Université de Montpellier - Master informatique Janvier 2024

Théorie des bases de données et de connaissances (HAI933I)

Correction partie "modèles stables"

Après le retour de Marie-Laure qui a assuré la surveillance de l'examen, j'étais inquiet pour cette partie modèles stables. Inquiétude justifiée après correction de vos copies. Vous trouverez ici une correction de cette partie de l'examen, avec, dans les cadres jaunes, la correction à proprement parler (toujours un peu verbeuse, vous n'avez pas besoin d'en faire autant dans vos copies) et, dans les cadres verts mes remarques diverses, que ce soit sur les erreurs rencontrées dans vos copies, des rappels de cours, des éléments de méthodologie ou ce qui pourrait avoir été fait dans vos brouillons.

1 Modèles stables (7 points)

L'objectif de cette partie de l'examen est de tester la satisfiabilité d'une formule de 2QBF en utilisant les modèles stables (vous utiliserez la syntaxe des règles existentielles avec négation (REN) vue en cours : en particulier, vous n'aurez pas droit à la disjonction en tête de règle).

Remarque préliminaire: *Les questions 1 à 7 ne nécessitent pas de comprendre 2QBF pour y répondre. Je vous conseille cependant fortement de lire la section 1.1 (introduction à 2QBF) pour comprendre là où les questions doivent vous mener.*

Je veux bien admettre que 2QBF est un peu compliqué à comprendre dans le temps qui vous était imparti pour cet examen. Mais dans les questions 1 à 6 (qui comptaient pour plus de la moitié des points), il n'était question que des modèles stables qui encodent toutes les valuations possibles d'un ensemble de variables booléennes. Et ça, en M2 Informatique, ça ne devrait poser aucun problème conceptuel...

Mention spéciale à l'étudiant qui, jugeant mon sujet "incompréhensible", a décidé d'écrire ce qu'il voulait. SPOLILER ALERT: ça énerve le correcteur et ça ne sert à rien.

Rappels de logique des propositions: une formule F de logique des propositions est *satisfiable* si il existe une valuation σ de ses variables (ou symboles propositionnels) telle que $\sigma(F)$ s'évalue à `true`. Elle est *valide* si, pour toute valuation σ de ses variables, $\sigma(F)$ s'évalue à `true`.

1.1 Une introduction à 2QBF

Une formule de 2QBF est de la forme $\exists\vec{x}\forall\vec{y}\,F(\vec{x},\vec{y})$ où $F(\vec{x},\vec{y})$ est une formule propositionnelle dont les variables (ou symboles propositionnels) sont celles de $\vec{x}\cup\vec{y}$. Une telle formule est satisfiable ssi il existe une valuation des variables de $\vec{x}$ dans {`true`, `false`} telle que pour toute valuation des variables de $\vec{y}$, la valeur de vérité de la formule $F(\vec{x},\vec{y})$ est `true`. En d'autres termes, cette formule est satisfiable ssi il existe une valuation σ des variables de $\vec{x}$ dans {`true`, `false`} telle que la formule $\sigma(F(\vec{x},\vec{y}))$ est valide.

Sans perte de généralité, nous considérons par la suite que la formule $F(\vec{x},\vec{y})$ est une 3DNF, c'est à dire une disjonction de conjonctions contenant chacune au plus 3 littéraux (variable x ou sa négation).

Exemples: La formule 2QBF $F=\exists x\exists y\forall z\,F'$ avec $F'=(x\wedge\neg y\wedge z)\vee(x\wedge\neg z)$ se lit "il existe des valeurs booléennes x et y telles que, pour toute valeur booléenne z, la formule F' est vraie". Soit la valuation $\sigma=\{x:\texttt{true},y:\texttt{false}\}$. On a $\sigma(F')=(\texttt{true}\wedge\texttt{true}\wedge z)\vee(\texttt{true}\wedge\neg z)$, qui est équivalente à $z\vee\neg z$ qui est valide. La formule F est donc satisfiable.

Soit la formule 2QBF $G=\exists x\forall y\,G'$ avec $G'=(x\wedge y)\vee(\neg x\wedge\neg y)$. Considérons les valuations $\sigma_1=\{x:\texttt{true}\}$ et $\sigma_2=\{x:\texttt{false}\}$. On a $\sigma_1(G')=(\texttt{true}\wedge y)\vee(\texttt{false}\wedge\neg y)$ qui est équivalente

à $y \vee \texttt{false}$, équivalente à y, qui est invalide (c'est-à-dire non valide). Nous avons également $\sigma_2(G') = (\texttt{false} \wedge y) \vee (\texttt{true} \wedge \neg y)$ qui est équivalente à $\texttt{false} \vee \neg y$, équivalente à $\neg y$, également invalide. Comme aucune substitution par une valuation de x ne produit de formule valide, alors la formule G est insatisfiable.

Au cours de cet exercice, vous devrez étudier (sans l'écrire explicitement) un *traducteur* qui, à partir d'une *formule* 2QBF $Q = \exists\vec{x}\forall\vec{y}\,F(\vec{x},\vec{y})$, génère un *programme* $\Pi(Q)$ écrit avec des règles existentielles avec négation (REN). Ce traducteur devra respecter la propriété suivante: la valuation $\sigma = \{x_1 : b_1, \ldots, x_k : b_k\}$ (où les x_i sont les variables quantifiées existentiellement de Q et les b_i sont les booléens `true` ou `false`) est telle que $\sigma(F(\vec{x},\vec{y}))$ est valide si et seulement si il existe un modèle stable de $\Pi(Q)$ dont les prédicats ayant pour nom de prédicat *val* sont exactement $val(x_1, b_1), \ldots, val(x_k, b_k)$. Ainsi, la formule Q sera satisfiable ssi le programme $\Pi(Q)$ admet un modèle stable.

1.2 Valuation des variables existentielles

Cette partie du travail vise à générer toutes les valuations possibles pour les variables quantifiées existentiellement d'une 2QBF.

Question 1 Écrire un programme Π^1 dont les modèles stables contiennent toutes les valuations possibles pour les variables propositionnelles x_1 et x_2. Pour chacune de ces deux variables (et ici nous considérons x_1), chaque modèle stable devra contenir soit l'atome $val(x_1, \texttt{true})$, soit l'atome $val(x_1, \texttt{false})$, mais pas les deux. Remarquons au passage que les variables propositionnelles de 2QBF écrites avec des minuscules deviennent des constantes en REN.

Je jugeais l'écriture de ce programme Π^1 simplissime. C'est pourquoi les questions suivantes (2 à 6) reposaient sur ce programme que vous deviez écrire. Malheureusement, ceux qui ont bloqué sur cette question n'ont pas pu continuer. C'est dommage, mais je ne suis pas persuadé qu'il s'agisse d'une erreur de conception du sujet (en maths, si vous ne savez pas faire un calcul de dérivée, vous êtes mal barré pour toute l'étude de fonction).

```
%% La variable x1 est évaluée à true ou false mais pas les 2
  val(x1, true) :- not val(x1, false). % R1
  val(x1, false) :- not val(x1, true). % R2
  ! :- val(x1, true), val(x1, false).    % R3
%% Même chose pour la variable x2
  val(x2, true) :- not val(x2, false). % R4
  val(x2, false) :- not val(x2, true). % R5
  ! :- val(x2, true), val(x2, false).    % R6
```

Il suffisait d'écrire ici 2 disjonctions binaires indépendantes avec des REN, exercice que nous avons fait et refait tout au long du cours.

Petite typologie des erreurs rencontrées dans vos copies, quand cette copie n'était pas vide:

1. il y a ceux qui ont mis tous les atomes `val(x1, true), val(x1, false), val(x2, true), val(x2, false).` menant à un unique modèle qui ne veut rien dire, et devrait être absurde si vous aviez pensé aux contraintes `R3` et `R6`.

2. il y a ceux qui ne connaissent pas la syntaxe des REN, et qui mettent dans le corps de la règle une formule quelconque (en imbricant des disjonctions, des conjonctions, et des négations dont on ne sait pas trop si c'est la négation classique ou celle par l'échec); ou qui mettent dans la tête de la règle de la disjonction (pourtant expressemment interdit dans le sujet) ou même de la négation (et la je n'avais même pas songé à l'interdire).

3. il y a ceux qui m'ont écrit des REN à peu près syntaxiquement correcte, mais on voit qu'ils n'ont aucune idée de comment on écrit une disjonction (manque de travail certain sur le cours). J'ai pu voir par exemple des règles compliquées mais sans négation dans le corps, ce qui ne peut donc mener qu'à un modèle stable, et ne répond donc pas à la question,

ou des règles où la valuation de `x1` dépend de celle de `x2`, ce qui ne peut manquer d'être faux.

4. parmi ceux qui ont fait quelque chose de ressemblant à la correction, on rencontre très souvent 2 problèmes:

   (a) la confusion entre une variable booléenne (une chose qui peut prendre 2 valeurs, `true` et `false`), et le nom que l'on donne à cette variable, qui est un identificateur, c'est à dire, quand on le manipule dans notre programme, une constante. Or si on écrit notre programme avec des variables, on se retrouve par exemple avec `val(X1, true) :- not val(X1, false).`, ce que nous avons interdit (la variable apparaît dans le corps négatif et la tête, mais pas dans le corps positif). Il fallait donc travailler avec les noms de variables (des constanes, donc), ce qui fonctionne très bien.

   (b) vous avez tous eu peur d'écrire des REN avec des corps positifs vides ! Pourtant la REN `val(x1, true) :- not val(x1, false).` est parfaitement définie, puisqu'il n'y a pas de variable (donc pas de variable partagée uniquement entre le corps négatif et la tête). Pour remédier à cette angoisse, j'ai vu plusieurs solutions, plus ou moins gênantes.

      i. `val(x1, true) :- x1, not val(x1, false).` pose un gros problème car `x1` n'est pas un atome, donc ce n'est pas une REN. Pour que ça ait quand même un "sens", il fallait aussi rajouter le "fait" `x1` (qui n'est toujours pas un atome) afin que la "règle" soit "déclenchable". Certains ont également vu ici une solution au "problème" 4.*a*, et on écrit la règle avec une variable `val(X, true) :- X, not val(X, false).`, où le `X` peut se lier aux "atomes" `x1` et `x2` présents dans la base de faits.

      ii. `val(x1, true) :- varname(x1), not val(x1, false).` repose sur la même idée, mais est cette fois-ci syntaxiquement correct. Pour que ça marche, il fallait rajouter `varname(x1)` dans la base de faits. Notons que cette fois-ci, cette modélisation aurait permis d'écrire les règles avec des variables (et permettait de passer de 6 règles à 3 règles), mais personne ne l'a fait. On aurait ainsi eu:

```
varname(x1). varname(x2). %% déclaration des variables
%% 3 règles quel que soit le nombre de variables
  val(X, true) :- varname(X), not val(X, false).
  val(X, false) :- varname(X), not val(X, true).
  ! :- val(X, true), val(X, false).
```

Pour ceux qui se situent dans les cas 1–3, la suite de l'exercice était sérieusement compromise. Pour le cas 4, j'ai compté juste lorsque les réponses étaient cohérentes dans les exercices suivants.

Question 2 Remarquons que le programme Π^1 est un programme propositionnel (sans variable). Soit l'ensemble d'atomes $E = \{val(x_1, \texttt{false})\}$. En utilisant le programme réduit $\Pi^1_{|E}$ et la définition par point fixe, montrez que E n'est pas un modèle stable de Π^1.

La remarque préliminaire aurait du faire comprendre à ceux qui utilisent des variables dans les règles qu'ils n'étaient pas sur la bonne piste. Dans ce cas, il fallait bien sûr pour avoir les points commencer par un grounding du programme, ce que personne n'a fait.

Le programme réduit $\Pi^1_{|E}$ est le suivant:

```
%% R1 est supprimée car val(x1, false) est dans E
  val(x1, false) :- .                    % obtenue à partir de R2
  ! :- val(x1, true), val(x1, false).    % R3 ne change pas
  val(x2, true) :- .                     % obtenue à partir de R4
  val(x2, false) :- .                    % obtenue à partir de R5
  ! :- val(x2, true), val(x2, false).    % R6 ne change pas
```

Si on calcule la saturation $(\Pi^1_{|E})^*$ de ce programme, on a:

$$(\Pi^1_{|E})^* = \{val(x1, false), val(x2, true), val(x2, false), !\}$$

Comme $(\Pi^1_{|E})^* \neq E$ (et de plus il contient absurde !), alors E n'est pas un modèle stable de Π^1.

Curieusement, cette question n'a été que peu traitée (et rarement bien), même parmi ceux qui se trouvent dans le cas 4 de la question 1. Pourtant, c'est un exercice qui a été fait plusieurs fois en cours. Une réponse correcte (n'utilisant pas la méthode demandée) dans le cas d'une réponse de type 4 à la question 1 a été de dire que comme E ne contenait pas les faits contenus dans le programme, il ne pouvait pas être un modèle stable. J'ai généreusement accordé les points.

Question 3 Soit Q une 2QBF, nous disposons d'une fonction `varexist`(Q) qui retourne l'ensemble des variables quantifiées existentiellement dans Q, d'une fonction `renprogram`() qui crée un nouveau programme (vide) de REN et d'une fonction `addrule`(P, R) qui ajoute une règle, dont la représentation par string est R, à un programme P. Le traducteur qui génère à partir d'une 2QBF Q le programme (en REN) $\Pi^1(Q)$ dont les modèles stables contiennent toutes les valuations possibles des variables existentielles de Q (voir question 1) devrait ressembler à :

```
myprog = renprogram()
rule1 =
rule2 =
rule3 =
Pour nomvar in varexist(Q):
    addrule(myprog, rule1.format(nomvar))
    addrule(myprog, rule2.format(nomvar))
    addrule(myprog, rule3.format(nomvar))
    % j'utilise la méthode format de python
    % qui remplace {} dans une chaîne par son argument
    % par exemple "hello {}".format("world")
    % retourne "hello world"
return myprog
```

Écrire les 3 "patterns de règle" (`rule1`, ...) nécessaires pour compléter le traducteur. Ceci devrait beaucoup ressembler à votre réponse à la question 1.

```
rule1 = "val({}, true) :- not val({}, false)."
rule2 = "val({}, false) :- not val({}, true)."
rule3 = "! :- val({}, true), val({}, false)."
```

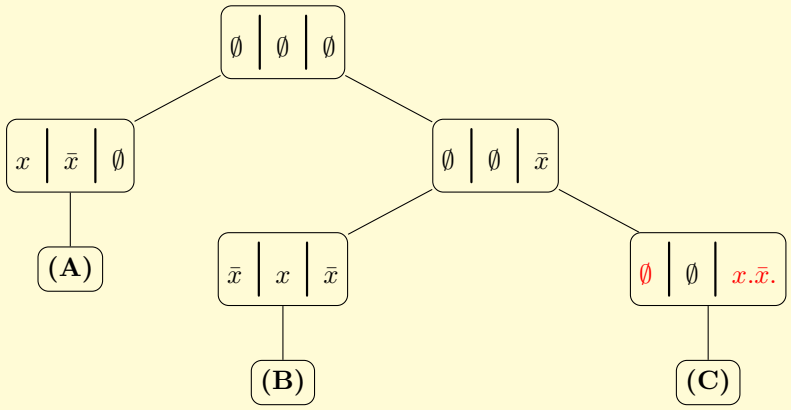
On voit bien que la réponse à cette question ressemble énormément à celle de la question 1 ... Curieusement, cette question a pourtant été beaucoup mieux traitée que la question 1. Et je me demande encore par quel miracle ceux qui ont écrit n'importe quoi à la question 1 (cas 1-3) ont donné une réponse à peu près correcte (cas 4) à cette question. Et, question subsidiaire: pourquoi ceux qui ont eu une inspiration subite pour cette question 3 ne se sont pas rendu compte qu'ils avaient fait n'importe quoi à la question 1 et ne l'ont pas corrigé ? Le mystère reste entier.

Question 4 Soit la 2QBF $Q = \exists x \forall y (x \wedge y)$. Représentez l'arbre ASPERIX obtenu en faisant tourner le programme REN $\Pi^1(Q)$ (que vous expliciterez) généré par le traducteur de la question 4. Quels sont les modèles stables de ce programme? Vous justifierez soigneusement votre réponse.

Le programme $\Pi^1(Q)$ généré est:

```
val(x, true) :- not val(x, false). # R1
val(x, false) :- not val(x, true). # R2
! :- val(x, true), val(x, false).  # R3
```

Si on fait tourner l'algorithme ASPERIX sur ce programme, on obtient (où on note x l'atome `val(x, true)` et $\bar{x}$ l'atome `val(x, false)`):



Le sommet **(A)** et l'autre fils de la racine sont générés à partir de la règle `R1`. On n'a pas besoin d'évaluer `R2` sur **(A)** car elle est immédiatement bloquée. Les sommets **(B)** et **(C)** sont obtenus en évaluant `R2` sur l'autre fils de la racine. La règle `R3` n'étant applicable nulle part, l'arbre est bien *complet*. Examinons maintenant les feuilles de cet arbre.

- **(A)** ne viole pas la contrainte du OUT, et satisfait toutes celles de MBT (il n'y en a pas). C'est donc un modèle sable qui contient l'atome `val(x, true)`.
- **(B)** ne viole pas l'unique contrainte OUT et satisfait l'unique contrainte MBT, c'est donc un modèle stable qui contient l'atome `val(x, false)`.
- **(C)** viole les 2 contraintes du MBT (il aurait suffit d'une), ce n'est pas un modèle stable.

Question à peu près bien traitée par ceux qui faisaient partie du cas 4 de la question 1. Pour ceux qui font partie des cas 1–3, c'était plus difficile. Une chose vue plusieurs fois, et qui m'énerve particulièrement: ceux qui m'ont donné à la question 1 des règles qui font n'importe quoi, mais qui trafiquent leur arbre pour qu'il donne (vu de loin) le résultat attendu. C'est de l'escroquerie, et ça n'a pas sa place en sciences. Je préfère qu'on me dise *"les modèles stables générés par ASPERIX ne correspondent pas à ce qui a été demandé, il y a donc une erreur dans ma modélisation ou mon utilisation de l'algorithme, mais je n'arrive pas à la corriger."*

Question 5 Si vous avez bien respecté les consignes de la question 1, votre programme $\Pi^1(Q)$ de la question 4 comporte une règle exprimant qu'une variable ne peut pas être valuée à la fois à `true` et à `false`. Cette règle est-elle utilisée dans la construction de votre arbre ASPERIX ? On pourrait penser que cette règle est inutile, et utiliser un traducteur ne générant pas cette règle dans le programme $\Pi^1_{v2}(Q)$. Montrez qu'on peut compléter ce programme avec des règles (ou des faits) de façon à obtenir un programme dont un modèle stable contient à la fois une valuation de x à `true` et une valuation de x à `false`. Vous justifierez soigneusement que ce programme répond bien à la question.

Nous avons développé à la question précédente un arbre ASPERIX complet dans lequel la règle `R3` n'était pas applicable. Elle est donc inutile dans *ce* programme. Mais imaginons le programme suivant, dans lequel on a enlevé `R3` mais rajouté deux faits:

```
val(x, true) :- not val(x, false). # R1
val(x, false) :- not val(x, true). # R2
```

```
val(x, true). val(x, false).       # faits rajoutés
```

En faisant tourner ASPERIX sur ce programme on obtient l'arbre complet (car les règles `R1` et `R2` sont immédiatement bloquées) suivant:

`val(x, true), val(x, false).`	$\emptyset$	$\emptyset$

Il y a un unique modèle stable, qui contient à la fois `val(x, true)` et `val(x, false)`, et dans ce cas la contrainte `R3` aurait été bien utile.

Presque tous les étudiants ont répondu, juré-craché et main sur le coeur, que cette règle de votre programme n'était pas évaluée dans l'arbre ASPERIX. Même, curieusement, ceux qui n'avaient pas mis cette règle dans le programme. La encore, on est à la limite de l'escroquerie.

Pour a deuxième partie de la question, la solution avait été donnée au cours de la discussion sur l'encadré du transparent 20, premier cours. Le contrexemple de la correction avait été évoqué à ce moment, et c'est celui qui a souvent été donné par les étudiants.

Question 6 Soit Q une 2QBF, et Π un programme quelconque contenant $\Pi^1_{v2}(Q)$ (voir question 5). Donnez une condition suffisante sur le programme Π permettant d'assurer que, pour chaque modèle stable de Π, il n'existe aucune variable existentielle de Q valuée à la fois à `true` et `false`. Vous pourrez vérifier à la fin du devoir que le programme final respecte cette condition, et donc que la règle interdisant la double valuation d'une variable était inutile.

Imaginons un arbre de recherche ASPERIX sur un programme contenant `R1` et `R2`. Pour éviter le problème identifié à la question 6, **(condition 1)** nous interdisons dans ce programme tout fait contenant un atome de prédicat `val`.

Comme l'ordre d'évaluation des règles ne change pas la sémantique dans ASPERIX, on peut commencer par évaluer `R1` puis `R2`, et le début de notre arbre ASPERIX sera celui de la question 4 (avec le IN de la racine contenant les faits, donc **(condition 1)** aucun atome de prédicat `val`).

Nous voulons maintenant interdire aux évaluations des autres règles de faire apparaître `val(x, true)` ou `val(x, false)`. Ceci peut se faire en imposant également la **(condition 2)** en interdisant dans le programme toute règle ayant un atome de prédicat `val` dans la tête.

Ainsi, aucun atome de prédicat `val` ne pourra apparaître sous les sommets **(A)**, **(B)** et **(C)**, et tout modèle stable en contiendra 1 (pour les branches issues de **(A)** et **(B)**) ou 0 (pour les branches issues de **(C)**).

La contrainte `R3` ne pourra alors jamais être déclenchée, et cette règle est donc inutile si le programme respecte les conditions 1 et 2.

Remarquons qu'interdire, par exemple dans la condition 2, les règles qui ont simplement `val(x, true)` ou `val(x, false))` en tête de règle ne suffit pas, puisque cette interdiction syntaxique pourrait être contournée sémantiquement, par exemple avec:

```
val(x1, VX1):- something1(). # génère une valeur inconnue pour x1
VX1 = true :- val(x1, VX1).  # cette valeur inconnue, c'est true
```

Cette question n'a été traitée que 2 ou 3 fois, mais à peu près correctement.

1.3 Invalidité de la formule propositionnelle

Aucun étudiant n'a attaqué cette partie ...

Nous supposons maintenant que nous avons généré (via le traducteur Π^1) toutes les valuations possibles des variables existentielles $\vec{x}$ de $Q = \exists\vec{x}\forall\vec{y}\,Q'$, et nous souhaitons, dans le sous-arbre ASPERIX correspondant à une de ces valuations σ, générer $\bot$ (absurde) si la formule $\sigma(Q')$ est invalide. Ainsi, il ne restera que les modèles stables correspondant à une valuation σ pour laquelle $\sigma(Q')$ est valide, et nous aurons respecté la propriété demandée à notre transformation.

1.3.1 Analyse d'une conjonction

Pour tester l'invalidité de $\sigma(Q')$ (avec $Q' = c_1 \vee \cdots \vee c_k$), nous allons d'abord générer, pour chaque conjonction c_i, toutes les valuations σ_i qui étendent σ et pour lesquelles $\sigma_i(c_i) = \texttt{false}$. Le calcul (par le traducteur) de ces valuations sera utilisé pour générer la partie $\Pi^2(Q)$ de notre programme.

Question 7 Soit la 3-conjonction $c_1 = x \wedge \neg y \wedge z$ apparaissant dans une 2QBF Q où x est quantifiée existentiellement (et y et z sont quantifiées universellement). Nous souhaitons écrire toutes les règles de la forme:

$$\texttt{fail/2}(c_1, y, V_y, z, V_z) : -val(x, V_x).$$

où les V sont instanciés par des booléens `true` où `false`, et où la règle signifie "si la variable existentielle x est valuée à V_x, alors valuer y par V_y et z par V_z entraîne une évaluation de la conjonction c_1 à `false`. Écrire les 7 règles (positives) générées par le traducteur à partir de Q permettant d'exprimer les cas d'échec de c_1.

Pour répondre à cette question, nous pouvons par exemple construire la table de vérité de la 3-conjonction c_1:

x	y	z	$\mathbf{c_1 = x \wedge \neg y \wedge z}$
true	true	true	false
true	true	false	false
true	false	true	true
true	false	false	false
false	true	true	false
false	true	false	false
false	false	true	false
false	false	false	false

On peut voir sur cette table que:

- si x est évaluée à `true`, il y a 3 valuations de $\{y, z\}$ qui évaluent la conjonction c_1 à `false`. Ceci peut s'exprimer par 3 règles.
- si x est évaluée à `false`, il y a 4 valuations de $\{y, z\}$ qui évaluent la conjonction c_1 à `false`. Ceci peut s'exprimer par 4 règles.

Nous n'avons plus qu'à écrire ceci avec les prédicats utilisés dans l'énoncé.

```
fail/2(c1, y, true, z, true) :- val(x, true).
fail/2(c1, y, true, z, false) :- val(x, true).
fail/2(c1, y, false, z, false) :- val(x, true).

fail/2(c1, y, true, z, true) :- val(x, false).
fail/2(c1, y, true, z, false) :- val(x, false).
fail/2(c1, y, false, z, true) :- val(x, false).
fail/2(c1, y, false, z, false) :- val(x, false).
```

Remarque: Nous avons ici écrit 7 règles car nous voulions des têtes atomiques, afin de pouvoir faire traiter notre programme par clingo (qui considère une disjonction en tête). Puisque les REN autorisent une conjonction en tête, nous n'aurions eu besoin que de 2 règles (une avec 3 atomes en tête et une avec 4 atomes en tête) pour exprimer la même connaissance.

Question 8 Même question avec la 3-conjonction c_1 de la question 7, mais apparaissant dans une 2QBF qui quantifie existentiellement x et y. Puisque nous n'avons besoin que de valuer une seule variable quantifiée universellement, nous aurons besoin d'un prédicat `fail/1`. Écrire les 5 règles (positives) permettant d'exprimer les cas d'échec de c_1.

Nous partons de la même table de vérité que dans la question 7. Nous remarquons que:

- il suffit que x soit évalué à `false` ou que y soit évalué à `true` pour falsifier c_1, et dans ce cas les 2 valuations possibles de z falsifient c_1. Ceci peut s'exprimer par 4 règles.
- si x est évalué à `true` et y à `false`, alors une seule des valuations de z falsifie c_1. Ceci peut s'exprimer par 1 règle.

Ainsi, en utilisant les prédicats de l'énoncé:

```
fail/1(c1, z, true) :- val(x, false).
fail/1(c1, z, false) :- val(x, false).

fail/1(c1, z, true) :- val(y, true).
fail/1(c1, z, false) :- val(y, true).

fail/1(c1, false) :- val(x, true), val(y, false).
```

Question 9 En supposant que toutes les conjonctions contiennent exactement 3 variables distinctes, comptez le nombre de règles nécessaires pour exprimer les règles générant respectivement `fail/0` (toutes les variables sont existentielles) et `fail/3` (toutes les variables sont universelles). Vous justifierez votre réponse en donnant la forme des règles obtenues.

Notre guide doit toujours être la table de vérité de la question 7...

Si toutes les variables $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ sont existentielles: nous considérons alors les cas suivants:

- si la valuation est celle qui satisfie la formule c_1, alors il n'y a pas de cas de "fail", et donc pas de règle à écrire.
- la formule est falsifiée dès qu'il y a une "mauvaise" valuation de x, de y ou de z, ce qui peut s'exprimer par 3 règles:

```
fail/0(c1) :- val(x, false).
fail/0(c1) :- val(y, true).
fail/0(c1) :- val(z, false).
```

Nous avons donc dans ce cas besoin de 3 règles.

Si aucune variable n'est existentielle: dans ce cas, aucune variable n'est fixée et il faut énumérer toutes les possibilités de falsifier c_1. Il y en a 7. Ceci peut s'exprimer par 7 faits atomiques (ou 7 règles avec hypothèse vide et tête atomique, ou 1 règle avec hypothèse vide et une 7-conjonction en tête...). Nous le présentons ci avec des faits:

```
fail/3(c1, x, true, y, true, z, true).
fail/3(c1, x, true, y, true, z, false).
fail/3(c1, x, true, y, false, z, false).
fail/3(c1, x, false, y, true, z, true).
fail/3(c1, x, false, y, true, z, false).
fail/3(c1, x, false, y, false, z, true).
fail/3(c1, x, false, y, false, z, false).
```

1.3.2 C'est la contrainte finale

Pour des raisons de temps, je ne vous ai pas demandé d'écrire le traducteur générant les règles du programme $\Pi^2(Q)$ telles que vous les avez écrites aux questions 7-9.

Question 10 Reprenons l'exemple F de la section 1.1. Écrire le programme $\Pi^1(F) \cup \Pi^2(F)$ associé à cette formule (vous pourrez ne donner que l'échantillon des règles de $\Pi^2(F)$ nécessaire pour répondre à la question 12).

Nous souhaitons étudier la satisfiabilité de $F = \exists x \exists y \forall z\, F'$ avec $F' = (x \wedge \neg y \wedge z) \vee (x \wedge \neg z)$. Nous notons $c_1 = (x \wedge \neg y \wedge z)$ et $c_2 = (x \wedge \neg z)$. Les variables existentielles étant x et y, nous pouvons déjà écrire $\Pi^1(F)$:

```
val(x, true) :- not val(x, false). % R1
val(x, false) :- not val(x, true). % R2
! :- val(x, true), val(x, false).  % R3, optionnel

val(y, true) :- not val(y, false). % R4
val(y, false) :- not val(y, true). % R5
! :- val(y, true), val(y, false).  % R6, optionnel
```

Oui, c'est le même qu'à la question 1, à un renommage des noms de variables près ... Si, de plus, on remarque que les règles suivantes satisfont aux conditions de la section 6, on peut supprimer `R3` et `R6`.

Et on remarque que la 3-conjonction c_1 avec variables existentielles x et y correspond exactement à la question 8. On peut donc recopier les règles telles quelles (ou référencer la réponse à cette question).

Les règles de $\Pi^2(F)$ nécessaires pour répondre à la quetion 12 sont indiquées par ** par la suite (vous n'aviez besoin de ne donner que celles-ci dans votre copie).

La partie du programme $\Pi^2(F)$ généré à partir de c_1 a déjà été donnée à la question 8:

```
fail/1(c1, z, true) :- val(x, false).              # R7
fail/1(c1, z, false) :- val(x, false).             # R8

fail/1(c1, z, true) :- val(y, true).               # R9  **
fail/1(c1, z, false) :- val(y, true).              # R10

fail/1(c1, false) :- val(x, true), val(y, false). #R11
```

Pour la partie de $\Pi^2(F)$ générée à partir de c_2, nous nous faisons la table de vérité de c_2 (ici, elle est plus simple)

x	z	$\mathbf{c_2 = x \wedge \neg z}$
true	true	false
true	false	true
false	true	false
false	false	false

Il n'y a plus qu'à écrire les règles:

La partie du programme $\Pi^2(F)$ généré à partir de c_2 est:

```
fail/1(c2, z, true) :- val(x, true).   # R12   **

fail/1(c2, z, true) :- val(x, false).  # R13
fail/1(c2, z, false) :- val(x, false). # R14
```

Je remarque juste maintenant que les 3 règles précédentes peuvent se factoriser en:

```
fail/1(c2, z, true).

fail/1(c2, z, false) :- val(x, false).
```

Ce qui ouvre des perspectives d'optimisation avec un peu plus de boulot dans l'écriture du générateur ...

Question 11 Quelle contrainte négative (qui serait générée automatiquement par le traducteur $\Pi^3(Q)$) faut-il ajouter pour tester si $\sigma(F')$ est insatisfiable dans la branche correspondant à la valuation σ ?

On remarque que, dans la partie Π^2 du programme, toutes les règles sont positives (il n'y a pas de `not`). Ces règles respectent également les conditions de la question 6 (elles se servent de `val` dans le corps, mais ne le génèrent pas dans la tête.)

On peut donc voir une exécution de $\Pi = \Pi^1 \cup \Pi^2$ en deux phases:

- **Phase 1:** génération des valuations possibles des variables existentielles (chaque feuille contient soit une valuation de toutes les variables existentielles, soit une valuation partielle et dans ce cas ne mènera pas à un modèle stable car le MBT ne pourra pas être satisfait).
- **Phase 2:** génération des prédicats fail/n à partir des feuilles de la phase 1, ce qui ne créera pas de branchement et génère, pour chaque valuation possible des existentiels, toutes les valuations des universels qui falsifient la formule.

On peut donc voir qu'une branche prouve que la valuation des existentiels n'est pas une solution lorqu'il existe une valuation des universels qui falsifie chacune des conjonctions. Nous en faisons donc une contrainte qui est déclenchée dans ce cas.

La branche ne doit pas mener à un modèle stable lorsqu'il existe une valuation de z qui falsifie les 2 conjonctions (cette valuation, que nous notons ici par la variable `VZ`, doit être la même dans les 2 atomes).

```
! :- fail/1(c1, z, VZ), fail1(c2, z, VZ).
```

Question 12 Montrez que la branche de l'arbre ASPERIX correspondant à la valuation $\sigma = \{x : \texttt{true}, y : \texttt{true}\}$ (vous pourrez en faire la racine de l'arbre) ne mène pas à un modèle stable.

Après la phase 1, un des sommets que nous obtenons est:

```
val(x, true), val(y, true). | val(x, false).  val(y, false). | val(x, true), val(y, true).
```

qui correspond à la valuation $\sigma = \{x : \texttt{true}, y : \texttt{true}\}$. Nous construisons maintenant le sous-arbre de ce sommet (réduit à une seule branche, car il ne reste plus que des règles positives).

Après avoir généré un sommet **(D)** contenant `val(x, true)` et `val(y, true)` en utilisant les règles (évaluées positivement) `R1` et `R4`, nous pouvons saturer le IN puisque nous n'avons plus que des règles positives. Nous générons, par exemple:

```
val(c1, z, true). # en appliquant R9
val(c2, z, true). # en appliquant R12
!.                # en appliquant la contrainte finale
```

Le résultat de l'unique branche issue de **(D)** n'est pas un modèle stable car il contient absurde.