

**Informatique MP**  
**TP**  
**Résolution propositionnelle.**  
**Théorème de Robinson (1965).**

Antoine MOTEAU  
antoine.moteau@wanadoo.fr

---

.../Robins-C.tex (2003)  
.../Robins-C.tex Compilé le dimanche 11 mars 2018 à 13h 47m 10s avec [LaTeX](#).  
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

---

Ordre du TP : premier tiers d'année (TP 3 ou 4 )

Reprise sur TP Jean-Christophe Filliâtre

G:\Web Info MP 13 Mars 2002\TP filiatre\Résolution\

<http://caml.inria.fr/polycopies/filliatre/resolution.ps>

**Modifications apportées : ...**

Éléments utilisés :

- Logique
- listes
- arbres
- Flux (complètement donné dans l'énoncé et squelette)
- fonctions récursives (c'est un peu chaud ici)
- 

Éléments évoqués :

- flux
- Exceptions (gestion des)
- 

Documents relatifs au TP :

- Texte du TP : Robins-C. tex , .dvi, .pdf
- Squelette de programme :
  - Dans le pdf :
    - déclarations,
    - construction d'une proposition depuis un flux de char (expression algébrique parenthésée)
    - dessin de l'arbre d'une proposition
  - Robins-S.ml (quasi identique à ce qui est écrit dans le .pdf)
- Programme corrigé : Robins-C.ml

## Résolution propositionnelle. Robinson (1965).

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

Les ressources pour ce TP sont dans le dossier .../MPx/TP-Info/Robinson/ et après avoir copié (tout) le dossier **/Robinson/** dans votre dossier personnel, vous y trouverez les fichiers :

- Robins-S.pdf : ce texte,
- Robins-S.ml : fichier Caml à compléter (squelette de programme).

### 1 Logique propositionnelle (rappels)

Soit  $\mathcal{T}$  un ensemble fini de symboles de *propositions atomiques* ("variables" propositionnelles).

On considère l'ensemble  $\mathcal{A}(\mathcal{T})$  des *formules propositionnelles* définies sur l'ensemble  $\mathcal{T}$  comme étant le plus petit ensemble tel que :

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

où ces règles sont appliquées un nombre fini de fois.

Une formule propositionnelle définie sur  $\mathcal{T}$  sera représentée par (et même assimilée à) un arbre dont les nœuds sont les connecteurs logiques ( $\neg$ ,  $\Rightarrow$ ,  $\wedge$ ,  $\vee$ ) 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  $\neg$ Faux (ici, Vrai ne sera pas un littéral).

Conventions, restrictions :

- on écrira parfois
  - formule ou proposition à la place de formule propositionnelle,
  - atome ou variable à la place de proposition atomique ;
- on suppose que les éléments de  $\mathcal{T}$  sont repérés par des lettres minuscules, de  $a$  à  $z$  ;
- bien que les formules de  $\mathcal{A}(\mathcal{T})$  soient définies de façon (quasi) totalement parenthésée (EATP), pour améliorer la lisibilité, on utilisera (dans ce texte, mais pas dans les programmes) des écritures simplifiées avec un minimum de parenthèses (EANTP), interprétées selon les règles de priorité usuelles.

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

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.

## 2 Représentation arborescente des propositions

### 2.1 Type Caml des propositions

```
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;;
```

Par exemple, depuis des formules EANTP (à parenthèses implicites), interprétées selon les règles de priorité usuelles :

- $a \Rightarrow (a \Rightarrow b) \Rightarrow b$ , de forme totalement parenthésée unique  $(a \Rightarrow ((a \Rightarrow b) \Rightarrow b))$  sera représentée par  
Imp (a, Imp (Imp (a,b), b))
- $a \vee b \vee c \vee d$  pourra être représentée par
  - Ou (a, Ou (b, Ou (c,d) ) ) , d'après la forme totalement parenthésée  $(a \vee (b \vee (c \vee d)))$
  - Ou (Ou (a, b), Ou (c, d) ) , d'après la forme totalement parenthésée  $((a \vee b) \vee (c \vee d))$

### 2.2 Interface

#### 2.2.1 Formules contenues dans une chaîne

Les formules (quasi complètement parenthésées), seront conformes à la grammaire :

$$A ::= a \mid \dots \mid z \mid 0 \mid 1 \mid \underbrace{\neg A}_{\substack{\text{Non } A \\ \text{avec N}}} \mid \underbrace{(A \wedge A)}_{\substack{(A \text{ Et } A) \\ (A \text{ et } A)}} \mid \underbrace{(A \vee A)}_{\substack{(A \text{ Ou } A) \\ (A \text{ ou } A)}} \mid \underbrace{(A \Rightarrow A)}_{(A \Rightarrow A)}$$

où les lettres  $a, \dots, z$  servent de noms pour les variables propositionnelles, Faux est représenté par 0 et Vrai par 1, les espaces étant ici non significatifs.

Les formules pourront être représentées initialement à l'aide de chaînes de caractères, sous forme (quasi) EATP, (où il est préférable de laisser des espaces !), par exemple "(e et t) => (a ou Non b)"; ; (ce qui est plus lisible que "((eEt t)=>(aouNon b))"; ; ), avant d'être converties en propositions.

*Remarque.* Et peut s'écrire aussi et, Ou peut s'écrire aussi ou, mais pour des raisons pratiques de programmation, Non ne peut pas s'écrire avec une minuscule.

#### 2.2.2 Analyse et transformation d'une formule chaîne en un arbre proposition

On disposera de la fonction de construction de proposition (préparée dans le fichier squelette) :

proposition\_of\_string : string -> proposition

proposition\_of\_string s renvoie la proposition construite à partir de la formule (quasi) EATP contenue dans la chaîne de caractères s. Provoque les exceptions Parse\_failure ou Parse\_error en cas d'erreur d'analyse du flux intermédiaire (formule vide, contenant des caractères non reconnus, incorrecte, mal parenthésée).

On voudra, bien entendu, disposer de la fonction réciproque, plus facile à écrire :

string\_of\_proposition : proposition -> string

A ECRIRE

string\_of\_proposition p renvoie la chaîne représentant, sous forme (quasi) EATP, la formule représentée par l'arbre proposition p.

(cette construction se fait à l'aide d'un parcours en profondeur classique de l'arbre).

#### 2.2.3 Représentation graphique d'un arbre proposition

On disposera de la ressource de dessin d'arbre (préparée dans le fichier squelette) :

dessine\_arbre : proposition -> unit = <fun>

dessine\_arbre p représente (tant bien que mal) l'arbre proposition p (pas trop compliqué) sur la fenêtre graphique de Caml.

## 3 Forme normale conjunctive

### 3.1 Définitions

**Définition 3.1.** (Clauses et formes clausales)

Une *clause* est une disjonction  $\ell_1 \vee \ell_2 \vee \dots \vee \ell_n$  où les  $\ell_i$  sont des littéraux.

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

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

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

```
type clause == proposition list;;
type forme_clausale == clause list;;
```

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

Le coût de comparaison, deux à deux, des ensembles sous-jacents à des listes étant élevé, le choix de la représentation, par liste de listes, d'une forme normale conjunctive nous amène à y admettre des clauses identiques, c'est à dire comportant exactement les mêmes littéraux (dans un ordre différent).  
On espère que cette stratégie ne conduira pas à des ambiguïtés ou à une complexité trop élevée ...

### 3.2 Unions

1. Union disjonctive de deux clauses  $C_1 = \ell_1^1 \vee \dots \vee \ell_1^n$  et  $C_2 = \ell_2^1 \vee \dots \vee \ell_2^m$  :

On note  $C_1 \nabla C_2$  la disjonction  $\ell_1^1 \vee \dots \vee \ell_1^n \vee \ell_2^1 \vee \dots \vee \ell_2^m$  dans laquelle on a supprimée toute répétition.

2. Union conjonctive de deux formes clausales  $F_1$  et  $F_2$  :

On note  $F_1 \overline{\wedge} F_2$  la conjonction des clauses de  $F_1$  et  $F_2$ , parmi lesquelles on a supprimé toute répétition.

*Remarque.* Mais, dans cette conjonction, pour des raisons pratiques (voir encadré ci-dessus) on sera aussi amené à admettre des clauses comportant exactement les mêmes littéraux (mais dans un ordre différent), en espérant que ...

Comme les clauses et les formes clausales sont représentées par des listes, les deux opérations  $\nabla$  et  $\overline{\wedge}$  sont tout simplement réalisées par la 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.

▷ **Question** : Quelle est la complexité de la fonction union en fonction de la taille des arguments et en terme de nombre d'opérations :: ?

### 3.3 Produit de formes clausales

Soit  $A = A_1 \wedge \dots \wedge A_n$  et  $B = B_1 \wedge \dots \wedge B_m$  deux formes clausales (les  $A_i$  et  $B_i$  étant des clauses). On note  $A \otimes B$  la forme clausale (normale conjunctive) définie par

$$A \otimes B = \overline{\bigwedge_{i,j} A_i \nabla B_j}$$

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

► prod : forme\_clausale -> forme\_clausale -> forme\_clausale

A ECRIRE

ou

prod : 'a list list -> 'a list list -> 'a list list, ce qui permet d'écrire le test simple :

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

▷ **Question** : Quelle est la complexité de la fonction prod en fonction de la taille des formes clausales passées en arguments et en terme de nombre d'opérations :: ?

### 3.4 Mise d'une proposition sous forme normale conjonctive

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

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

- Si  $P$  est un littéral, alors  $\mathcal{C}(P) = P$ .
- $\mathcal{C}(\text{Vrai}) = \neg \text{Faux}$  et  $\mathcal{C}(\neg \text{Vrai}) = \text{Faux}$
- $\mathcal{C}(A \wedge B) = \mathcal{C}(A) \wedge \mathcal{C}(B)$
- $\mathcal{C}(A \vee B) = \mathcal{C}(A) \vee \mathcal{C}(B)$
- $\mathcal{C}(A \Rightarrow B) = \mathcal{C}(\neg A) \vee \mathcal{C}(B)$
- $\mathcal{C}(\neg(A \wedge B)) = \mathcal{C}(\neg A) \vee \mathcal{C}(\neg B)$
- $\mathcal{C}(\neg(A \vee B)) = \mathcal{C}(\neg A) \wedge \mathcal{C}(\neg B)$
- $\mathcal{C}(\neg(A \Rightarrow B)) = \mathcal{C}(A) \wedge \mathcal{C}(\neg B)$
- $\mathcal{C}(\neg(\neg A)) = \mathcal{C}(A)$

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

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

► fnc : proposition -> forme\_clausale

A ECRIRE

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

**Proposition 3.1.**

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

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

## 4 Résolution propositionnelle

### 4.1 Règle de résolution

**Définition 4.1.**

À deux clauses  $A = \alpha_1 \vee \dots \vee \alpha_p \vee \ell \vee \alpha_{p+1} \vee \dots \vee \alpha_n$   
et  $B = \beta_1 \vee \dots \vee \beta_q \vee \neg \ell \vee \beta_{q+1} \vee \dots \vee \beta_m$   
la règle de résolution (propositionnelle) associe la clause  
$$R = (\alpha_1 \vee \dots \vee \alpha_p \vee \alpha_{p+1} \vee \dots \vee \alpha_n) \vee (\beta_1 \vee \dots \vee \beta_q \vee \beta_{q+1} \vee \dots \vee \beta_m)$$
  
(lorsque cette clause est vide on pose  $R = \text{Faux}$ ).  
On dit que  $R$  est une résolution de  $A$  et  $B$ .

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.

Exemple :,

- la résolution de  $A = p \vee \neg q$  et  $B = q \vee r \vee \neg p$  est la liste de deux clauses :  $[p \vee r \vee \neg p; \neg q \vee q \vee r]$
- la résolution de  $A = p \vee q$  et  $B = q \vee r$  est  $[]$  (ou  $[[\text{Faux}]]$ ).

Dans la suite, on pourra utiliser (mais il faut savoir les ré-écrire) les fonctions Caml :

value list\_\_except : 'a -> 'a list -> 'a list  
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.

value 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.

Il ne reste plus qu'à mettre en place le calcul des résolutions :

► resolutions : clause -> clause -> clause list

A ECRIRE

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

## 4.2 Résolution des tautologies

**Proposition 4.1.** (Robinson, 1965)

Soit  $P$  une proposition et  $C_1 \wedge \dots \wedge C_n$  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  $C_1, \dots, C_n$ .

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

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

1. Outils (on a déjà vu la fonction `list__except`)

- Ecrire la fonction :

► `paires : 'a list -> (('a * 'a) * 'a list) list = <fun>` A ECRIRE  
`paires L` renvoie la liste de tous les couples  $((x, y), L \setminus \{x, y\})$  où  $x$  et  $y$  sont des éléments distincts de  $L$ .

- On rappelle la fonction `Caml` (mais il faut savoir la réécrire) :

`value list__exists : ('a -> bool) -> 'a list -> bool`  
`exists p [a1 ; ... ; an] is (p a1) or (p a2) or ... or (p an)`. Rem `exists p [] is false`.

2. Processus de test par dérivation d'une liste de clauses  $L$  (arrêt sur test) :

(a) On calcule la liste  $L2 = \text{paires } L$ , de tous les couples  $((x, y), L \setminus \{x, y\})$

(b) Pour chaque élément  $((x, y), L')$  de  $L2$ ,

on calcule (avec `resolutions`) la liste  $R$  de toutes les résolutions de  $x$  et  $y$ .

- Si cette liste contient la clause réduite à Faux (`mem [Faux] R = true`), alors on retourne `true`
- Sinon, pour chaque résolution  $r$  de  $R$ ,  
on recommence le processus de dérivation (qui est donc récursif), sur la liste  $L' \cup \{r\}$ .

Ce processus est réalisé par la fonction :

► `derive_test : proposition list list -> bool = <fun>` A ECRIRE

▷ Questions :

- Prouver la terminaison de `derive_test`.
- Quelle est la complexité de `derive_test L` en fonction du nombre  $n$  de clauses de  $L$  et de la taille maximale  $t$  des clauses de  $L$  ?  
Comparer cette complexité avec la méthode des tables de vérité, appliquée à la proposition  $L$ .

### 4.2.2 Test d'une proposition

► `tautologie : proposition -> bool` A ECRIRE  
`tautologie P` renvoie `true` si  $P$  est une tautologie, `false` sinon.

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

---

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 inconsistent on arrive toujours à générer la clause vide. Par contre, le problème de la consistance (satisfaisabilité) n'étant pas décidable en logique des prédicat, 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 des 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

Archibald McArthur, descendant commun des deux clans McPherson et McKinnon, souhaite fusionner en un club McArthur les membres du club McPherson et ceux du club 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 définit les règles du club McArthur 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 contingente)

On considère les trois formules

$$\begin{cases} E_1 = (b \Rightarrow ((a \wedge \neg c) \vee (\neg a \wedge c))) \\ E_2 = ((c \vee (\neg a \wedge \neg b)) \Rightarrow (c \wedge (a \vee b))) \\ E_3 = ((a \Rightarrow c) \wedge (\neg a \Rightarrow \neg b)) \end{cases}$$

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

1. Représenter l'arbre de la formule  $R$ .

*Remarque.* On ne cherchera pas, par la méthode de résolution propositionnelle, à savoir si  $R$  est ou non une tautologie (ou alors, à l'année prochaine, en 5/2, avec Uncaught exception : Out\_of\_memory !).

2. Calculer, sous forme de formule propositionnelle,  $R'$ , forme normale conjonctive de  $R$ .

*Remarque.* Il s'agit de faire des "ou" entre les littéraux de chaque clause de la forme clausale de  $R$ , puis de faire des "et" entre toutes les formules obtenues. On pourra faire des fonctions générales :

- ▶ `proposition_of_clause : proposition list -> proposition`  
`proposition_of_clause l` renvoie la proposition correspondant à la clause `l`.
- ▶ `proposition_of_forme_clausale : proposition list list -> proposition`  
`proposition_of_forme_clausale l` renvoie la proposition correspondant à la forme clausale `l`.

- ▷ 3. Quelle est la forme de l'arbre de  $R'$  ?  
Comment fallait-il faire pour avoir un arbre de hauteur minimale ?

## 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 aux variables propositionnelles  $a, b, \dots$  :

► `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)  $a \dots z$  sont consécutifs.

### 5.2 Application (table de vérité)

On reprend l'exemple précédent, avec les trois formules :

$$\begin{cases} E_1 = (b \Rightarrow ((a \wedge \neg c) \vee (\neg a \wedge c))) \\ E_2 = ((c \vee (\neg a \wedge \neg b)) \Rightarrow (c \wedge (a \vee b))) \\ E_3 = ((a \Rightarrow c) \wedge (\neg a \Rightarrow \neg b)) \end{cases}$$

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

▷ 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 memory" !

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

$\langle \mathcal{F} \mathcal{I} \mathcal{N} \rangle$  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' ); 't' >] -> (fun f1 f2 -> Et (f1,f2)) (* et, Et *)
  | [<'(' | '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 devenir vite gros, surtout si on n'a pas une tautologie *)

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

$\langle \mathcal{F} \mathcal{I} \mathcal{N} \rangle$  **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 *)
```

Complexité de union  $L_1 L_2$  : Avec  $n_1 = |L_1|$  et  $n_2 = |L_2|$  :

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

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

```
(*-- 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 bq) :: (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  $\text{prod } A \ B$  :

$$C < \sum_i \sum_j \frac{(|a_i| + |b_j|)(|a_i| + |b_j| - 1)}{2}, \text{ soit } C < |A| \times |B| \times \frac{\max(|a_i| + |b_j|)^2}{2}.$$

(\*-- 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 :

$$\text{On définit la taille } t \text{ d'une proposition par : } \begin{cases} t(\text{Atome}_-) = t(\text{Faux}) = t(\text{Vrai}) = 1 \\ t(A \vee B) = t(A \wedge B) = t(A \Rightarrow B) = 1 + t(A) + t(B) \\ t(\neg A) = 1 + t(A) \end{cases}$$

la taille des propositions décroît 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  $\mathcal{C}(P)$  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  $n(h) = (n(h-1) + n(h-1)) + n(h-1)$  dans un deuxième temps. On aura (largement)  $n(h) \leq 4n(h-1)$ .

- Chaque clause contient des littéraux 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 littéraux, et on prend en compte les littéraux 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 =
    let rec parcours_rec = function
      [] -> []
      | (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  $\leq n$ , il ne peut y avoir plus de  $n$  résolutions, car les littéraux de chaque clause sont distincts (assuré par union).  
Chaque résolution est une clause de taille  $\leq n - 1$ . On a donc une complexité de  $n(n-1)$ .

```

(*-- 4.2.1. Dérivation -----*)
let paires l =
  let rec prod_rec x = function
    [] -> []
    | y::r -> let l' = except y (except x l) in
      ((x,y), l') :: (prod_rec x r)
  in
  let rec iter = function
    [] -> []
    | 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  $\leq t$ .  
Alors  $C(n) = \frac{n(n-1)}{2} \times t \times C(n-1)$ , d'où  $C(n) = \frac{(n!)^2 \times t^n}{n2^n} \underset{n \rightarrow +\infty}{\sim} A \times (B \times n)^{2n}$  (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 :  $(\exists x : (r_1 = V) \wedge (r_2 = V) \wedge \dots \wedge (r_6 = V))$
- Non existence d'un membre :  $(\forall x, (r_1 = F) \wedge (r_2 = F) \wedge \dots \wedge (r_6 = F))$

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

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

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

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

```

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 contingente) *)
(*-----*)
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 équilibrée 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 = evaluate
  where rec evaluate = function
    | Atome c -> let n = int_of_char c - int_of_char 'a' in v.(n)
    | Vrai -> 1
    | Faux -> 0;
    | Non (p) -> let u = evaluate p in (u+1) mod 2
    | Imp (g,d) -> let u = evaluate g and v = evaluate d in if u = 1 then v else 1
    | Et (g,d) -> let u = evaluate g and v = evaluate d in u * v
    | Ou (g,d) -> let u = evaluate g and v = evaluate 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 satisfiable, uniquement avec  $a = 1, b = 0, c = 1$ .



```

(*== 6. Résolutions (Liste des réductions cumulées de P) ----- *)
(* cela peut devenir 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) *)
(*****)

```

$\langle \mathcal{FIN} \rangle$  TP Robinson. Corrigé.