"`, `"X+Faux+Y"` à `"X+Y"`, `"X+Vrai+Y"` à `"Non Faux"`, ..., la suppression des répétitions, par tri unique selon l'ordre naturel, dans chaque clause, devrait permettre d'éviter la répétition de clauses identiques à l'ordre près, dans les formes clausales (au prix de coûts temporels supplémentaires ...).

```
let reduce_clause c =
  let rec reduit = function
     | [] -> []
     | (Var a) ::q -> if List.mem (Non (Var a)) q then [Non Faux] else (Var a)::(reduit q)
     | (Non (Var a))::q -> if List.mem (Var a) q then [Non Faux] else (Non (Var a))::(reduit q)

     | Faux::q       -> reduit q
     | (Non Faux)::q -> [Non Faux]
     | Vrai::q       -> [Non Faux]
     | (Non Vrai)::q -> reduit q

     |_ -> failwith "Not a clause"   (* h::q -> h::(reduit q) *)
  in
  let w = reduit c in
  if ( List.mem (Non Faux) w ) then [Non Faux]
  else if w = [] then [Faux] else w
;;
(* val reduce_clause : formule list -> formule list = <fun> *)

let standardize_clause c = tri_uniq_nat @@ reduce c ;;
(* val standardize_clause : formule list -> formule list = <fun> *)

standardize_clause [];;          (* [Faux] *)

let c = [Non (Var "x"); Var "v"; Var "x"; Var "r"; Non (Var "x"); Var "y"; Var "z"; Var "u"];;
let s = elnontp_of_fc [c];;  (* "~x+v+x+r+~x+y+z+u" *)
standardize_clause c;;       (* [Non Faux] *)

let c = [Var "x";Var "y";Var "z";Faux;Var "x";Var "b";Var "y";Non (Var "c");Var "a";Var "z"];;
let s = elnontp_of_fc [c];;                      (* "x+y+z+0+x+b+y+~c+a+z" *)
let ss = elnontp_of_fc [standardize_clause c];;  (* "a+b+x+y+z+~c" *)
```

3.7.2 Standardisation (réduction) des formes clausales

Après standardisation des clauses, suivi d'un tri par longueur croissante des clauses,

- réduction des séquences "X*Vrai*Y" à "X*Y", "X*Faux*Y" à "Faux", ...,
- suppression des clauses contenant une autre clause (avec les clauses ordonnées par longueur croissante),
- tri unique selon l'ordre naturel.

Remarque. Cela a un coût temporel ...

```
let tri_length w  =
  List.sort (fun a b -> compare (List.length a) (List.length b)) w
;;
(*   val tri_length : 'a list list -> 'a list list = <fun *)
tri_length [["b";"c";"d";"e"]; ["a"]; ["x";"y"]];; (* [["a"]; ["x"; "y"]; ["b"; "c"; "d"; "e"]] *)

let rec isinclude c1 c2 =
  match c1 with
    |[] -> true
    | a::q -> (List.mem a c2) && (isinclude q c2)
;;
(* val isinclude : 'a list -> 'a list -> bool = <fun> *)
isinclude ["a"; "u"] ["a"; "c"; "b"];;        (* false *)
isinclude ["b"; "a"] ["a"; "c"; "u"; "b"];;   (* true *)

let reduce_fc f =
  let rec supcontains c = function
    |[] -> []
    |h::q -> if isinclude c h then (supcontains c q) else (h::(supcontains c q))
  in
  let rec reduit = function
    | [] -> []

    | [Vrai]::q     -> reduit q
    | [Non Faux]::q -> reduit q
    | [Faux]::q     -> [[Faux]]
    | [Non Vrai]::q -> [[Faux]]

    | h::q -> h::(reduit (supcontains h q))
  in
  let w = reduit (tri_length (List.map standardize_clause f)) in
  if (List.mem [Faux] w) || (List.mem [Non Vrai] w) then [[Faux]]
  else if w = [] then [[Non Faux]] else w
;;
(*  val reduce_fc : formule list list -> formule list list = <fun> *)

let standardize_formeclausale c = tri_uniq_nat @@ reduce_fc c;;
(* val standardize_formeclausale : formule list list -> formule list list =  <fun> *)
standardize_formeclausale [];;          (* [Non Faux] *)

let s = "(a*b*c)+(a => c+b) + (x+a+b)*(c+y)";;
let f = formule_of_elnontp s;;
let g = fnc_of_formule f;;
elnontp_of_fc g;; (* "(a+~a+c+b+x)*(a+~a+c+b+y)*(b+~a+c+x+a)*(b+~a+c+y)*(c+~a+b+x+a)*(c+~a+b+y)" *)
let h = standardize_formeclausale g;;
elnontp_of_fc h;; (* "b+c+y+~a" *)
```

On vérifiera que "((a*b*c) + (a => c+b) + (x+a+b)*(c+y)) <=> (b+c+y+~a)" est une tautologie.

4 Résolution propositionnelle

4.1 Règle de résolution

Définition 4.1.

À deux clauses $A = \alpha_1 \vee \ldots \vee \alpha_p \vee \ \ell \ \vee \alpha_{p+1} \vee \ldots \vee \alpha_n$ [a]
et $B = \beta_1 \vee \ldots \vee \beta_q \vee \neg\ell \ \vee \beta_{q+1} \vee \ldots \vee \beta_m$
la *règle de résolution* (propositionnelle) associe la clause

$$R = \left(\alpha_1 \vee \ldots \vee \alpha_p \vee \alpha_{p+1} \vee \ldots \vee \alpha_n\right) \veebar \left(\beta_1 \vee \ldots \vee \beta_q \vee \beta_{q+1} \vee \ldots \vee \beta_m\right)$$

(lorsque cette clause est vide on pose $R = \texttt{Faux}$).

On dit que R est une résolution de A et B.

a. Le littéral ℓ est un atome ou la négation d'un atome.

La règle de résolution peut s'appliquer ou ne pas s'appliquer à deux clauses A et B et, le cas échéant, elle peut s'appliquer de plusieurs façons.

La *résolution* de deux clauses A et B est la liste des clauses que l'on peut obtenir à partir de A et B par résolution.

Remarque. La résolution de A et B est la liste vide [a] si la règle de résolution ne peut pas s'appliquer.

a. Une forme clausale vide est assimilée à `Vrai` (`Non Faux`).

En réduisant les résolutions de A et B, par suppression de toutes les répétitions et remplacement de toutes les résolutions contenant un p et un $\neg p$ par `Vrai`, on obtient une seule clause, nommée *résolvante* de A et B.

Remarque. La résolvante de A et B est `Non Faux` [a] si la règle de résolution ne peut pas s'appliquer.

a. Cela ne peut pas être la clause vide, assimilée à `Faux`, donc `Non Faux` est judicieux.

La règle de résolution généralise la règle du modus ponens : $\begin{cases} \{p\ ,\ (p \Rightarrow q)\} \models q \\ \dfrac{p\ ,\ (p \Rightarrow q)}{q} \end{cases}$ (de p et de $p \Rightarrow q$, on déduit q).

exprimée ici sous la forme $\{X \vee A, \neg X \vee B\} \models A \vee B$ ou $\dfrac{X \vee A, \neg X \vee B}{A \vee B}$ (de $\{X \vee A$ et de $\neg X \vee B$, on déduit $A \vee B$).

De manière informelle, si on a X, alors on n'a pas $\neg X$, donc ayant $\neg X \vee B$, on a nécessairement B, donc $A \vee B$. De même, si on a $\neg X$, on n'a pas X, donc ayant $X \vee A$, on a nécessairement A, donc $A \vee B$. Dans les deux cas on a $A \vee B$.

Par exemple, résolution (sans réduction des clauses) en bleu, résolvante (avec réduction) en rouge :

- $A = x$ et $B = \neg x \longrightarrow [[\texttt{Faux}]]$ ou `Faux`
- $A = p \vee \neg q$ et $B = q \vee r \vee p \vee a \longrightarrow [p \vee r \vee p \vee a]$ ou $a \vee p \vee r$
- $A = p \vee \neg q$ et $B = q \vee r \vee \neg p \longrightarrow [p \vee r \vee \neg p\ ;\ \neg q \vee q \vee r]$ ou `Non Faux` (soit `Vrai`)
- $A = p \vee q$ et $B = q \vee r \longrightarrow$ liste vide $[]$ ou `Non Faux` **la règle ne s'applique pas !**
- $A = p \vee \neg q \vee \neg s \vee t$ et $B = q \vee r \vee s \vee \neg t \longrightarrow$
  $[p \vee \neg q \vee \neg s \vee q \vee r \vee s\ ;\ r \vee s \vee \neg t \vee p \vee \neg s \vee t\ ;\ q \vee r \vee \neg t \vee p \vee \neg q \vee t]$ ou `Non Faux` (soit `Vrai`)
- $A = p \vee \neg q \vee \neg s \vee t$ et $B = q \vee r \vee \neg p \vee s \vee \neg t \longrightarrow$
  $[\neg q \vee \neg s \vee t \vee q \vee r \vee s \vee \neg t\ ;\ p \vee \neg q \vee \neg s \vee q \vee r \vee \neg p \vee s\ ;\ r \vee \neg p \vee s \vee \neg t \vee p \vee \neg s \vee t\ ;\ q \vee r \vee \neg p \vee \neg t \vee p \vee \neg q \vee t]$
  ou `Non Faux` (soit `Vrai`)

▶ `val resolutions : formule list -> formule list -> formule list list = <fun>` A ECRIRE

resolutions A B renvoie la liste de toutes les résolutions possibles des deux clauses A et B.

▶ `val resolvante : formule list -> formule list -> formule list = <fun>` A ECRIRE

resolvante A B renvoie la résolvante des deux clauses A et B.

```
let a = [Var "p"; Non (Var "q")] and b = [Var "q"; Var "r"; Non (Var "p")];;
List.map (function u -> elnontp_of_clause u) (resolutions a b);; (* ["~q+q+r"; "r+~p+p"] *)
resolvante a b;; (* [Non Faux] *)

let a = [Var "p"; Var "q"] and b = [Var "q"; Var "r"];;
List.map (function u -> elnontp_of_clause u) (resolutions a b);;  (* [] *)
resolvante a b;;  (* [Non Faux] *)
```

4.2 Résolution des tautologies

Proposition 4.1. *(Robinson, 1965)*

> *Soit P une proposition et* $C_1 \wedge \ldots \wedge C_n$ *la forme normale conjonctive de* $\neg P$ (`Non P`).
> *P est une tautologie si et seulement si* `Faux` *est dérivable par résolution propositionnelle à partir des clauses* $C_1, \ldots, C_n$.

(les clauses étant prises deux à deux, dérivable signifiant "déduit par calcul itéré").

4.2.1 Test d'une forme normale conjonctive sous forme de liste de clauses

1. Ecrire la fonction :

▶ `paires : 'a list -> (('a * 'a) * 'a list) list = <fun>` A ECRIRE

paires u renvoie la liste de tous les couples $\big((x,y), u \setminus \{x,y\}$ où x et y sont des éléments distincts de u.

2. Processus de test par dérivation d'une liste de clauses u :
   (a) On calcule la liste $xyv =$ `paires u`, de tous les couples $\big((x,y), u \setminus \{x,y\}\big)$
   (b) Pour chaque élément $((x,y),v)$ de xyv,
   on calcule la resolvante r de x et y.
   - Si $r =$ `Faux`, alors on retourne true
   - Sinon, on recommence le processus de dérivation (qui est donc récursif), sur la liste $v \cup \{r\}$.

   Ce processus est réalisé par la fonction :

▶ `derive : formule list list -> bool = <fun>` A ECRIRE

▷ Questions :

- `derive` se termine-telle dans tous les cas ?
- Que se passe-t'il dans `derive u` si `u` n'est pas une tautologie ?
- Que penser de la complexité de `derive` ?
- Si `derive u` se termine, le résultat est-il juste ?
- Problème : x et y ne sont plus exploités dans $v \cup \{r\}$.

4.2.2 Test d'une proposition

▶ `tautologie : formule -> bool` A ECRIRE

tautologie P renvoie true si P est une tautologie, false sinon.

▷ Exemple : Etudier les propositions totalement parenthésées $\big((a \wedge b) \Rightarrow a\big)$ et $\big(a \Rightarrow (a \wedge b)\big)$.

Voir `http://www.grappa.univ-lille3.fr/%7Echampavere/Enseignement/0607/l2miashs/ia/logique.pdf`
Voir `http://lecomte.al.free.fr/ressources/M1_IHS/res1.pdf`

> D'après Wikipédia (Règle de résolution), extrait :
>
> La règle de résolution ou principe de résolution de Robinson est une règle d'inférence logique que l'on peut voir comme une généralisation du modus ponens ("de A et de $A \Rightarrow B$, on déduit B").
> Cette règle est principalement utilisée dans les systèmes de preuve automatiques, elle est à la base du langage de programmation logique Prolog.
>
> Stratégie d'application de la règle.
>
> Le principe de résolution étant complet, si l'ensemble de clauses considéré est inconsistant (insatisfiable) on arrive toujours à générer la clause vide. Par contre, le problème de la consistance (satisfaisabilité) n'étant pas décidable en logique des prédicats, il n'existe pas de méthode pour nous dire quelles résolutions effectuer et dans quel ordre pour arriver à la clause vide.
>
> On peut facilement trouver des exemples où l'on "s'enfonce" dans la génération d'une infinité de clauses sans jamais atteindre la clause vide, alors qu'on l'aurait obtenue en faisant les bons choix.
>
> Différentes stratégies ont été développées pour guider le processus. Le système Prolog se base, par exemple, sur l'ordre d'écriture des clauses et l'ordre des littéraux dans les clauses. D'autres systèmes, comme CyC utilisent une stratégie de coupure (en fonction des ressources consommées) pour éviter de générer des branches infinies.

4.3 Application : le Club Ecossais (variation, d'après ?)

Archibald McArthur, descendant commun des deux clans McPherson et McKinnon, souhaite fusionner en un seul club le club des descendants de McPherson et le club des descendants de McKinnon. Or les membres de ces clubs tiennent énormément à respecter scrupuleusement leurs traditions séculaires.

Le club McPherson obéit, depuis 1304, aux règles suivantes :

- tout membre non écossais porte des chaussettes rouges ;
- tout membre porte un kilt ou ne porte pas de chaussettes rouges ;
- les membres mariés ne sortent pas le dimanche.

Le club McKinnon respecte la charte de 1431 :

- un membre sort le dimanche si et seulement s'il est écossais ;
- tout membre qui porte un kilt est écossais et marié ;
- tout membre écossais porte un kilt.

Archibald McArthur propose de définir les règles du nouveau club comme la totalité des règles des anciens clubs. Son cousin Murdoch MacArthur soutient qu'alors ce club sera si fermé qu'il ne pourra accepter personne.

- Donner une proposition P telle que P est vraie si et seulement si aucun membre ne peut être accepté dans ce club.
- Appliquer la fonction `tautologie` ci-dessus à la proposition P pour montrer qu'aucun membre ne peut être accepté dans ce club.

4.4 Exemple (formule contigente)

On considère les trois formules $\begin{cases} E_1 = \Big(b \Rightarrow \big((a \wedge \neg c) \vee (\neg a \wedge c)\big)\Big) \\ E_2 = \Big(\big(c \vee (\neg a \wedge \neg b)\big) \Rightarrow \big(c \wedge (a \vee b)\big)\Big) \\ E_3 = \Big(\big(a \Rightarrow c\big) \wedge \big(\neg a \Rightarrow \neg b\big)\Big) \end{cases}$

et la formule $R = \Big(\big(E_1 \wedge E_2 \wedge E_3\big) \vee \big(\neg E_1 \wedge \neg E_2 \wedge \neg E_3\big)\Big)$.

1. Représenter l'arbre de la formule R.
2. Calculer, sous forme de formule propositionnelle, R', forme normale conjonctive de R.
3. ▷ Quelle est la forme de l'arbre de R' ?
   Comment fallait-il faire pour avoir un arbre de hauteur minimale ?

Remarque. Mon implantation de la méthode de Robinson, avec une limite de 6 000 000 calculs de résolvantes, me donne

- R : 4, bool = false (n'est pas une tautologie)
- $R+R$: 4, bool = false (n'est pas une tautologie)
- $R*R$: 4, bool = false (n'est pas une tautologie)
- `Non` R : 979894, : bool = false (n'est pas une tautologie)
- $R => R$: 6000001, Exception : Failure "depasse".

5 Evaluation d'une formule

5.1 Fonction d'évaluation

Ecrire une fonction calculant la valeur, 0 (associée à `Faux`) ou 1 (associée à `vrai`), d'une formule propositionnelle, en fonction des valeurs (0 ou 1) attribuées au variables propositionnelles a, b, ... :

▶ `eval : int vect -> proposition -> int`

eval v p renvoie la valeur prise par p lorsque a prends la valeur $v.(0)$, b prends la valeur $v.(1)$ etc ...

Remarque. Le code ascii d'un caractère c s'obtient par `int_of_char c`, et les codes ascii des lettres (caractères) a ... z sont concécutifs.

5.2 Application (table de vérité)

On reprend l'exemple précédent, avec les les trois formules : $\begin{cases} E_1 = \Big(b \Rightarrow \big((a \wedge \neg c) \vee (\neg a \wedge c)\big)\Big) \\ E_2 = \Big(\big(c \vee (\neg a \wedge \neg b)\big) \Rightarrow \big(c \wedge (a \vee b)\big)\Big) \\ E_3 = \Big((a \Rightarrow c) \wedge (\neg a \Rightarrow \neg b)\Big) \end{cases}$

et la formule $R = \Big(\big(E_1 \wedge E_2 \wedge E_3\big) \vee \big(\neg E_1 \wedge \neg E_2 \wedge \neg E_3\big)\Big)$.

▷ A l'aide d'une séquence d'instructions Caml adéquate (on ne fait pas une fonction), calculer et éditer ligne après ligne, colonne après colonne la table de vérité de R .

On éditera, simplement, sans prétentions, mais de façon bien alignée, le contenu de cette table sous la forme habituelle, à l'aide de `printf` : `printf "%2u" i` édite l'entier i sur 2 places).

En déduire si R est ou non une tautologie.

Si R n'est pas une tautologie, est-elle satisfiable ?

Si R est satisfiable, préciser pour quelles valeurs des variables R prends la valeur `Vrai`.

6 Résolutions

On peut aussi s'amuser à dresser la liste de toutes les résolutions réduites obtenues à partir d'une proposition
On pourra s'inspirer de `derive_test`

Remarque. Cela peut devenir vite gros pour une formule P, surtout si $\neg P$ n'est pas une tautologie ... d'où très facilement ... "Out of mémory" !

Concrêtement, on sera amené à mettre une borne artificielle pour arrêter les calculs ... .

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ TP Robinson. Enoncé. (suite = préambule + squelette).

7 Préambule : éléments de programme pré-écrits

```
#open "graphics";; (* Caml : bibliothèque graphique *)
#open "printf";;   (* Caml : bibliothèque d'éditions formatées *)

(* === Types : proposition, clause, forme_clausale === *)
type proposition_atomique == char        (* nom de proposition atomique, ici de a à z *)
;;
type proposition =
    Atome of proposition_atomique
  | Vrai
  | Faux
  | Non of proposition
  | Et  of proposition * proposition
  | Ou  of proposition * proposition
  | Imp of proposition * proposition
;;
type clause == proposition list
;;
type forme_clausale == clause list
;;

let A = Atome `a` and B = Atome `b` in Imp (A, Imp (Imp (A,B), B));;

(*=====================================================================*)
(* === Construction d'une proposition à partir d'un flux de char === *)
(*=====================================================================*)

let rec saute_blancs s =
  match s with
     [< '( ` ` | `\t` ) >] -> saute_blancs s
   | [< 'c >]              -> [< 'c; saute_blancs s>]
   | [< >]                 -> s
;;
(* saute_blancs : char stream -> char stream = <fun> *)

let rec connecteur s =
  match s with
     [< '( `e` | `E` ); '`t` >]  -> (fun f1 f2 -> Et (f1,f2))   (* et, Et *)
   | [< '( `o` | `O` ); '`u` >]  -> (fun f1 f2 -> Ou (f1,f2))   (* ou, Ou *)
   | [< '`=`; '`>` >]            -> (fun f1 f2 -> Imp (f1,f2))
;;
(* connecteur : char stream -> proposition -> proposition -> proposition = <fun> *)

let rec formule s =
  match s with
     [< '`(`; formule f1; connecteur c; formule f2; '`)` >] -> c f1 f2
   | [< '`N`; '`o`; '`n`; formule f >]  -> Non f   (* N imposé: sinon pb avec "n ou a" *)
   | [< '`1` >]                         -> Vrai
   | [< '`0` >]                         -> Faux
   | [< '(`a`..`z` as p) >]             -> Atome p
;;
(* formule : char stream -> proposition = <fun> *)

let proposition_of_string s = construction (stream_of_string (s ^ "#"))
  where construction w =
    match (saute_blancs w) with
     [< formule f; '`#` >] -> f
;;
(* proposition_of_string : string -> proposition = <fun> *)

let s = "((a et b) => (a => (b ou c)))";;  proposition_of_string s;;
let s = "((a et b) => (n ou Non b))";;     proposition_of_string s;;
```

```
(* === Dessin (primitif) des (petits) arbres proposition === *)

let semix = fst (text_size "W") /2;;              (* demi largeur caractère *)
let semiy = snd (text_size "H") /2;;              (* demi hauteur caractère *)
let graph_dx = (15 * fst (text_size "0") )/15;;   (* espacement en x *)
let graph_dy = (3 * snd (text_size "0") );;       (* espacement en y *)

let rec ecart = function                 (* écartement variable entre noeuds *)
  | Atome _ | Vrai | Faux -> 1
  | Non (p)   -> 1+ecart p
  | Imp (g,d) -> (ecart g) + (ecart d)
  | Et (g,d)  -> (ecart g) + (ecart d)
  | Ou (g,d)  -> (ecart g) + (ecart d)
;;
(* ecart : proposition -> int = <fun> *)

let dessine_arbre t =
     let  x = (size_x () /2) and y = (size_y () - graph_dy)
     in clear_graph ();
     moveto x y; dessine_noeud x y t

where rec dessine_noeud x y t =
  let tmp x y g d s =
        let zg = max 1 (ecart g) and zd = max 1 (ecart d ) in
        lineto x y; dessine_noeud (x - zd * graph_dx) (y - graph_dy) g;
        moveto x y; dessine_noeud (x + zg * graph_dx) (y - graph_dy) d;
        moveto x y; draw_circle x y 14;
                    set_color white; fill_circle  x y 13; set_color black;
                    let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

  in match t with
    | Atome c   -> lineto x y;
                   let s = string_of_char c in
                   let (u,v) = text_size s
                   in moveto (x-u/2) (y-v/2); draw_string s;
    | Vrai      -> lineto x y;
                   let s = "V" in
                   let (u,v) = text_size s
                   in moveto (x-u/2) (y-v/2); draw_string s;
    | Faux      -> lineto x y;
                   let s = "F" in
                   let (u,v) = text_size s
                   in moveto (x-u/2) (y-v/2); draw_string s;

    | Non (p)   -> lineto x y;
                   dessine_noeud x (y - graph_dy) p;

                   moveto x y; draw_circle x y 14;
                   set_color white; fill_circle  x y 13; set_color black;
                   let s = string_of_char (char_of_int 172) in
                   let (u,v) = text_size s in moveto (x-u/2) (y-v/2); draw_string s;

   | Imp (g,d) -> tmp x y g d "=>"
   | Et (g,d)  -> tmp x y g d "Et"
   | Ou (g,d)  -> tmp x y g d "Ou"
;;
(* dessine_arbre : proposition -> unit = <fun> *)

let s = "((a ou Non b) => ((a et d) => (b ou c)))";;
let p = proposition_of_string s;;
dessine_arbre p;;

(*-------------------------------------------------------------------------------------*)
(*====================== Suite = Squelette. INCORRECT A PARTIR D'ICI ====================*)
(*-------------------------------------------------------------------------------------*)
```

8 Squelette de programme (suite du préambule), non exécutable !

```
(*-- 2.2.2. expression d'un arbre, sous forme de chaîne EATP ---*)                (* A ECRIRE  *)

(*-- 3.2. Union (de deux listes), sans répétitions ----*)                         (* A ECRIRE  *)

(*-- 3.3. Produit -----*)                                                        (* A ECRIRE  *)
(*----------------- + Question : complexité à traiter .... *)

(*-- 3.4. Mise sous forme clausale (pas tout à fait normale conjonctive) --*)     (* A ECRIRE  *)
(*----------------- + Questions : terminaison et correction de la définition .... *)

(*----------------- Exemples -----*)
let s1 = "( (a et b) => a )";;       let F1 = proposition_of_string s1;;  fnc F1;;
let s2 = "( a => (a et b))";;        let F2 = proposition_of_string s2;;  fnc F2;;
let s3 = "( a => ((a => b) => b))";; let F3 = proposition_of_string s3;;  fnc F3;;

(*-- 4.1. Calcul des résolutions ----------*)                                     (* A ECRIRE  *)

(*-- 4.2.1. Dérivation ----------*)                                              (* A ECRIRE  *)
(*----------------- + Questions : terminaison, complexité à traiter .... *)

(*-- 4.2.2. Test d'une formule propositionnelle ---*)                            (* A ECRIRE  *)

(*------------------ exemples simples -----*)
let s1 = "( (a ou b) => a )";; let F1 = proposition_of_string s1;; tautologie F1;;
let s2 = "( a => (a ou b))";;  let F2 = proposition_of_string s2;; tautologie F2;;

(*-- 4.3. Le Club Ecossais -------*)                                             (* A ECRIRE  *)

(*-- 4.4. Exemple (trop gros ?) -------*)
let sE1 = "(b => ((a et Non c) ou (Non a et c) ) )";;
let sE2 = "((c ou (Non a et Non b)) => (c et (a ou b)))";; let sE3 = "((a => c) et (Non a => Non b))";;

let E1 = proposition_of_string sE1;;
let E2 = proposition_of_string sE2;; let E3 = proposition_of_string sE3;;

(*-- 4.4.1. Dessin de la proposition R ---*)                                     (* A ECRIRE  *)

(*-- 4.4.2. Transformation d'une forme clausale en proposition ---*)             (* A ECRIRE  *)
(*--- 4.4.2.a) forme totalement déséquilibrée !!! -----*)
(*--- 4.4.2.b) forme  équilibrée : indication -----*)
let rec partage = function
     []  -> [], []
   | [a] -> [a], []
   | a::b::q -> let u,v = partage q in a::u, b::v
;;
(* partage : 'a list -> 'a list * 'a list = <fun> *)

(*-- 5. Evaluation ----------------- *)
(*-- 5.1. Fonction d'évaluation   ---*)                                          (* A ECRIRE  *)

(*-- 5.2. Exemple (cf ci-dessus)  ---*)

(*----- tableau des valeurs de vérité de R --*)                                 (* A ECRIRE  *)

(*-- 6. Résolutions (Liste des réductions cumulées) ----------------- *)         (* A ECRIRE  *)
(* cela peut devennir vite gros, surtout si on n'a pas une tautologie *)

(************************************)
(* Fin de TP Robins-S  (Squelette)  *)
(************************************)
```

$< \mathscr{FJN} >$ **Robinson** (énoncé + préambule + squelette).

9 TP Robinson : corrigé

```
(*== 1. ===============*)
(*== 2. ===============*)
(*== 2.1. =============*)
(*== 2.2. =============*)
(*-- 2.2.1. -----------*)
(*-- 2.2.2. ---------------------------------- chaîne EATP d'un arbre ----*)
let rec string_of_proposition = function
    Atome a   -> char_for_read a   | Vrai -> "1"  | Faux -> "0"
  | Non f     -> "Non " ^ string_of_proposition f
  | Imp (g,d) -> "(" ^ string_of_proposition g ^ " => " ^ string_of_proposition d ^ ")"
  | Et (g,d)  -> "(" ^ string_of_proposition g ^ " Et " ^ string_of_proposition d ^ ")"
  | Ou (g,d)  -> "(" ^ string_of_proposition g ^ " Ou " ^ string_of_proposition d ^ ")"
;;
(* string_of_proposition : proposition -> string = <fun> *)

let s = "((a et b) => (a => (b Ou c)))";;
let P = proposition_of_string s;;
string_of_proposition P;;

(*-- 2.2.3. -----------*)
(*== 3. ===============*)
(*-- 3.1. -------------*)
(*-- 3.2. --------------------------- Union (de deux listes L1 et L2) ----*)
(* --- En partant d'un accumulateur vide ([]), on ajoute en fin de liste   *)
(*     les éléments de L1 puis ceux de L2, s'il ne sont pas déjà dans      *)
(*     l'accumulateur, ce qui donne un résultat sans répétitions           *)
let union l1 l2 =
  let rec add x = function        (* 'a -> 'a list -> 'a list *)
      []          -> [x]
    | (a::r) as l -> if a = x then l else a::(add x r)
  in
  let rec iter acc = function     (* 'a list -> 'a list -> 'a list *)
      []    -> acc
    | a::r -> iter (add a acc) r
  in
  iter (iter [] l1) l2
;;
(* union : 'a list -> 'a list -> 'a list *)
```

Complexite de union $L_l\ L_2$: Avec $n_1 = |L_1|$ et $n_2 = |L_2|$:

$$C(n_1,n_2) = \underbrace{1+2+\cdots+(n_1-1)}_{\text{ajout } \ell_1} + \underbrace{n_1+(n_1+1)+(n_1+2)+\cdots+(n_1+n_2-1)}_{\text{ajout } \ell_2} = \frac{(n_1+n_2)\,(n_1+n_2-1)}{2}.$$

La complexité de $(\texttt{union}\ \ L_1\ L_2)$ est $\dfrac{(|L_1|+|L_2|)\,(|L_1|+|L_2|-1)}{2} = O\big((|L_1|+|L_2|)^2\big)$.

```
(*-- 3.3. Produit -------------------*)
(* let (prod : forme_clausale -> forme_clausale -> forme_clausale) = fun a b ->  *)
let prod = fun a b ->
  let rec prod_aux ai = function    (* clause -> forme_clausale -> forme_clausale *)
      []     -> []
    | bj::bq -> (union ai bj) :: (prod_aux ai bq)
  in
  let rec iter = function           (* forme_clausale -> forme_clausale *)
      []     -> []
    | ai::aq -> (prod_aux ai b) @ (iter aq)
  in
  iter a
;;
(* prod : 'a list list -> 'a list list -> 'a list list = <fun>  *)
(* (* prod : forme_clausale -> forme_clausale -> forme_clausale *) *)

prod [["a11";"a12"]; ["a21";"a22";"a23"] ] [["b11";"b12";"b13"]; ["b21";"b22";"b23"] ] ;;
```

Remarques.

- On pourrait se passer de `@` à l'aide d'un accumulateur, ce qui éviterait certains doublons ... .
- La conception de `union` permet de l'appliquer à des listes de listes.
- L'égalité utilisée dans `union` étant structurelle, on risque d'avoir des doublons dans une forme clausale, avec des clauses "égales" mais pas représentées dans le même ordre. Par exemple la clause $[\ell_1, \ell_2]$ sera considérée comme différente de la clause $[\ell_2, \ell_1]$.

Pour chaque a_i de la liste A, on transforme la liste B en faisant l'union de chacun des b_i avec a_i puis, pour la liste A, on fait l'union (ici la mise bout par `a`) du tout. Complexité de `prod A B` :
$C < \sum_i \sum_j \frac{\left(|a_i| + |b_j|\right)\left(|a_i| + |b_j| - 1\right)}{2}$, soit $C < |A| \times |B| \times \frac{\max\left(|a_i| + |b_j|\right)^2}{2}$.

```
(*-- 3.4. Mise sous forme clausale --*)

let rec (fnc : proposition -> forme_clausale) =  = fun p ->
  match p with
      Atome _ | Non (Atome _) | Faux | Non Faux  -> [[p]]    (* groupement, nommé avec match p *)

    | Vrai  -> [[Non Faux]]
    | Non Vrai -> [[Faux]]

    | Et (a,b)        -> let ca = fnc a and cb = fnc b in union ca cb
    | Ou (a,b)        -> let ca = fnc a and cb = fnc b in prod ca cb
    | Imp (a,b)       -> let ca = fnc (Non a) and cb = fnc b in prod ca cb
    | Non (Et (a,b))  -> let ca = fnc (Non a) and cb = fnc (Non b) in prod ca cb
    | Non (Ou (a,b))  -> let ca = fnc (Non a) and cb = fnc (Non b) in union ca cb
    | Non (Imp (a,b)) -> let ca = fnc a and cb = fnc (Non b) in union ca cb
    | Non (Non a)     -> fnc a
;;
(* fnc : proposition -> forme_clausale *)
```

On risque fortement d'avoir des doublons, avec des clauses identiques mais pas dans le même ordre ... .

- Terminaison et preuve :
  On définit la taille t d'une proposition par : $\begin{cases} t(\texttt{Atome\_}) = t(\texttt{Faux}) = t(\texttt{Vrai}) = 1 \\ t(A \vee B) = t(A \wedge B) = t(A \Rightarrow B) = 1 + t(A) + t(B) \\ t(\neg A) = 1 + t(A) \end{cases}$
  la taille des propositions décroit strictement, tous les cas sont étudiés et on termine avec les cas de base.
- Complexité :
  - Si h est la hauteur de la proposition P alors la forme normale conjonctive $\mathscr{C}(P)$ de P contient au plus $n(h) = 2^{(2^h)}$ clauses.
    Explication : un exemple de pire cas est $(P \Rightarrow Q)$ qui donne $(\overline{P} \vee Q)$, avec $n(h) = "n(h)" + n(h-1)$ dans un premier temps puis $n(h) = \big(n(h-1) + n(h-1)\big) + n(h-1)$ dans un deuxième temps. On aura (largement) $n(h) \leqslant 4\,n(h-1)$.
  - Chaque clause contient des litéraux distincts et en contient donc au maximum $2 \times (2^h)$.
    Explication : l'arbre étant de hauteur p, a dans le pire cas au plus 2^p feuilles qui sont des litéraux, et on prend en compte les litéraux négatifs.

```
(*-------- Exemples : *)
let s1 = "( (a et b) => a )";; let F1 = proposition_of_string s1;; fnc F1;;
                               (* [[Non (Atome `a`); Non (Atome `b`); Atome `a`]] *)

let s2 = "( a => (a et b))";; let F2 = proposition_of_string s2;; fnc F2;;
                               (* [[Non (Atome `a`); Atome `a`]; [Non (Atome `a`); Atome `b`]] *)
```

```
(*== 4. ========================*)
(*-- 4.1. Calcul des résolutions ----------*)
let resolutions p q =
  let parcours l1 l2 =                      (* clause -> clause -> forme_clausale *)
    let rec parcours_rec = function         (* clause -> forme_clausale *)
           []         -> []
         | (Non a)::l -> if mem a l1
                         then let r = union (except a l1) (except (Non a) l2) in
                              (if r = [] then [Faux] else r) :: (parcours_rec l)
                         else parcours_rec l
         | _::l       -> parcours_rec l
    in
    parcours_rec l2
  in
 (parcours p q) @ (parcours q p)            (* ou union ? mais doublons "ensemble" ? *)
;;
(* resolutions : clause -> clause -> clause list *)
(* resolutions : proposition list -> proposition list -> proposition list list = <fun> *)
```

Complexité de `resolutions p q` : Pour deux clauses de taille $\leqslant n$, il ne peut y avoir plus de n résolutions, car les litéraux de chaque clause sont distincts (assuré par `union`).
Chaque résolution est une clause de taille $\leqslant n-1$. On a donc une complexité de $n(n-)$.

```
(*-- 4.2.1. Dérivation ----------*)
let paires l =
  let rec prod_rec x = function   (* 'a -> 'a list -> (('a * 'a) * 'a list) list *)
      []   -> []
    | y::r -> let l' = except y (except x l) in
              ((x,y) , l') :: (prod_rec x r)
  in
  let rec iter = function         (* 'a list -> (('a * 'a) * 'a list) list *)
      []   -> []
    | x::r -> (prod_rec x r) @ (iter r)
  in
  iter l
;;
(* paires : 'a list -> (('a * 'a) * 'a list) list = <fun> *)

(*--------------------------------*)
let rec derive_test l =
  exists essai (paires l)

  where essai ((x,y),l') =         (* (clause * clause) * forme_clausale-> bool *)
    let lr = resolutions x y in
    (mem [Faux] lr) or exists (fun r -> derive_test (r::l')) lr
;;
(* derive_test : proposition list list -> bool = <fun> *)
```

- Terminaison de `derive_test` ℓ : ...
- Complexité de `derive_test` ℓ : On suppose que ℓ contient n clauses de taille $\leqslant t$.
  Alors $C(n) = \dfrac{n(n-1)}{2} \times t \times C(n-1)$, d'où $C(n) = \dfrac{(n!)^2 \times t^n}{n\,2^n} \underset{n\to+\infty}{\sim} A \times (B \times n)^{2n}$ (Stirling).
  Cette méthode est donc inapplicable en pratique !

  Comparaison avec la méthode utilisant les tables de vérité : Le nombre de propositions atomiques p de P se situe entre $\log_2(n)$ et n et la méthode des tables de vérité est en 2^p.
  La méthode utilisant les tables de vérité est donc plus réaliste que la méthode par résolution.

```
(*-- 4.2.2. Test d'une formule propositionnelle ---*)
let tautologie p = derive_test (fnc (Non p) )
;;
(* tautologie : proposition -> bool = <fun> *)

(* -- Exemples simples : *)
let s1 = "( (a ou b) => a )";;  let F1 = proposition_of_string s1;; tautologie F1;;

let s2 = "( a => (a ou b))";;   let F2 = proposition_of_string s2;; tautologie F2;;
```

```
(*--------------------------------*)
(*-- 4.3. Le Club Ecossais -------*)
(*--------------------------------*)
(*---------- atomes --*)
let E = Atome `e`  (* est Ecossais *)
and C = Atome `c`  (* porte des Chaussettes rouges *)
and K = Atome `k`  (* porte un Kilt *)
and M = Atome `m`  (* est Marié *)
and D = Atome `d`  (* sort le Dimanche *) ;;
(*----------- traduction des règles, dans l'ordre de l'énoncé *)
let r1 = Imp (Non E, C)              and  r2 = Ou (K, Non C)       and  r3 = Imp (M, Non D)
and r4 = Et (Imp (D, E) , Imp (E,D))  and  r5 = Imp (K, Et (E, M))  and r6 = Imp (E, K) ;;
```

Construction de la proposition P de non existence d'un membre :

- Existence d'un membre : $\big(\exists\, x :\ (r_1 = V) \wedge (r_2 = V) \wedge \ldots \wedge (r_6 = V)\big)$
- Non existence d'un membre : $\big(\forall\, x,\ (r_1 = F) \wedge (r_2 = F) \wedge \ldots \wedge (r_6 = F)\big)$

Donc la proposition de non existence d'un membre est $P = \Big((r_1 \wedge r_2 \wedge \ldots \wedge r_6) \Rightarrow F\Big)$.

Autre forme : On sait que $\big((p \wedge q) \Rightarrow F\big) \equiv \big(p \Rightarrow (q \Rightarrow F)\big)$ et, par extrapolation :

$$\Big((r_1 \wedge r_2 \wedge \ldots \wedge r_6) \Rightarrow F\Big) \equiv \Big(r_1 \Rightarrow (r_2 \Rightarrow (r_3 \Rightarrow (r_4 \Rightarrow (r_5 \Rightarrow (r_6 \Rightarrow F))))))\Big)$$

Donc $P \equiv Q$ avec $Q = \Big(r_1 \Rightarrow (r_2 \Rightarrow (r_3 \Rightarrow (r_4 \Rightarrow (r_5 \Rightarrow (r_6 \Rightarrow F))))))\Big)$.

```
let P = Imp (Et( r1, Et (r2, Et (r3, Et (r4, Et (r5, r6))))), Faux);;
tautologie P;;  (* - : bool = true *)
let Q = Imp (r1, Imp (r2, Imp(r3 ,Imp (r4, Imp(r5, Imp (r6, Faux)))))) ;;
tautologie Q;; (* - : bool = true *)

(*-------------------------------------*)
(*-- 4.4. Exemple (formule contigente) *)
(*-------------------------------------*)
let sE1 = "(b => ((a et Non c) ou (Non a et c) ) )";;       let E1 = proposition_of_string sE1;;
let sE2 = "((c ou (Non a et Non b)) => (c et (a ou b)))";;  let E2 = proposition_of_string sE2;;
let sE3 = "((a => c) et (Non a => Non b))";;                let E3 = proposition_of_string sE3;;
dessine_arbre E1;; dessine_arbre E2;; dessine_arbre E3;;
tautologie E1;;    tautologie E2;;    tautologie E3;;

(*-- 4.4.1. Dessin de la proposition R  (R n'est vrai que pour a=1, b=0 et c=1) *)
let R = Ou ( Et (Et (E1,E2),E3), Et (Et (Non E1, Non E2), Non E3));;
dessine_arbre R;;
(* tautologie R;;  *) (* on peut attendre longtemps et un jour, Uncaught exception: Out_of_memory *)

(*-- 4.4.2. Transformation d'une forme clausale en proposition  ---- *)

(*--- 4.4.2.a) forme totalement déséquilibrée !!! -----*)
let proposition_of_fnc  w =
  let h = fun x' y' -> Ou (x', y') in
  let g u = it_list h (hd u) (tl u) in
  let f = fun x y -> Et ( x, g y ) in
  it_list f (g (hd w)) (tl w)
;;
(* proposition_of_fnc : proposition list list -> proposition = <fun> *)

let CE1 = fnc( (Non E1));; let E1' = proposition_of_fnc CE1;; dessine_arbre E1';;
let CE2 = fnc( (Non E2));; let E2' = proposition_of_fnc CE2;; dessine_arbre E2';;
let CE3 = fnc( (Non E3));; let E3' = proposition_of_fnc CE3;; dessine_arbre E3';;

let CR = fnc( (Non R));;          (* c'est gros *)
let R' = proposition_of_fnc CR;;  (* c'est énorme !  *)
dessine_arbre R';;                (* c'est illisible *)
```

Ici, chaque clause ou chaque forme clausale conduit à un arbre peigne.

Pour avoir une forme équilibére de hauteur réduite, on partage les clauses ainsi que les formes clausales en deux parties (fils gauche et fils droit) égale à 1 près.

```
(*-- 4.4.2.b) forme équilibrée -----*)
let rec partage = function
     []  -> [], []
   | [a] -> [a], []
   | a::b::q -> let u,v = partage q in a::u, b::v
;;
(* partage : 'a list -> 'a list * 'a list = <fun> *)

let rec proposition_of_clause = function
  | []     -> failwith "erreur"
  | [a]    -> a
  | [g; d] -> Ou (g, d)
  | q      -> let g, d = partage q in Ou (proposition_of_clause g, proposition_of_clause d)
;;
(* proposition_of_clause : proposition list -> proposition = <fun> *)

let rec proposition_of_forme_clausale = function
  | []     -> failwith "erreur"
  | [a]    -> proposition_of_clause a
  | [g; d] -> Et (proposition_of_clause g, proposition_of_clause d)
  | q      -> let g, d = partage q
              in Et (proposition_of_forme_clausale g, proposition_of_forme_clausale d)
;;
(* proposition_of_forme_clausale : proposition list list -> proposition = <fun> *)

let CE1 = fnc( (Non E1));; let E1'' = proposition_of_forme_clausale CE1;; dessine_arbre E1'';;
let CE2 = fnc( (Non E2));; let E2'' = proposition_of_forme_clausale CE2;; dessine_arbre E2'';;
let CE3 = fnc( (Non E3));; let E3'' = proposition_of_forme_clausale CE3;; dessine_arbre E3'';;

let CR = fnc( (Non R));;                        (* c'est gros *)
let R'' = proposition_of_forme_clausale CR;;    (* c'est gros !  *)
dessine_arbre R'';;                             (* c'est équilibré mais trop gros *)

(*-- 5. Evaluation =================*)
(*-- 5.1. Fonction d'évaluation ---*)
let eval v = evalue
  where rec evalue = function
    | Atome c   ->  let n = int_of_char c - int_of_char `a` in v.(n)
    | Vrai      -> 1
    | Faux      -> 0;
    | Non (p)   -> let u = evalue p in (u+1) mod 2
    | Imp (g,d) -> let u = evalue g and v = evalue d in if u = 1 then v else 1
    | Et  (g,d) -> let u = evalue g and v = evalue d in u * v
    | Ou  (g,d) -> let u = evalue g and v = evalue d in max u v
;;
(* eval : int vect -> proposition -> int = <fun> *)

(*-- 5.2. Exemple (cf ci-dessus) ---*)
let sE1 = "(b => ((a et Non c) ou (Non a et c) ) )";;       let E1 = proposition_of_string sE1;;
let sE2 = "((c ou (Non a et Non b)) => (c et (a ou b)))";;  let E2 = proposition_of_string sE2;;
let sE3 = "((a => c) et (Non a => Non b))";;                let E3 = proposition_of_string sE3;;
let R = Ou ( Et (Et (E1,E2),E3), Et (Et (Non E1, Non E2), Non E3));;
dessine_arbre R;;

(*----- tableau des valeurs de vérité de R  ---*)
print_newline ();
for i = 0 to 1
do for j = 0 to 1
   do for k = 0 to 1
      do printf "%2u" i; printf "%2u" j; printf "%2u" k;
         printf "%2u" (eval [|i;j;k|] R);
         print_newline () done
   done
done;;
```

R est statisfiable,uniquement avec $a = 1, b = 0, c = 1$.

```
(*== 6. Résolutions (Liste des réductions cumulées de P) ------------------- *)
(* cela peut devennir vite gros, surtout si (Non P) n'est pas une tautologie *)

let rec derive2_test l =
  it_list essai2 [] (paires l)

  where essai2 accu ((x,y),l') = (* 'a ->  (clause * clause) * forme_clausale -> forme_clausale *)
    let lr = resolutions x y in
    match l' with
       [] -> lr
     | _  -> it_list (fun u r -> u @ derive2_test (r::l')) [] lr
;;
(* derive2_test : proposition list list -> proposition list list = <fun> *)

let s1 = "( (a ou b) => a )";;  let F1 = proposition_of_string s1;;
tautologie F1;;
let P = fnc( (Non F1));;  let l = derive2_test P;;

let s3 = "( a => ((a => b)=>b))";; let F3 = proposition_of_string s3;;
tautologie F3;;
let P = fnc( (Non F3));;  let l = derive2_test P;;

let sE1 = "(b => ((a et Non c) ou (Non a et c) ) )";;
let E1 = proposition_of_string sE1;;
let P = fnc( (Non E1));; let l = derive2_test P;;

(************************************)
(* Fin de TP Robins-C  (Corrigé)     *)
(************************************)
```

$< \mathscr{FIN} >$ TP Robinson. Corrigé.