TP 2
BASE DE CONNAISSANCES

Introduction

Pour le deuxième TP vous allez devoir construire une base de connaissances. Une base de connaissances sera un ensemble de **type** et de **foncteur** qui pourra être consulté. Votre logiciel va prendre le nom d'un fichier d'entrées sur la ligne de commande. Ce fichier va contenir la base de connaissances. Ensuite, il va construire la base de connaissance. Finalement, l'utilisateur pourra lancer des requêtes au clavier.

Description des entrées

Lors du lancement du logiciel, l'utilisateur doit placer un nom de fichier comme argument sur la ligne de commande. Votre logiciel va lire l'information dans le fichier. Cette information sera une séquence de **type** et de **foncteur**. Un identificateur est une suite de caractères minuscules et majuscules qui n'est pas un mot réservé (`type`, `foncteur`). Les identificateurs sont utilisés pour les noms de **type**, les noms de **foncteur** et pour les éléments d'un type.

Un type sera représenté de la façon suivante :

`type` s = {$e_1, \ldots, e_k$}

Où

- `type` : est un mot réservé indiquant le début d'une déclaration de **type**.
- s : est un identificateur représentant le nom du type. Vous devez vérifier qu'il n'y a pas d'autre **type** ou **foncteur** qui ait le même nom.
- $e_1, \ldots, e_k$: est une séquence d'identificateurs représentants les éléments possibles pour ce **type**. Il ne doit pas y avoir de doublon dans cette séquence.

Un foncteur est représenté de la façon suivante :

`foncteur f ::` $s_1, \ldots, s_n$
$(e_1^1, \ldots, e_n^1)$
.
.
.
$(e_1^m, \ldots, e_n^m)$

Ce foncteur à une arité (nombre d'arguments) de 'n' et contient 'm' clauses.

- `foncteur` : est un mot réservé indiquant le début d'une déclaration de **foncteur**.
- f : est un identificateur représentant le nom du **foncteur**. Vous devez vérifier qu'il n'y a pas d'autre **type** ou **foncteur** qui ait le même nom.
- $s_1, \ldots, s_n$: est une séquence de nom de **type**. Cette séquence représente le type de chacun des arguments du **foncteur**. Ces types doivent avoir été déclarés dans la base de connaissance.
- $(e_1^1, \ldots, e_n^1) \ldots (e_1^m, \ldots, e_n^m)$: une séquence de m clauses contenant n identificateurs chacun. Vous devez vérifier que : $\forall i \in [1..n], \forall j \in [1..m], e_i^j \in s_i$.

Voici un exemple de fichier d'entrées :

```
type booleen = {vrai, faux}
type nombre = {un, deux, trois}
foncteur et :: booleen, booleen, booleen
(vrai, vrai, vrai)
(vrai, faux, faux)
(faux, vrai, faux)
(faux, faux, faux)
foncteur eq :: nombre, nombre, booleen
(un, un, vrai)
(un, deux, faux)
(un, trois, faux)
(deux, un, faux)
(deux, deux, vrai)
(deux, trois, faux)
(trois, un, faux)
(trois, deux, faux)
(trois, trois, vrai)
```

Traitement

Vous devez conserver l'information entrée en mémoire. Pour cela vous allez construire deux instances de la même structure : un arbre AVL. Vous devez construire un arbre AVL qui permettra d'associer une valeur à une clef. Une première instance de cet arbre utilisera les noms de type comme clef et les séquences d'identificateurs comme valeurs. La deuxième instance de cet arbre utilisera les noms de foncteur comme clef et les séquences de clauses comme valeurs.

Vous ne pouvez pas utiliser la classe 'map' de la 'stl'. Vous devez utiliser votre structure d'arbre AVL. Pour tout le reste, vous pouvez utiliser les structures de la 'stl' (sauf 'map').

Commandes de requête

Lorsque toute l'information a été lue, votre logiciel doit interagir avec l'utilisateur en lui demandant une requête. Voici les requêtes possibles :

- $f(e_1^j, \ldots, ?, \ldots, e_n^j)$: Ici, vous devez afficher toutes les valeurs possibles qui peuvent remplacer le '?' lorsque la clause existe dans la base de connaissance. Par exemple, `et(faux, ?, faux)` affiche `{vrai, faux}`. Remarque : les valeurs dans l'ensemble sont dans le même ordre qu'elles ont été rencontrées dans les clauses. Aussi, il n'y a pas de doublon.
- *f*? : affiche toutes les clauses du **foncteur** *f*. Par exemple, `et?` affiche :

```
(vrai, vrai, vrai)
(vrai, faux, faux)
(faux, vrai, faux)
(faux, faux, faux)
```

  Remarque : les clauses sont dans le même ordre qu'elles ont été rencontrées.
- *s*? : affiche tous les éléments du **type** *s*. Par exemple, `booleen?` affiche `{vrai, faux}`. Remarque : les éléments sont dans le même ordre qu'ils ont été rencontrés.

Directives

1. Le TP est à faire seul ou en équipe de deux.
2. Commentaire :
   a. Commentez l'entête de chaque fonction. Ces commentaires doivent contenir la description de la fonction et le rôle de ces paramètres.
   b. Une ligne contient soit un commentaire, soit du code, pas les deux.
   c. Utilisez des noms d'identificateur significatifs.
3. Code :
   a. Pas de `goto`.
   b. Un seul `return` par méthode.
4. Indentez votre code. Assurez-vous que l'indentation est faite avec des espaces.

Remise

Remettre le TP par l'entremise de Moodle. Remettez vos fichiers `'nomEtudiantTP2.tar.gz'`, nous utiliserons un makefile pour le deuxième TP. Le TP est à remettre avant le 31 mars 23:55.

Évaluation

- Fonctionnalité (15 pts) : Votre TP doit compiler sans erreur (il peut y avoir des warnings). J'utilise la ligne de compilation suivante : `make` sur les serveurs Linux de l'université. Le makefile doit construire un exécutable du nom de '`tp2`'.
- Structure (2 pt) : Il faut avoir **plusieurs** fonctions. Construisez un code bien structuré.
- Lisibilité (3 pts) : commentaire, indentation et noms d'identificateur significatif.