Lambda-calcul et programmation fonctionnelle : Projet — Épisode 1

Due : le dimanche 5 mars 2023 23:59

1 L'histoire

Peut-être connaissez-vous déjà l'histoire de la princesse ou du tigre ? Un prisonnier doit choisir entre deux cellules dont l'une cache une princesse et l'autre un tigre. S'il choisit la princesse, il doit l'épouser, mais s'il tombe sur le tigre, il est dévoré.

En lisant cette histoire le roi de votre contée eu une idée. « C'est exactement ce qu'il me faut pour en finir avec les prisonniers ! Mais je ne veux pas que leur choix soit uniquement dû au hasard, car ça ne serait pas drôle ; c'est pourquoi je vais afficher des inscriptions sur les portes des cellules. Ceux qui se montreront astucieux et qui auront l'esprit assez logique pour en tirer partie seront graciés. »

Le roi expliqua aux prisonniers que dans les cinq premiers épreuves, chacune de cellules contenait un vrai tigre ou une peluche, et toutes les combinaisons était possibles ; il pouvait y avoir deux tigres, deux peluches, ou un tigre et une peluche.

Épreuves 1 : Pour la première épreuve, le roi a collé un affiche sur la porte de chaque cellule : « Une des affiches dit la vérité », promit le roi, « et l'autre ment. »

–1–	–2–
Il y a une peluche dans cette cellule et un tigre dans l'autre.	Il y a une peluche dans une cellule et il y a un tigre dans une cellule.

En voyant cela, l'un des prisonniers a eu l'idée d'écrire un programme en Haskell pour lui aider, ainsi que ses collègues, à éviter de se faire dévorer par un tigre. Le problème est qu'il était tellement stressé par l'événement, qu'il n'arrive pas a se concentrer pour finir son programme. Donc, il a demandé votre aide pour sauver leur vies. Comme vous trouvez que cette manière de traiter les prisonniers n'est pas du tout digne (cela ne respecte pas les droits humains), vous avez accepté la tâche d'aider les prisonniers.

Dans cette première étape du projet, vous devez écrire un programme en Haskell capable d'aider les prisonniers dans leurs épreuves. Comme il n'est pas possible de connaître en avance le contenu des épreuves qui seront imposées par le roi, vous devez écrire un programme générique, capable de réaliser du raisonnement logique, car le roi a dit « les esprits logiques en tireront partie ». Le programme doit être capable de raisonner sur des formules qui expriment des phrases comme celles de la première épreuve.

Vous devez indiquer le type de chaque fonction et expression (fonction sans argument) de votre projet. Comme pour les TP, vous devez implémenter un test pour chaque fonction spécifiée dans le sujet. Vous devez également suivre le Guide de style de Haskell, disponible sur le site de la discipline.

Question 1. Après vous remercier grandement, le prisonnier programmeur vous montre quelques lignes de code qu'il a pu écrire :

```
door1 :: Formula
door1 = And (Var "p1") (Var "t2")
```

Le prisonnier est super stressé, mais il est tout de même capable de vous expliquer le début de son idée : « Je veut créer un type `Formula` pour pouvoir exprimer les épreuves sous la forme d'une formule logique. Ici, j'exprime le fait qu'il y a une peluche dans la cellule 1 (`Var "p1"`) et qu'il y a un tigre dans la cellule 2 (`Var "t2"`). Pour joindre les deux dans une seule phrase, j'utilise une conjonction : `And`. »

Écrivez le type structuré `Formula` pour représenter des formules booléennes (c.-à-d., des formules logiques). Une formule logique est définie récursivement, comme suit :

- `T` est une formule logique (représentant le vrai).
- `F` est une formule logique (représentant le faux).
- Si `s` est une chaîne de caractères, alors `Var s` est une formule logique (variables propositionnelles).

— Si `phi` est une formule logique, alors `Not phi` est une formule logique. (négation)
— Si `phi` et `psi` sont des formules logiques, alors `And phi psi` est une formule logique (conjonction).
— Si `phi` et `psi` sont des formules logiques, alors `Or phi psi` est une formule logique (disjonction).
— Si `phi` et `psi` sont des formules logiques, alors `Imp phi psi` est une formule logique (implication).
— Si `phi` et `psi` sont des formules logiques, alors `Eqv phi psi` est une formule logique (équivalence).

Par exemple, ceci est une formule logique (je vous laisse deviner sa signification pour l'histoire des prisonniers) :

```
And (Or (Var "p1") (Var "p2")) (Or (Var "t1") (Var "t2"))
```

Obs. : Le type `Formula` doit dériver de la classe `Show`, pour pouvoir s'afficher.

Question 2. « Génial ! » se réjouit le prisonnier. « Maintenant, nous avons un langage pour exprimer des formules logiques. Pourtant, les deux formules que nous avons ne suffiront pas pour surmonter la première épreuve. », a dit le prisonnier programmeur. « Nous avons aussi besoin d'une formule `constraint` qui décrit le fait qu'il ne peut pas y avoir un tigre et une peluche (en même temps) dans chaque cellule, ainsi que d'une formule `reglement` pour exprimer le fait que, dans la première épreuve au moins, l'une des portes dit la vérité et l'autre ment. »

Écrivez les deux formules ainsi qu'une troisième formule `challenge1` qui fait la conjonction de toutes les formules de la première épreuve.

Question 3. Après avoir vu votre travail, le prisonnier est plus confient dans son avenir. Il est même capable de vous expliquer d'avantage son idée (qui d'ailleurs fait penser aux films de science fiction) : « Maintenant, il faut décider d'une manière de représenter tous les *mondes possibles*. Vous savez, un monde possible est la représentation d'une vérité concevable. Par exemple, le roi a dit qu'il peut y avoir deux peluches. J'exprime ce monde possible comme suit :

```
w0 :: World
w0 = ["p1", "p2"]
```

Il a dit aussi qu'il peut y avoir deux tigres :

```
w1 :: World
w1 = ["t1", "t2"]
```

Mais pour que le raisonneur logique soit complet, mêmes des mondes qui ne semblent pas concevables doivent être représentés. Par exemple, deux peluches et deux tigres :

```
w2 :: World
w2 = ["p1", "p2", "t1", "t2"]
```

En effet, l'ensemble de tous les mondes possibles contient toutes les combinaisons de toutes les variables propositionnelles du problème, y compris la combinaison vide. »

Définissez le type `World`, pour représenter le type des mondes possibles. Ensuite, écrivez la fonction `genAllWorlds` qui, pour une liste de noms de variables propositionnels (tel que `["p1", "p2", "t1", "t2"]`) génère la liste de tous les mondes possibles pour ces variables.

Question 4. « C'est bien ! Mais c'est maintenant que ça se corse. », a dit le prisonnier un peu préoccupé. « Car la partie la plus difficile du programme reste à faire. » Après une pause, il continue : « Nous devons pouvoir vérifier si un monde possible *satisfait* une formule. Autrement dit, nous devons pouvoir vérifier si une formule est vrai dans un monde possible. Les deux premiers cas sont assez simples : la formule `T` est vraie dans tous les mondes possibles, alors que la formule `F` est toujours fausse. Pour les variables propositionnelles, cela dépend. Par exemple, la formule `Var "p1"` est vraie dans un mode possible `w` si et seulement si `w` contient `"p1"`. Et pour les autres, » a continué le prisonnier « c'est un peu différent. Par exemple : la formule `And phi psi` est vraie dans `w` si et seulement si `phi` est vraie dans `w` et `psi` aussi. Et ainsi de suite ... »

Écrivez la fonction `sat :: World -> Formula -> Bool` qui, pour un monde possible `w` et une formule `phi` passés en arguments, vérifie si `w` satisfait `phi`.

Question 5. « Nous y sommes presque maintenant. » Le prisonnier programmeur commence vraiment à avoir l'espoir. « Pour finir notre programme, il faut juste pouvoir trouver dans quelle monde possible les formules qui représentent l'épreuve sont vraies. Il suffit de générer tous les mondes possibles pour une formule et vérifier un par un. »

Écrivez la fonction `findWorlds :: Formula -> [World]` qui, pour une formule `phi` renvoie la liste de tous mondes possibles qui satisfont `phi`.

Astuce : Il faut connaître les variables propositionnelles du problème pour pouvoir générer tous les mondes possibles. Ceci peut être fait avec une fonction auxiliaire qui "extrait" les noms de variables de la formule.

Question 6. Le moment de vérité est venu. Vous allez maintenant résoudre une par une les épreuves crées par le roi. C'est-à-dire, pour chaque épreuve `N` ci-dessous, écrivez une formule `challengeN :: Formula` pour l'exprimer. Je vous souhaite bonne chance ! Si vous arrivez à sauver la peau des prisonniers, l'histoire continue dans l'épisode 2 du projet.

Les épreuves

Le roi a expliqué aux prisonniers que dans les trois premiers épreuves, chacune de cellules contient un tigre ou une peluche, et toutes les combinaisons sont possibles ; il peut y avoir deux tigres, deux peluches, ou un tigre et une peluche.

Épreuve 1 : Pour la première épreuve, le roi a collé un affiche sur la porte de chaque cellule : « Une des affiches dit la vérité », promit le roi, « et l'autre ment. »

–1–	–2–
Il y a une peluche dans cette cellule et un tigre dans l'autre.	Il y a une peluche dans une cellule et il y a un tigre dans une cellule.

Épreuve 2 : « Dois-je croire ces affiches ? » murmura le prisonnier en tremblant. « Elles sont sincères toutes les deux, ou bien elles sont fausses toutes les deux. », affirma le roi.

–1–	–2–
Une au moins de deux cellules contient une peluche.	Il y a un tigre dans l'autre cellule.

Épreuve 3 : Les affiches disent toutes les deux la vérité ou bien mentent toutes les deux.

–1–	–2–
Il y a un tigre dans cette cellule ou il y a une peluche dans l'autre.	Il y a une peluche dans l'autre cellule.

Épreuve 4 : Pour cette épreuve, l'affiche collé sur la cellule 1 dit la vérité quand il y a une peluche dans cette cellule et ment quand c'est un tigre. Pour la cellule 2 c'est exactement le contraire ; quand il y a une peluche l'affiche ment et quand c'est un tigre l'affiche dit la vérité.

–1– Choisis n'importe quelle cellule, ça n'a pas d'importance !	–2– Il y a une peluche dans l'autre cellule.

Épreuve 5 : Encore une fois, l'affiche collé sur la cellule 1 dit la vérité quand il y a une peluche dans cette cellule et ment quand c'est un tigre. Pour la cellule 2 c'est exactement le contraire ; quand il y a une peluche l'affiche ment et quand c'est un tigre l'affiche dit la vérité.

–1– Choisis bien ta cellule ça'a a de l'importance !	–2– Tu ferais mieux de choisir l'autre cellule !

Épreuve 6 : Pour cette épreuve, le roi expliqua au prisonnier qu'une seule cellule renfermait une peluche et qu'il avait fait mettre un tigre dans chacune des deux autres. Le roi ajouta qu'une seule de trois affiches était sincère.

–1– Il y a un tigre ici.	–2– Cette cellule contient une peluche.	–3– Il y a un tigre dans la cellule 2.

2 Ce que vous devez rendre

Vous devez rendre 8 fichiers :

`Readme:` Ce fichier contiendra (au moins) les noms et prénoms des intégrants de l'équipe du projet.

`CPL.hs:` Ce fichier contiendra le code source du module de même nom. Il s'agit du programme capable de raisonner sur des formules logiques. Ce module doit exporter les objets listés ci-dessous. Leurs spécifications sont données dans les questions de la section 1.

- `Formula (..)`
- `World`
- `genllWorlds` et `testGenAllWorlds`
- `sat` et `testSat`
- `findWorlds` et `testFindWorlds`
- `testAll`

`Cha1.hs ...Cha6.hs:` Un fichier pour chaque épreuve (challenge). Chaque fichier contiendra un module de même nom contenant la modélisation de la n-ième épreuve. Chaque fichier doit exporter l'objet `challengeN` qui correspond à la formule logique de l'épreuve (challenge) en question. Chacun de ces modules doivent importer le module `CPL` pour pouvoir compiler.

3 Correction

La correction du projet sera faite automatiquement, de la manière suivante :

1. Les fichiers rendus seront téléchargés dans un répertoire vide.
2. Ensuite, les fichiers seront compilés.
3. Si les modules ne compilent pas, alors votre note sera **zéro**.

4. Si les modules compilent, ils seront testés avec les cas de test que vous avez proposé.
5. Ensuite, vos modules seront testés avec des cas de test crées par votre enseignant. Ce derniers contiendront la modélisation des épreuves faite par l'enseignant lui même, ainsi que d'autres cas de test qui ne seront pas fournis. Il y aura des cas de test pour toutes les fonctions demandées dans le sujet.
6. Finalement, votre code sera inspecté manuellement, notamment pour la notation du style, la documentation et le plagia.

Donc, cela veut dire que vous devez suivre scrupuleusement la spécification du projet. Les noms des modules, fonctions et expressions, ainsi que leur types sont très importants. Si cela n'est pas fait correctement, les tests proposés par l'enseignant échoueront et vous perdrez des points.

Notez aussi que l'absence d'une fonction dans un module n'empêche pas sa compilation. Mais une erreur dans une fonction, si. Si vous rendez un projet qui compile mais sans quelques fonctions, les tests seront adaptés et le projet sera corrigé tout de même.

La note du projet sera calculée selon le barème suivant :

— (25%) pour le style. Le projet doit être réalisé en respectant le Guide de style de Haskell et doit être bien documenté. L'utilisation du style Haddock des commentaires n'est pas obligatoire mais comptera en votre faveur.

— (75%) pour la correction. Votre projet doit compiler sans erreurs et sans *warnings* et exécuter les tâches demandés dans le sujet. Cela veut dire que, si la date limite est proche, vous devez utiliser votre temps pour corriger les bugs dans les fonctionnalités que vous avez déjà écrit, au lieu d'essayer d'en programmer une nouvelle.

Observation : L'option GHC `-w`, ainsi que les options `-fno-warn-...` (qui servent à éviter l'affichage des *warnings*) sont strictement interdites. Cela vous fera un malus dans la note du projet. En plus, l'enseignant effacera ceci du code source, les *warnings* seront affichées de toutes manières et vous aurez le malus pour cela encore.