Projet

Projet : la méthode des tableaux.

1 Avant-propos

Les évaluations de TP, notamment ce projet, sont des examens universitaires comptant pour la délivrance du diplôme national de licence.

À ce titre, toute fraude ou plagiat entraine la rédaction d'un procès verbal à l'attention de la section disciplinaire de l'université, qui peut infliger, en plus de la nullité de l'épreuve (note de 0), une sanction allant de l'avertissement à l'exclusion définitive de tout établissement public d'enseignement supérieur.

Ainsi, si vous utilisez un morceau de code que vous n'avez pas produit, vous devez indiquer très clairement sa source (site internet, camarade de promotion, *etc.*) : vous ne serez pas accusé de plagiat, mais ce morceau de code ne sera pas compté dans votre note finale.

2 Modalités de réalisation et de rendu

Le projet est à réaliser par groupes de **deux étudiants maximum**. Vous devrez rédiger un rapport de projet contenant notamment les choix de programmation ou les difficultés rencontrées et les solutions apportées.

Vous devrez rendre le projet (rapport et code source dans une archive) pour le lundi 15 mai au plus tard, en déposant celui-ci sur universitice dans le dépôt correspondant. Une soutenance de 15 minutes par binôme aura lieu à partir du lundi 22 mai afin de vous interroger sur votre réalisation : vous serez interrogés au tableau sur l'application de l'algorithme (détaillé dans la suite) sur une formule donnée, puis devrez expliquer les différentes étapes de cet algorithme dans votre code.

3 Introduction

La méthode des tableaux est une solution simple permettant de déterminer la satisfaisabilité d'une formule sans nécessairement construire l'intégralité d'une table de vérité. La formule est traitée en simulant la construction d'une forme clausale **disjonctive** (et non conjonctive) afin de déterminer si chaque clause obtenue est contradictoire. Si une clause (**conjonctive**) est satisfaisable, alors la formule l'est également.

Le but de ce projet est d'implanter cet algorithme en OCaml ainsi qu'une méthode permettant de tester empiriquement vos résultats à l'aide de formules construites (pseudo-)aléatoirement.

Dans la suite sont détaillés les différents algorithmes que vous devrez mettre en œuvre.

4 Génération aléatoire d'une formule

La génération aléatoire d'une formule peut se réaliser en considérant la méthode récursive suivante.

Chercher à construire aléatoirement une formule avec n opérateurs et des atomes dans Σ, c'est :

- Si n vaut 0, renvoyer un atome de Σ choisi aléatoirement ;
- Si n vaut 1, choisir aléatoirement un des cas suivants :
  - un opérateur nullaire ;
  - un opérateur unaire et un atome de Σ ;
  - un opérateur binaire et deux atomes de Σ ;
- Sinon, choisir aléatoirement un des cas suivants :
  - un opérateur unaire et une formule aléatoire avec $(n-1)$ opérateurs ;
  - un opérateur binaire, un entier aléatoire k compris entre 0 et $(n-1)$ inclus, et deux formules aléatoires avec respectivement k et $(n-k-1)$ opérateurs.

Vous devrez utiliser le module `Random`, en vous aidant notamment de l'Annexe A.

5 Méthode des tableaux

5.1 Idée de la méthode

La méthode des tableaux est basée sur la construction d'une structure arborescente dont les nœuds sont des formules de plus en plus petites au fur et à mesure des étapes, déterminées comme des formules à traiter les unes après les autres. Ces formules sont construites en éliminant les opérateurs à la racine des arbres syntaxiques, et leur traitement se réalise séquentiellement en barrant les formules déjà traitées et en réalisant des opérations particulières jusqu'à ce qu'il n'y ait plus de formules à traiter :

— en cas de conjonction ou d'une négation d'une disjonction, les sous-formules sont ajoutées successivement dans des nœuds successeurs, l'une sous l'autre ;
— en cas de disjonction ou d'une négation de conjonction, une fourche se crée dans l'arbre, et on ajoute les deux sous-formules l'une à côté de l'autre, dans deux sous-arbres différents ;
— pour un autre opérateur, une transformation s'opère, simulant la conversion de l'opérateur en combinaison de négation, de conjonction et de disjonction.

Si un littéral est présent, il suffit de chercher parmi ses prédécesseurs s'il apparait d'une façon opposée. Dans ce cas, il y a une contradiction trouvée le long de cette branche, qui est dite **fermée**. Une branche est également fermée lorsqu'elle contient $\bot$.

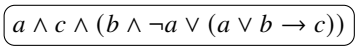
Si toutes les formules ont été traitées sans trouver de contradiction, la branche est dite **ouverte** et la formule est satisfaisable.

Remarquons que la présence de $\top$ ne modifie pas le caractère fermé ou ouvert d'une branche.

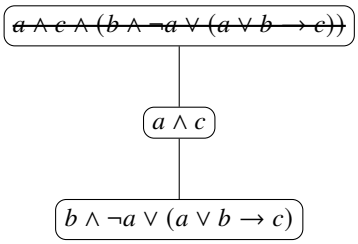
5.2 Exemples

Considérons la formule $F = a \wedge c \wedge (b \wedge \neg a \vee (a \vee b \to c))$.

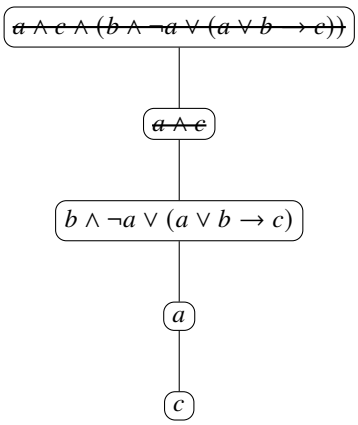
Initialement, la formule F est positionnée à la racine d'un arbre.



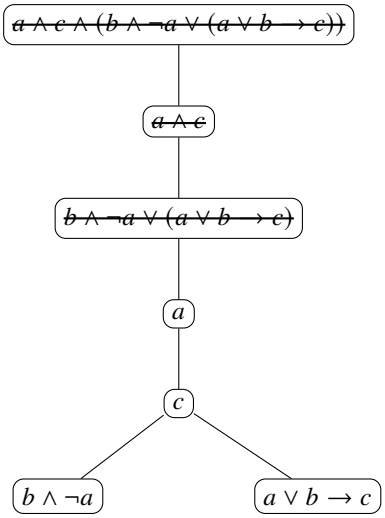
Étant une conjonction, on traite cette formule (graphiquement en la barrant) puis on ajoute ses deux sous-formules, $a \wedge c$ et $b \wedge \neg a \vee (a \vee b \to c)$, l'une sous l'autre, pour les traiter successivement.



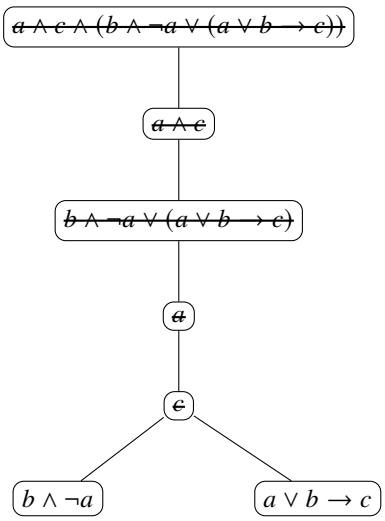
La formule suivante à traiter est $a \wedge c$, qui est aussi une conjonction. Ainsi, on la traite en ajoutant ses deux sous-formules a et c pour les traiter successivement.



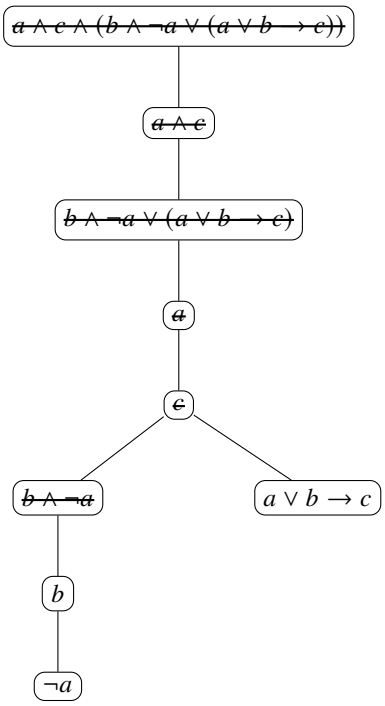
Vient alors une disjonction, la formule $b \wedge \neg a \vee (a \vee b \rightarrow c)$: une fourche est créée en ajoutant les deux sous-formules, $b \wedge \neg a$ et $a \vee b \rightarrow c$, dans des sous-arbres différents.



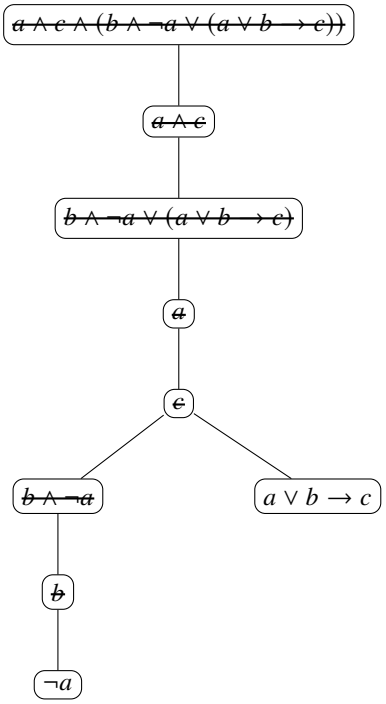
Les formules suivantes, les littéraux a et c, ne participent pas à la création de nouveaux nœuds. Ici, il s'agit de vérifier qu'il n'y a pas eu de littéraux opposés rencontrés plus tôt, ce qui est bien le cas.



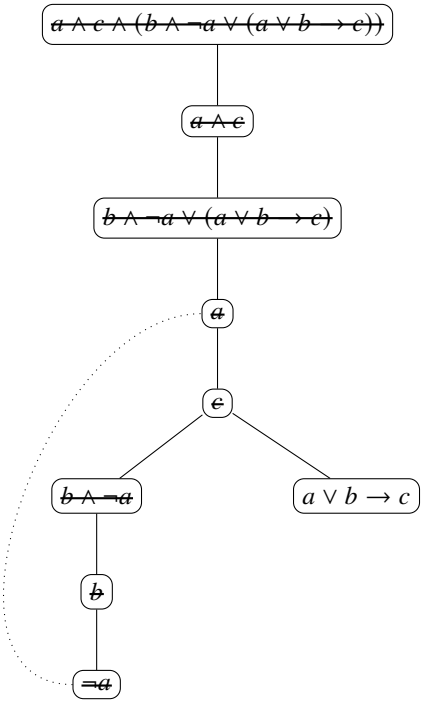
La formule suivante est une conjonction, $b \wedge \neg a$. On ajoute les deux sous-formules successivement.



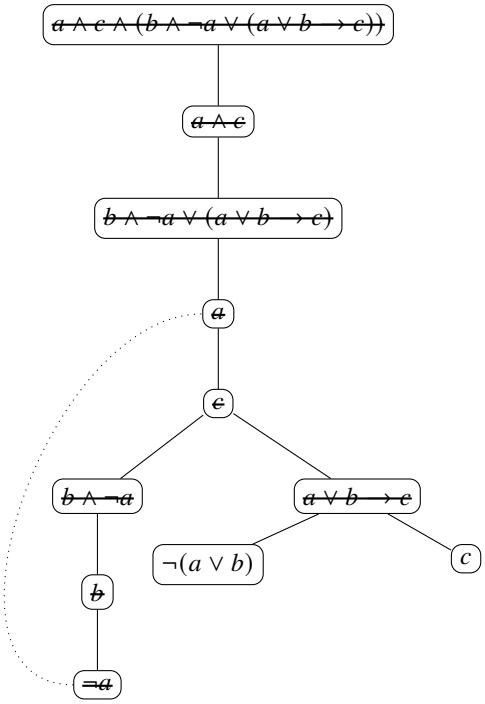
La formule suivante est un atome, b, rencontré pour la première fois.



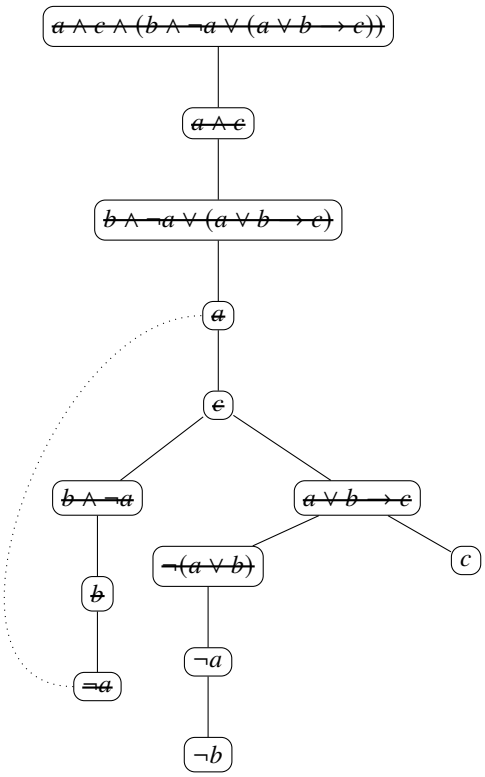
On rencontre alors le littéral $\neg a$. Comme le littéral opposé a a déjà été rencontré, on en déduit que la branche est contradictoire (on a rencontré a et $\neg a$, ce qui donne nécessairement une contradiction) et cette branche est fermée. Cette fermeture est représentée graphiquement comme un lien vers le littéral complémentaire.



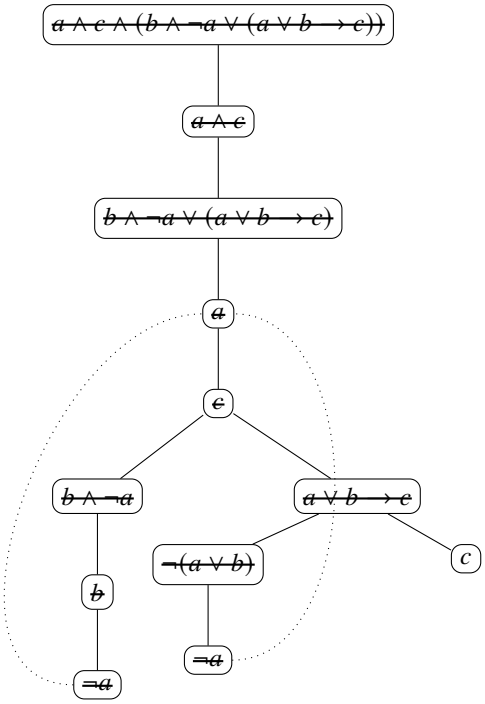
On passe alors à la branche suivante, de racine $a \vee b \rightarrow c$. La formule étant une implication, il suffit de la traiter à l'aide de la conversion classique, $a \vee b \rightarrow c \equiv \neg(a \vee b) \vee c$, comme une disjonction, créant une nouvelle fourche. On place les deux sous-formules, $\neg(a \vee b)$ et c, dans deux sous-arbres différents.



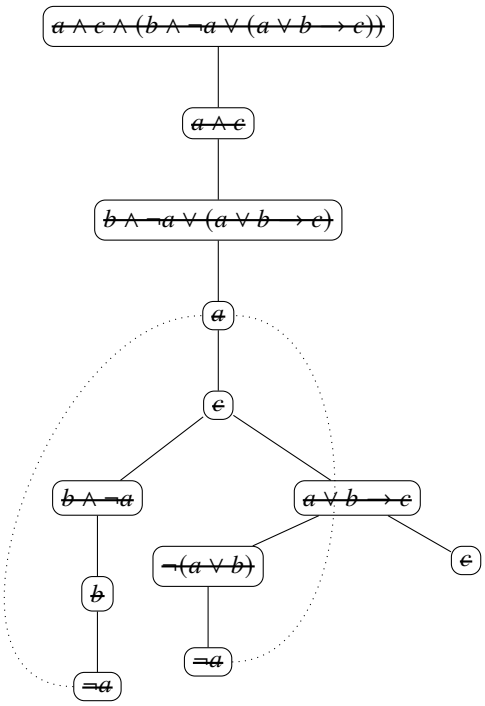
La formule à traiter, $\neg(a \vee b)$, est une négation d'une disjonction. On la traite à l'aide de l'équivalence $\neg(a \vee b) \equiv \neg a \wedge \neg b$, c'est-à-dire comme une conjonction. On place les deux sous-formules $\neg a$ et $\neg b$ successivement dans le même sous-arbre.



Une nouvelle fois, le littéral $\neg a$ nous permet de fermer la branche. Une contradiction ayant été trouvée, il n'est pas nécessaire de continuer sur cette branche et le littéral $\neg b$ est éliminé, son traitement étant inutile.

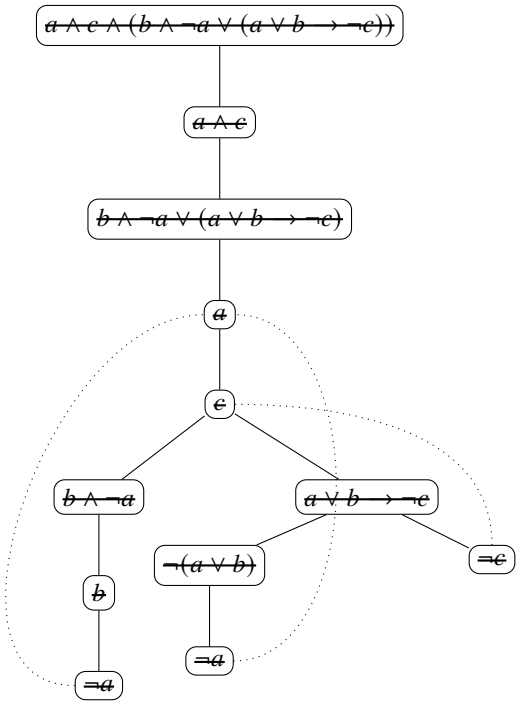


Il reste un dernier élément à traiter, le littéral c, déjà rencontré.



Cette dernière branche est ouverte, et la formule est ainsi satisfaisable : il suffit en effet de considérer une interprétation telle que les atomes a et c soient vrais, obtenus par le parcours le long de cette branche ouverte.

En considérant une autre formule, $a \wedge c \wedge (b \wedge \neg a \vee (a \vee b \rightarrow \neg c))$, la méthode des tableaux permet de montrer qu'il s'agit d'une contradiction, toutes les branches étant fermées.



5.3 Idée d'implantation

En terme d'implantation, il n'est pas nécessaire de construire une structure arborescente. Il est suffisant de considérer une branche en cours d'étude comme une liste de formules, dont la tête est la prochaine formule à traiter :

- s'il s'agit d'une conjonction, on ajoute les deux sous-formules dans la liste à traiter ;
- s'il s'agit d'une disjonction, on traite les deux cas en parallèle, en ajoutant une des deux sous-formules pour chaque cas dans la liste des formules à traiter ;
- s'il s'agit de $\bot$, on s'arrête en fermant la branche ;
- s'il s'agit de $\top$, on peut l'éliminer et continuer ;
- s'il s'agit d'un littéral, on doit vérifier qu'on ne l'a pas déjà rencontré sous sa forme opposée ;
- s'il s'agit d'une autre opération, on utilise les conversions nécessaires.

Ainsi, il peut être intéressant de réaliser une fonction auxiliaire à deux paramètres :

- une liste de littéraux déjà rencontrés ;
- une liste de formules à traiter.

En exemple, considérons de nouveau la formule $F = a \wedge c \wedge (b \wedge \neg a \vee (a \vee b \rightarrow c))$. Nous allons représenter d'une façon arborescente l'évolution des paramètres de la fonction auxiliaire au fur et à mesure des appels.

On maintient une liste de littéraux déjà rencontrés (liste vide pour le moment), et la liste des formules à traiter (la liste $[F]$).

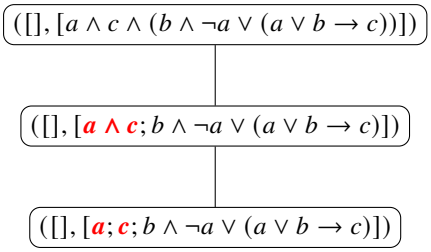
$([], [a \wedge c \wedge (b \wedge \neg a \vee (a \vee b \rightarrow c))])$

La tête de liste est une conjonction, on la remplace ainsi par ses deux sous-formules.

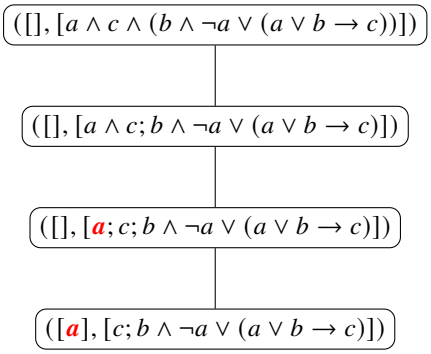
$([], [\mathbf{a \wedge c \wedge (b \wedge \neg a \vee (a \vee b \rightarrow c))}])$

$([], [\mathbf{a \wedge c}; \mathbf{b \wedge \neg a \vee (a \vee b \rightarrow c)}])$

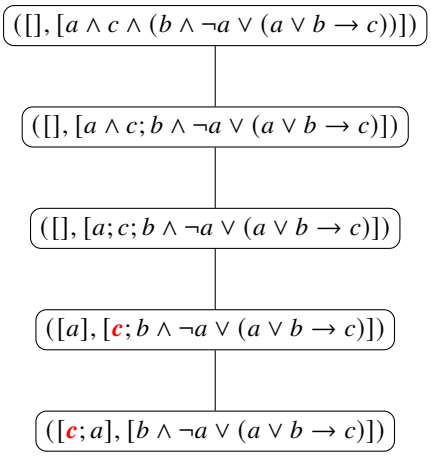
La tête de liste est encore une conjonction. La même opération est réalisée.



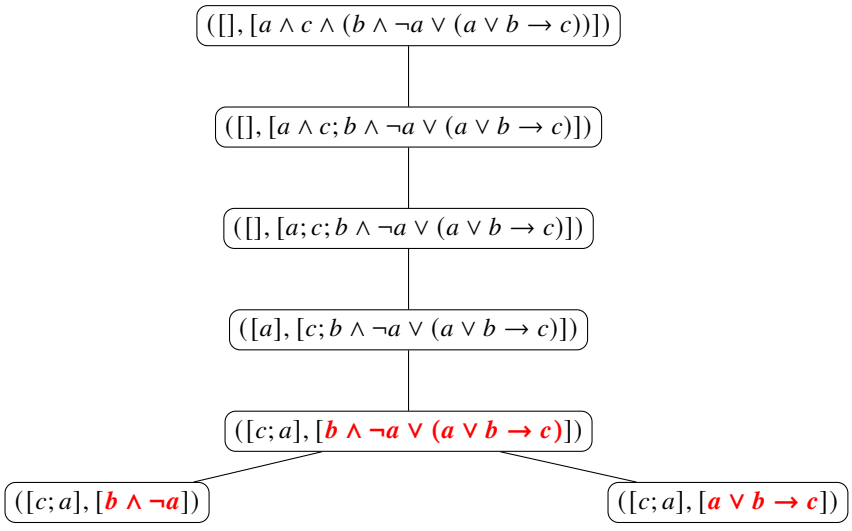
La tête de liste est un littéral, a. On le bascule dans la liste des littéraux rencontrés.



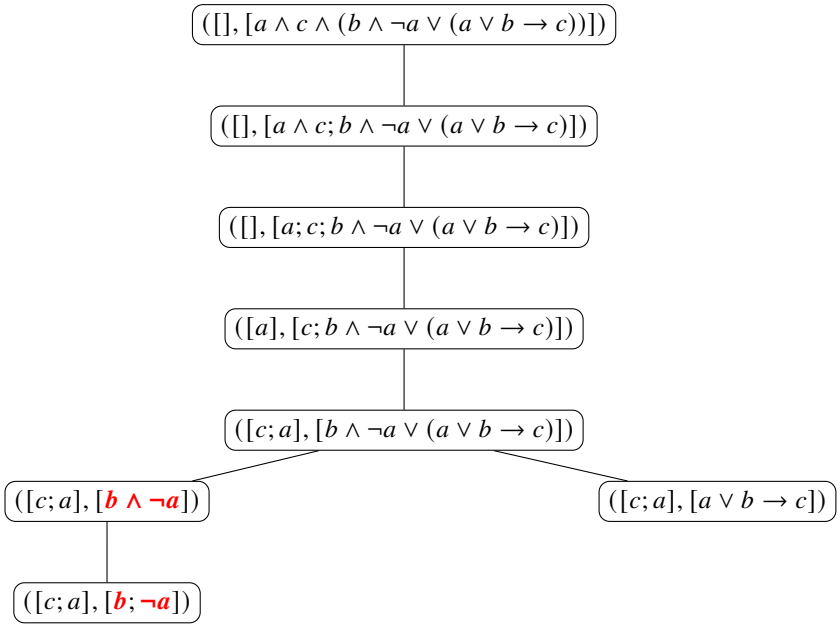
La tête de liste est de nouveau un littéral. On le bascule comme précédemment dans la liste des littéraux rencontrés.



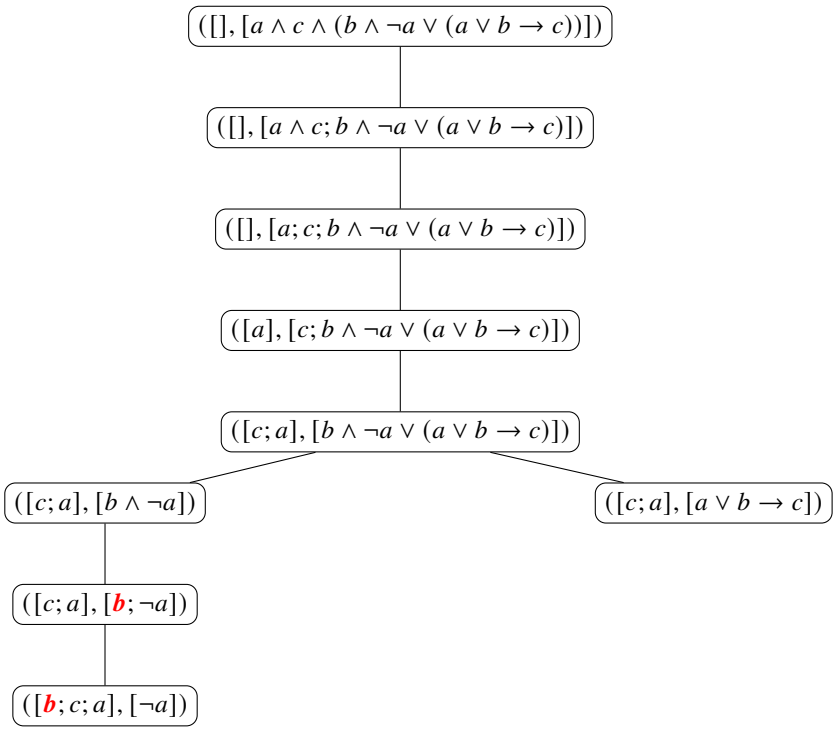
On rencontre une disjonction. Deux appels récursifs à la fonction auxiliaires doivent être réalisés, chacun avec une sous-formule en tête de la liste des formules à traiter.



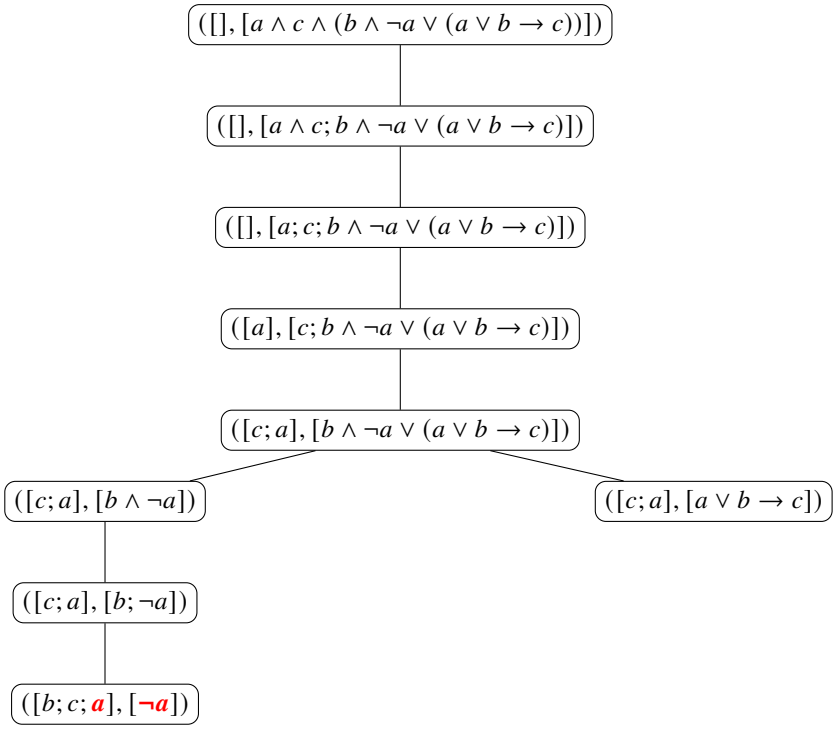
La formule en tête est une conjonction, on ajoute chacune des deux sous-formules.



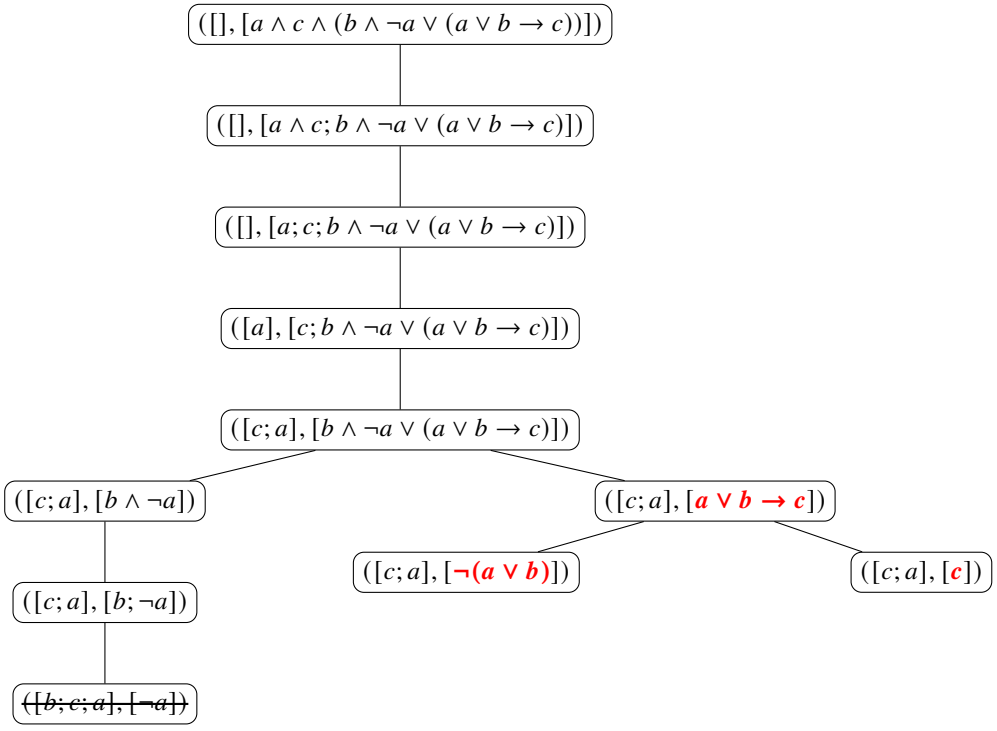
Le littéral b, rencontré pour la première fois, est transféré dans la première liste, celle des littéraux rencontrés.



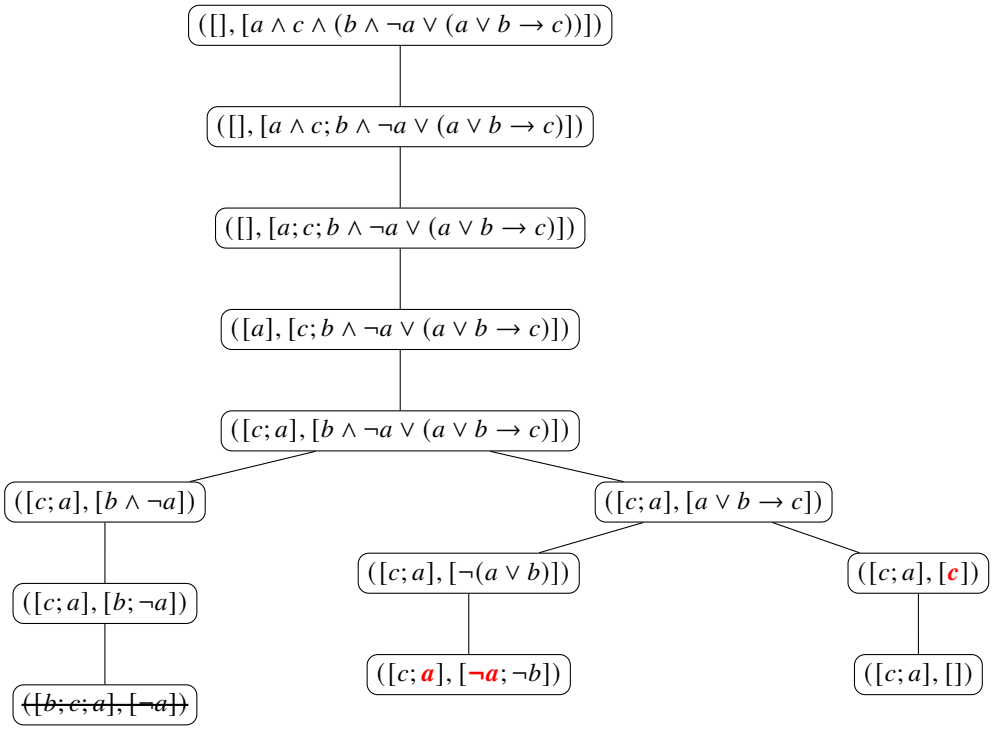
Le littéral négatif $\neg a$ est le prochain à traiter. Comme le littéral opposé a a déjà été rencontré, la branche est déclarée contradictoire et on passe à la suivante.



Comme précédemment, la formule $a \vee b \rightarrow c$ est traitée à l'aide de son équivalente $\neg(a \vee b) \vee c$, donnant lieu à deux appels récursifs, chacun avec une sous-formule en tête de la liste des formules à traiter.



L'application successive des étapes permet de fermer une nouvelle branche, puis d'atteindre dans la dernière branche une liste de formules à traiter vide, qui, n'ayant pas rencontré de contradiction due à des littéraux complémentaires, nous permet de conclure sur la satisfaisabilité de la formule de départ. La liste des littéraux rencontrés permet de déduire une condition suffisante pour une évaluation positive de la formule de départ.



6 Test empirique de validité des résultats

La méthode des tableaux décrite dans la section précédente sera implantée pour résoudre les mêmes problèmes de satisfaisabilité que ceux de l'Activité 2 (algorithme de Quine).

Ainsi, vous devrez notamment implanter des fonctions permettant :

— de renvoyer un témoin de satisfaisabilité lorsque celui-ci existe (c'est-à-dire une valeur de type `(string * bool) list option`);

— de renvoyer la liste de tous les témoins obtenus lors des parcours des branches (c'est-à-dire une valeur de type `(string * bool) list list`).

Pour valider vos résultats, il vous est demandé de transformer une valeur `(string * bool) list` en une interprétation (une valeur de type `string -> bool`). Pour cela, il vous suffira, pour toute `string s`, de renvoyer :

— un booléen `b` si le couple `(s, b)` est présent dans la liste ;

— un booléen aléatoire sinon.

Une fois cette conversion faite, vous pourrez vérifier que les témoins renvoyés se transforment en interprétation permettant d'évaluer la formule comme vraie.

7 Structure des ressources de départ

La structure du projet et l'utilisation de `dune` sont imposées.

Les formules utilisent les opérateurs classiques, mais également les opérateurs "ou exclusif" et "égalité".

Le projet est constitué d'un dossier `Proposition` contenant trois fichiers :

— un fichier `Formule.ml` contenant la définition du type des formules ainsi que des fonctions utilitaires ;

— un fichier `RandomFormule.ml` contenant une fonction de génération aléatoire de formules ;

— un fichier `Tableaux.ml` contenant les fonctions de tests de satisfaisabilité selon la méthode des tableaux ;

— un fichier `Test.ml` contenant une fonction réalisant des tests de validation sur des formules aléatoires.

Le fichier `Formule.ml` contient les éléments de manipulation de formules booléennes dont le type est imposé. Vous devrez implanter les fonctions suivantes en respectant les signatures fournies :

— `string_of_formule` transformant une formule en chaine de caractères,

— `eval` évaluant une formule à partir d'une interprétation donnée.

Le fichier `RandomFormule.ml` contient la fonction `random_form` prenant deux paramètres, une liste de `string` (l'alphabet de la formule) et un entier `n`, et renvoyant une formule aléatoire avec `n` opérateurs sur l'alphabet donné.

Le fichier `Tableaux.ml` contient votre implantation de la méthode des tableaux. Plus précisément, les fonctions suivantes sont attendues :

- `tableau_sat` renvoyant `true` si et seulement si la formule donnée en paramètre est satisfaisable, booléen déterminé par la méthode des tableaux ;
- `tableau_ex_sat` renvoyant un témoin (s'il existe, type optionnel) de la satisfaisabilité de la formule donnée en paramètre, obtenue en parcourant une branche générée par la méthode des tableaux ;
- `tableau_all_sat` renvoyant la liste des témoins de la satisfaisabilité de la formule donnée en paramètre, obtenue en parcourant toutes les branches générées par la méthode des tableaux.

Le fichier `Test.ml` contient les fonctions suivantes :

- `to_alea_inter` prenant en paramètre un témoin (une valeur de type `(string * bool) list`) et renvoyant une interprétation ;
- `test_valid`, prenant un paramètre entier `n` et réalisant :
  - la génération d'une formule aléatoire `f` avec `n` opérateurs sur l'alphabet `[ "a"; "b"; "c"; "d" ]` ;
  - l'affichage de cette formule sur la sortie standard ;
  - la validation des résultats des fonctions `tableau_ex_sat` et `tableau_all_sat` de `Tableaux.ml`, en vérifiant que chaque témoin obtenu peut être converti en une interprétation évaluant la formule `f` comme vraie et retournant `true` si tous les résultats sont corrects, `false` sinon.

Une fois ces éléments implantés, vous pourrez les tester à l'aide du fichier `Test.ml`, en lançant la commande `dune utop` depuis le dossier racine du projet et en chargeant les modules nécessaires :

```
open Proposition;;
open Formule;;
open RandomFormule;;
open Test;;
test_valid 10;;
```

8 Barème indicatif

1. Code (13 points) :
   (a) Module `Formule` (2 points) :
   - représentation en chaine de caractères (1 point)
   - fonction d'évaluation (1 point)

   (b) Module `RandomFormule` (2 points) :
   - génération aléatoire d'une formule (2 points)

   (c) Module `Tableaux` (6 points) :
   - étude de la satisfaisabilité (1 point)
   - recherche d'un témoin de la satisfaisabilité (2 points)
   - recherche de tous les témoins de la satisfaisabilité (3 points)

   (d) Module `Test` (3 points) :
   - conversion d'un témoin en interprétation (1 point)
   - test de validation (2 points)
2. Rapport (4 points)
3. Soutenance (3 points)

Il s'agit d'un barème indicatif "de départ". À celui-ci s'ajouteront différentes pénalités, notamment si :

- votre code rencontre des erreurs de compilation ;
- votre code émet des avertissements à la compilation ;
- des contre-exemples sont détectés,

ainsi que des bonus, notamment si :

— votre rapport est écrit en LaTeX,
— vous codez d'une façon propre, lisible, élégante et efficace,
— vous gérez des opérateurs supplémentaires dans vos formules.

A Le module Random

Le module `Random` permet la génération (pseudo-) aléatoire de différents types de base. L'expression `Random.int b` permet de générer un nombre entre 0 (compris) et `b` (exclus), où `b` est un entier. Certaines fonctions utilisent le type `unit`, type ne possédant qu'une seule valeur, la valeur `()`. Cela est fait pour distinguer les constantes des fonctions : la génération aléatoire se doit d'être réalisée par des fonctions. Par exemple, comparez les éléments suivants :

```
let x = Random.int 30;;
let y () = Random.int 30;;
x;;
x;;
y ();;
y ();;
```

Ainsi, pour réaliser une fonction de génération d'un couple de caractères, comparez

```
let random_chars =
  (char_of_int (Random.int 256), char_of_int (Random.int 256));;
let random_chars' () =
  (char_of_int (Random.int 256), char_of_int (Random.int 256));;
random_chars;;
random_chars;;
random_chars' ();;
random_chars' ();;
```

Enfin, pour correctement utiliser les générateurs, il faut tout simplement initialiser le germe en utilisant `Random.self_init ()`, initialisant celui-ci d'une façon (pseudo-) aléatoire.