CS Games 2015 — Université de Sherbrooke

Compétition d'informatique théorique

L'*informatique théorique* concerne l'étude des aspects fondamentaux de l'informatique à l'aide des mathématiques et de la logique, en particulier l'étude de ceux liés à l'algorithmique et aux langages de programmation. Elle propose des concepts et des théories qui permettent de saisir, en termes descriptif et algorithmique, l'essence même des systèmes informatiques depuis leur spécification jusqu'à leur implémentation prouvée. Plus généralement, elle s'intéresse à des modèles abstraits, à des méthodes formelles et à des techniques de description et d'analyse utiles dans l'étude de questions fondamentales à propos de l'information et du traitement de l'information. Les questions de cette compétition ont été formulées dans cette perspective.

Instructions :

- Vous n'avez besoin que de feuilles de papier et de crayons.
- Les réponses trop longues peuvent être écrites sur des feuilles séparées.
- Les questions ne sont pas posées dans un ordre particulier (sauf la dernière question qui est liée à la question 4).
- Les réponses doivent être claires (compréhensibles par le correcteur), précises (sans erreur), complètes (toutes les étapes de résolution du problème doivent être présentes) et concises (la méthode de résolution est la plus courte possible).
- Vous ne pouvez pas poser de questions pendant cette compétition.
- Vous avez trois heures. Bonne chance !

Nom (1) : _____

Nom (2) : _____

Équipe : _____

Question	Note	Points
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
Total		90

1. (10 points) En théorie tout est connu, mais rien ne fonctionne. En pratique tout fonctionne, mais personne ne sait pourquoi. Quelle est la théorie mathématique à la base de chacun des langages de programmation suivants ?

Lisp (ou votre langage de programmation fonctionnelle favori) _____

Occam (ou votre langage de programmation concurrente favori) _____

Prolog (ou votre langage de programmation logique favori) _____

SQL (un langage d'interrogation de base de données) _____

Yacc (ou votre « langage de programmation » de construction de compilateurs) _____

2. (10 points) La combinatoire comme outil essentiel dans l'analyse d'algorithmes. Une chaîne de Prüfer est une chaîne de longueur $n-2$ sur un alphabet de n symboles. Un résultat intéressant à propos des chaînes de Prüfer est qu'il existe une bijection entre l'ensemble des arbres étiquetés de sommets $\{1, 2, \ldots, n\}$ et l'ensemble des chaînes de longueur $n-2$ sur $\{1, 2, \ldots, n\}$.

   L'algorithme suivant construit un arbre étiqueté de n sommets à partir d'une chaîne de Prüfer $c = c_1 c_2 \cdots c_{n-2}$, où $1 \leq c_i \leq n$. Dans cet algorithme, d_i est initialisé avec le degré de chaque sommet de l'arbre représenté par la chaîne c (étapes 2–7).

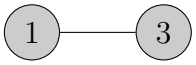
   `Input` une chaîne de Prüfer $c = c_1 c_2 \cdots c_{n-2}$

   `Output` un arbre (V, E) avec $V = \{1, 2, \ldots, n\}$

   0. `let` $E = \emptyset$
   1. `let` $c_{n-1} = n$
   2. `for` $i = 1$ `to` n `do`
   3. `let` $d_i = 1$
   4. `end for`
   5. `for` $i = 1$ `to` $n - 2$ `do`
   6. `let` $d_{c_i} = d_{c_i} + 1$
   7. `end for`
   8. `for` $i = 1$ `to` $n - 1$ `do`
   9. `let` $j = \min(k \mid d_k = 1)$
   10. `let` $E = E \cup \{(j, c_i)\}$
   11. `let` $d_j = 0$
   12. `let` $d_{c_i} = d_{c_i} - 1$
   13. `end for`

   L'exemple suivant illustre le fonctionnement de l'algorithme de décodage de Prüfer pour la chaîne $c = [1, 2, 6, 5, 1, 8]$. Le tableau d est initialisé à $[3, 2, 1, 1, 2, 2, 1, 2]$. Notez que pour une chaîne donnée c, le degré d'un sommet i (le nombre d'arêtes incidentes au sommet i) est un de plus que le nombre de fois où i apparaît dans c.

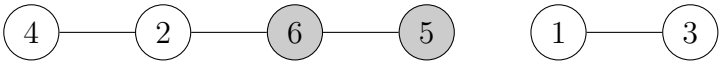
$j = \min(3, 4, 7) = 3$ et $c_1 = 1$, d'où l'ajout de $(3, 1)$.



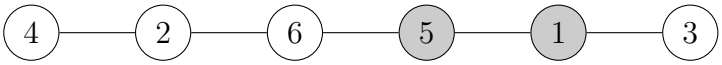
$j = \min(4, 7) = 4$ et $c_2 = 2$, d'où l'ajout de $(4, 2)$.

$j = \min(2, 7) = 2$ et $c_3 = 6$, d'où l'ajout de $(2, 6)$.

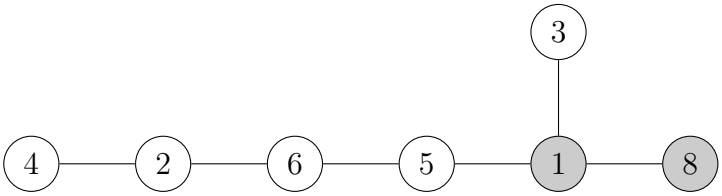
$j = \min(6, 7) = 6$ et $c_4 = 5$, d'où l'ajout de $(6, 5)$.



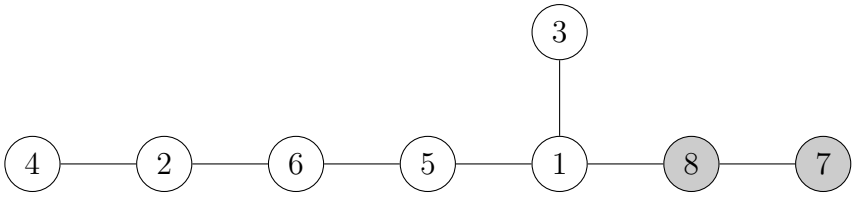
$j = \min(5, 7) = 5$ et $c_5 = 1$, d'où l'ajout de $(5, 1)$.



$j = \min(1, 7) = 1$ et $c_6 = 8$, d'où l'ajout de $(1, 8)$.



$j = \min(7) = 7$ et $c_7 = 8$, d'où l'ajout de $(7, 8)$.



(5 points) Faites une analyse de la complexité de calcul de l'algorithme de décodage précédent et exprimez vos résultats dans les notations asymptotiques usuelles Ω et O (vous pouvez annoter le programme si nécessaire).

(2 points) À quelle classe de complexité appartient le problème de décodage d'une chaîne de Prüfer ?

Les chaînes de Prüfer sont attrayantes parce qu'elles peuvent être facilement manipulées dans un algorithme génétique qui traite des arbres. Par exemple, il existe de nombreuses applications où il est nécessaire de trouver un arbre de recouvrement optimal avec des propriétés structurelles particulières (par exemple, un arbre optimal où le degré de chaque sommet est plus petit qu'un nombre entier donné $k > 0$). Le problème sous-jacent dans ces applications est la génération des chaînes de Prüfer.

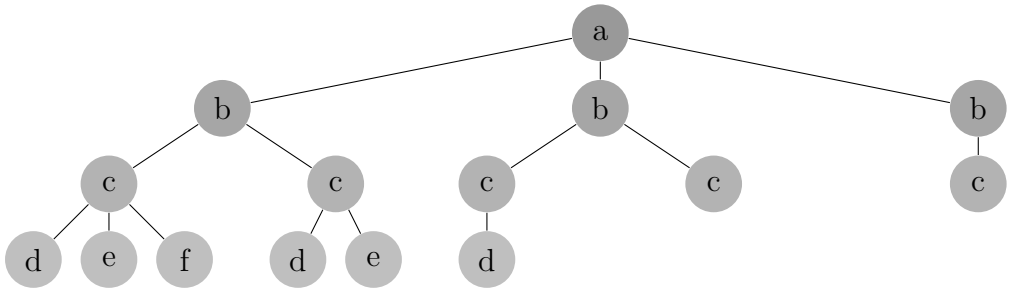
(3 points) Supposons qu'une application requiert de générer toutes les chaînes de Prüfer et, pour chaque chaîne, de trouver l'arbre correspondant afin de déterminer s'il est potentiellement l'optimal par rapport à certains critères. Un tel algorithme est un algorithme par *force brute*. À quelle classe de complexité appartient le problème résolu par cet algorithme ?

3. (10 points) Ha, les possibilités fascinantes des ordinateurs quantiques ! Tout le monde, même votre grand-mère, a entendu dire que les ordinateurs quantiques seront capables de résoudre des problèmes complexes qui sont au-delà de la puissance des meilleurs superordinateurs classiques de demain. En particulier, votre grand-mère veut votre opinion, à titre d'informaticien ou d'ingénieur logiciel, à ce sujet. Heureusement, votre grand-mère est brillante. Elle veut une réponse convaincante à la question notoire « P = NP ou P $\neq$ NP ? ». En d'autres termes, les ordinateurs quantiques peuvent-ils résoudre des problèmes *difficiles* aussi aisément que des problèmes *faciles* ? Transcrivez la réponse que vous souhaiteriez donner à votre grand-mère dans l'espace prévu ci-dessous.

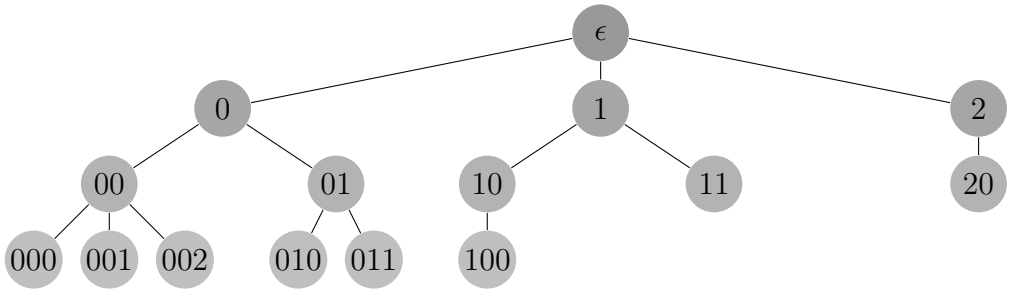
4. (10 points) Des définitions formelles comme source d'une théorie—la théorie des automates d'arbres ! Étant donné un alphabet Σ, un arbre étiqueté dans Σ, noté t, est décrit par son ensemble de sommets (le « domaine » $\mathrm{dom}(t)$) et une fonction d'étiquetage des sommets dans l'alphabet Σ. Le domaine $\mathrm{dom}(t)$ est inclus dans l'ensemble $\{0, 1, \ldots, k-1\}^*$. Formellement, un *arbre d'arité k étiqueté dans Σ* est une application $t : \mathrm{dom}(t) \to \Sigma$, où $\mathrm{dom}(t)$ est un ensemble non vide, fermé sous les préfixes, qui satisfait

$$wj \in \mathrm{dom}(t),\ i < j \Rightarrow wi \in \mathrm{dom}(t).$$

À titre d'exemple, l'arbre suivant est un arbre d'arité 3 étiqueté dans $\{a, b, c, d, e, f\}$. En particulier, $\mathrm{dom}(t) \subseteq \{0, 1, 2\}^*$ (rappel : ϵ désigne le mot vide).



t



$\mathrm{dom}(t)$

($2\frac{1}{2}$ points) Soit t un arbre quelconque d'arité k étiqueté dans Σ. Est-ce que le domaine $\mathrm{dom}(t)$ (l'ensemble des sommets) définit un langage régulier sur $\{0, 1, \ldots, k-1\}$? Justifiez votre réponse.

($2\frac{1}{2}$ points) Soit t un arbre quelconque d'arité k étiqueté dans Σ. La *frontière extérieure* de t contient les éléments $wi \notin \mathrm{dom}(t)$, où $w \in \mathrm{dom}(t)$ et $i < k$. Formellement,

$$\mathrm{fr}^+(t) = \{wi \notin \mathrm{dom}(t) \mid w \in \mathrm{dom}(t) \wedge i < k\}.$$

Quelle est la frontière extérieure de l'arbre de la page précédente (vous pouvez dessiner la frontière extérieure dans l'arbre de la page précédente) ?

($2\frac{1}{2}$ points) Soit t un arbre quelconque d'arité k étiqueté dans Σ. Donnez une définition formelle de la frontière de t, notée $\mathrm{fr}(t)$.

($2\frac{1}{2}$ points) Soit t un arbre quelconque d'arité k étiqueté dans Σ. Donnez une définition d'un chemin de la racine de t à un sommet donné qui appartient à $\mathrm{fr}(t)$.

5. (10 points) Le célèbre *problème de l'arrêt* ! Le problème de l'arrêt consiste, étant donné un programme quelconque, à déterminer s'il termine ou non. Bien sûr, nous pouvons exécuter le programme pendant un certain temps, mais que faire si le programme ne s'est pas arrêté après un milliard d'années. Turing a prouvé qu'il n'existe aucun programme qui résout le problème de l'arrêt.

   (1 point) Qui est Turing ?

   Les principaux arguments de sa démonstration sont les suivants. Supposons qu'un tel programme P existe. Nous pouvons modifier P pour produire un nouveau programme P' qui effectue les opérations suivantes. Étant donné un autre programme Q en entrée, P'

   — s'exécute sans jamais s'arrêter si Q termine avec son propre code en entrée, ou

   — s'arrête si Q s'exécute sans jamais s'arrêter avec son propre code en entrée.

   Donner à P' son propre code en entrée. Selon les deux conditions précédentes, P' s'exécutera sans jamais s'arrêter, s'il s'arrête, ou s'arrêtera s'il s'exécute sans jamais s'arrêter. Par conséquent, P' — et par implication P — ne peuvent pas avoir existé en premier lieu.

   En conclusion le problème de l'arrêt est indécidable.

   (6 points) Maintenant, considérons un programme *autopoïétique*, c'est-à-dire un programme capable de se reproduire. Tout comme pour le problème de l'arrêt, on peut se demander si un tel programme existe. Le cas le plus simple est un programme qui ne fait rien, sauf imprimer lui-même son propre code. Pensez-vous qu'un tel programme

pourrait être écrit dans votre langage de programmation préféré ? Si oui, à quoi pourrait-il ressembler ? Si non, quels sont vos arguments qui prouvent le contraire ?

(3 points) Qu'est-ce que vous auriez besoin de changer à votre réponse précédente si le programme faisait quelque chose d'utile ?

6. (10 points) Le langage XML en tant que langage formel. XML est un langage de balisage qui comporte un ensemble de règles de codage de documents dans un format qui est à la fois lisible par les humains et par les ordinateurs. Voici quelques règles avec de brèves explications tirées de la spécification XML 1.0 (texte anglais) : les règles [16] et [17] qui définissent les *processing instructions* et les règles [18] à [21] qui définissent les *CDATA sections*.

```
Processing Instructions
[16]  PI       ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17]  PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

PIs are not part of the document's character data, but MUST be passed through to the application. The PI begins with a target (PITarget) used to identify the application to which the instruction is directed. The target names "XML", "xml", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets. Parameter entity references MUST NOT be recognized within processing instructions.

```
CDATA Sections
[18]  CDSect  ::= CDStart CData CDEnd
[19]  CDStart ::= '<![CDATA['
[20]  CData   ::= (Char* - (Char* ']]>' Char*))
[21]  CDEnd   ::= ']]>'
```

Within a CDATA section, only the CDEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "<" and "&". CDATA sections cannot nest.

(2 points) Les règles [16] à [21] sont les règles de production d'une grammaire hors contexte écrites dans la notation étendue (les chaînes de caractères entre apostrophes sont des symboles terminaux). Le but de cette question est de définir les *processing instructions* **OU** les *CDATA sections* à l'aide d'une expression régulière, car les opérateurs de concaténation, de fermeture de Kleene ($*$), de disjonction ($|$), de zéro ou une occurrence ($?$) et de différence ($-$) sont également utilisés dans l'écriture des expressions régulières, sauf un ! Lequel ?

(5 ou 8 points) Donnez une expression régulière pour les *processing instructions*, c'est-à-dire pour le langage généré par la règle [16] (5 points) ou, si vous voulez accumuler plus de points, donnez une expression régulière pour les *CDATA sections*, c'est-à-dire pour le langage généré par les règles [18] à [21]. Pour simplifier le problème, ignorez `PITarget`, c'est-à-dire ne considérez que la règle suivante :

```
Processing Instructions
[16']  PI ::= '<?' (S (Char* - (Char* '?>' Char*)))? '?>'
```

Supposez que `Char` représente tout caractère acceptable par XML et que `S` représente un espace.

7. (10 points) DES PILES, DES PILES, DES PILES, . . . ! Un *automate à pile* est un 6-uplet $(Q, \Sigma, \Gamma, \delta, q_0, F)$, où Q est un ensemble fini d'états, Σ un ensemble fini appelé l'alphabet d'entrée, Γ un ensemble fini appelé l'alphabet de sortie, $q_0 \in Q$ l'état initial, $F \subseteq Q$ l'ensemble d'états finaux et δ la fonction de transition de $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ vers des sous-ensembles de $Q \times (\Gamma \cup \{\epsilon\})$. Il y a plusieurs conditions d'acceptation définies sur ces automates.

   (2 points) Considérez le langage $L = \{a^i b^j c^i d^j \mid i, j \geq 0\}$. Montrez en utilisant le lemme de l'étoile (*pumping lemma*) que le langage L n'est pas un langage régulier.

(2 points) En utilisant des arguments intuitifs, expliquez pourquoi ce langage n'est pas un langage hors contexte.

(2 points) Quelles modifications devraient être apportées à un *automate à pile* de sorte que le langage L puisse être accepté par une instance de votre nouveau type de machine ?

(2 points) Donnez la définition formelle complète de l'instance de la machine qui accepte L, c'est-à-dire donnez l'ensemble des états, les alphabets, l'état initial, les états finaux (avec les conditions d'acceptation), la fonction de transition, etc.

(2 points) Est-ce que votre machine a la même puissance de calcul qu'une machine de Turing ?

8. (10 points) Non, une preuve est une preuve. Quel genre de preuve ? C'est une preuve. Une preuve est une preuve, et quand vous avez une bonne preuve, c'est parce que c'est prouvé !$_{\text{ex-premier ministre Jean Chrétien}}$ Zoé a été assassinée, et Ada, Boris, et Charles sont suspects. Ada dit qu'elle ne l'a pas assassinée. Elle dit que Boris était l'ami de la victime, mais que Charles détestait la victime. Boris dit qu'il était hors de la ville le jour de l'assassinat, et d'ailleurs il ne connaissait pas la jeune fille. Charles dit qu'il est innocent et qu'il a vu Ada et Boris avec la victime juste avant l'assassinat. En supposant que tout le monde—sauf peut-être pour le meurtrier—dit la vérité, utilisez le principe de résolution pour résoudre le crime.

   L'idée du principe de résolution est simple. Si nous savons que P est vrai ou que Q est vrai et nous savons aussi que P est faux ou R est vrai, alors il doit être le cas que Q est vrai ou R est vrai. Formellement, étant donné une clause (une clause est un ensemble de littéraux représentant leur disjonction, où un littéral est un énoncé atomique ou la négation d'un énoncé atomique) contenant un littéral ϕ et une autre clause contenant le littéral $\neg\phi$, nous pouvons en déduire la clause constituée de tous les littéraux des deux clauses sans la paire complémentaire.

$$\frac{\begin{array}{ll}\Phi & \text{with } \phi \in \Phi \\ \Psi & \text{with } \neg\phi \in \Psi\end{array}}{(\Phi - \{\phi\}) \cup (\Psi - \{\neg\phi\})}$$

   Par exemple, considérez la déduction suivante. La première prémisse affirme que soit P ou Q est vrai. La deuxième prémisse stipule que soit P est fausse ou R est vrai. De ces prémisses, nous pouvons en déduire par le principe de résolution que Q est vrai ou R est vrai.

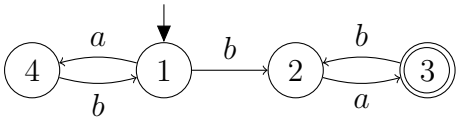
1.	$\{P, Q\}$	une prémisse
2.	$\{\neg P, R\}$	une prémisse
3.	$\{Q, R\}$	de 1,2

9. (10 points) MOTS INFINIS, ARBRES INFINIS ! Les expressions régulières et les grammaires hors contexte définissent des langages de mots finis. Ces derniers peuvent être respectivement acceptés par des automates finis et des automates à pile définis à partir du même alphabet. Cela signifie que pour une expression régulière donnée, il existe un automate fini qui accepte le langage défini par l'expression régulière (l'inverse est également vrai). De même, pour une grammaire hors contexte donnée, il existe un automate à pile qui accepte le langage engendré par la grammaire hors contexte (l'inverse est également vrai).

   Pour un alphabet donné Σ, l'ensemble des mots finis est noté Σ^*. Cette question concerne les automates qui acceptent un sous-ensemble de l'ensemble des mots *infinis* noté Σ^ω. Ne vous affolez pas à ce sujet, en particulier par la présence du symbole ω (oméga). Par exemple, le mot de ababaaa... $= ababa^\omega$ est le mot infini qui commence par $abab$ suivi par une suite infinie de a. Vous pouvez imaginer la différence avec l'ensemble des mots finis $ababa^*$.

   (2 points) Décrivez dans votre langue maternelle le mot $(ab)^*(ba)^\omega$.

   Un ω-automate est un automate qui reconnaît des mots infinis. Plus précisément, un ω-automate est un quintuplet $(Q, \Sigma, \delta, q_0, Acc)$, où Q est un ensemble fini d'états, Σ un ensemble fini appelé l'alphabet, $q_0 \in Q$ l'état initial, Acc la condition d'acceptation et $\delta : Q \times \Sigma \to 2^Q$ la fonction de transition. Il existe plusieurs conditions d'acceptation. La plus simple est appelée la *condition d'acceptation de Büchi* : $Acc = F \subseteq Q$ et un mot $\alpha \in \Sigma^\omega$ est accepté par un tel automate si et seulement si au moins l'un des états de F a été visité *infiniment souvent* lors de la reconnaissance de α. L'automate suivant accepte les mots $(ab)^*(ba)^\omega$ ($q_0 = 1$ and $F = \{3\}$).



(2 points) Construisez un ω-automate avec la *condition d'acceptation de Büchi* qui reconnaît le langage

$$L := \{\alpha \in \{a, b\}^\omega \mid \alpha \text{ termine avec } a^\omega \text{ ou termine avec } (ab)^\omega\}.$$

Par analogie à la conversion des expressions régulières en des automates finis (automates sur mots finis), des formules logiques peuvent être converties en des ω-automates (automates sur mots infinis). Ces formules doivent être, cependant, écrites en *logique temporelle linéaire*, ce qui est utile pour modéliser la dynamique des systèmes (ou « logiciels ») qui ne terminent jamais.

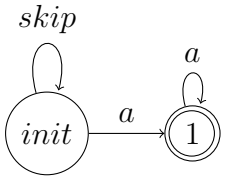
Cette logique est construite à partir de propositions atomiques $p_1, p_2, \ldots, p_n$ de connecteurs booléens (et, ou, négation), d'opérateurs unaires temporels X (« next »), F (« eventually »), G (« always ») et de l'opérateur binaire U (« until ») qui n'est pas considéré par la suite.

Une formule φ dans cette logique est interprétée sur des ω-sequences $\alpha \in (\{0, 1\}^n)^\omega$. Dans un mot de longueur n formé de 0 et 1, 0 dans la position i signifie que p_i n'est pas satisfaite et 1 dans la position i signifie que p_i est satisfaite. En outre, dans une

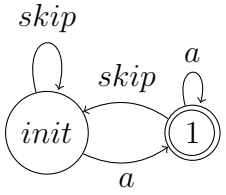
séquence infinie formée de mots de longueur n, $\alpha[i]$ réfère au i^{e} mot et $\alpha[i, \omega]$ réfère à la ω-séquence à partir du i^{e} mot. La relation $\alpha \models \varphi$ est définie par induction par les clauses suivantes (nous fournissons uniquement un sous-ensemble de clauses) :

$\alpha \models p_i$	ssi $\alpha[0]$ a un 1 dans sa i^{e} composante ;
$\alpha \models \varphi_1 \wedge \varphi_2$	ssi $\alpha \models \varphi_1$ et $\alpha \models \varphi_2$;
$\alpha \models \mathrm{X}\varphi$	ssi $\alpha[1, \omega] \models \varphi$ (φ est prochainement satisfaite) ;
$\alpha \models \mathrm{F}\varphi$	ssi $\exists i \geq 0$ tel que $\alpha[i, \omega] \models \varphi$ (φ est éventuellement satisfaite) ;
$\alpha \models \mathrm{G}\varphi$	ssi $\forall i \geq 0$, $\alpha[i, \omega] \models \varphi$ (φ est toujours satisfaite).

Il existe un outil extraordinaire sur le Web qui traduit toute formule en logique temporelle linéaire en un ω-automate. Il est appelé LTL2BA. Par exemple, la formule « F (G a) », qui est une formule canonique pour une propriété de *persistance*, est traduite en l'automate suivant :



La formule « G (F a) », qui est une formule canonique pour une propriété de *récurrence*, est traduite en l'automate suivant :



(2 points) Dessinez un ω-automate qui correspond à la formule suivante :

$$(\mathrm{X}\ a) \wedge (\mathrm{X}\ \mathrm{X}\ b) \wedge (\mathrm{X}\ \mathrm{X}\ \mathrm{X}\ \mathrm{G}\ c)$$

(2 points) Un automate d'arbres est un automate qui reconnaît des ensembles d'arbres infinis. Dans tout arbre infini d'arité k étiqueté dans Σ, chaque sommet possède k successeurs directs et les branches ne terminent jamais. Quel est alors $\mathrm{dom}(t)$ (voir la question #4) ?

(2 points) Qu'est-ce qu'un chemin dans un arbre infini d'arité k étiqueté dans Σ ?

(Fin du questionnaire)