TP ISS - Web sémantique

Maxime PAGES, Emilie ESTIVAL

December 7, 2020

1 Introduction

1.1 Contexte

L'objet de ces TP est de manipuler concrètement la notion d'ontologie, d'en voir les aspects principaux, et de d'appréhender le genre de déductions que peut faire un raisonneur à différentes étapes du développement de l'ontologie. Le deuxième TP permet la manipulation d'une ontologie dans notre code source, pour construire une application consciente de la sémantique à partir d'un exemple concret.

1.2 Ontologie

La définition formelle d'une ontologie a été formulée comme suit : *"Une ontologie donne une spécification formelle et explicite une configuration partagée".* Une ontologie contient :

- un ensemble de concepts issus du domaine qu'elle décrit
- des relations entre ces concepts
- des axiomes logiques pour décrire les connaissances :

On dispose ici d'un ensemble de données à décrire. Ces données sont dites brutes : elles ne sont associées à aucune métadonnée. Ces données sont exprimées dans un format spécifique, et ne sont pas interopérables. À l'inverse, on dispose d'une ontologie qui définit les termes d'un vocabulaire interopérable. On va donc annoter les données à l'aide du vocabulaire fourni par l'ontologie.

Le raisonnement est effectué par un logiciel appelé raisonneur, dans notre cas Hermit. Celui-ci s'appuie sur une base de connaissances, ainsi que sur un ensemble de règles qui lui permettent de conclure 'logiquement' de nouvelles connaissances qui viennent s'ajouter au contenu de la base de connaissances.

2 Création de l'ontologie

2.1 L'ontologie légère

2.1.1 Conception

Le contexte est le suivant : on veut développer une application de météorologie intelligente. Pour ce faire, on va décrire des stations météo pour rendre les données qui en sont issues le plus riches possibles.

Tout d'abord, afin de représenter au mieux les phénomènes météorologiques, nous devons définir les Classes, les Object Properties et les Data Properties. Pour cela, nous définissons nos différentes entités directement sur Protégé en suivant les instructions dictées dans les consignes de TP.

Nous commençons par définir les classes telles que présentées dans la **Figure 1**. Par exemple, nous avons représenté la connaissance *Le beau temps et le mauvais temps sont deux types de phénomènes* par une classe 'Phénomène' avec deux sous-classes 'Beau Temps' et 'Mauvais Temps'.

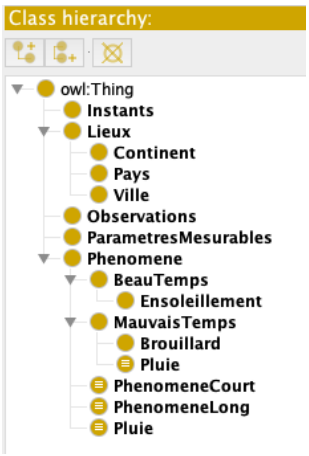

Figure 1: Définition des classes pour exprimer les connaissances énoncées

Ensuite, nous définissons les Object Properties, qui permettent de relier certaines classes à d'autres classes. Par exemple, la propriété "APourDate" permet d'indiquer qu'un objet de type **Observations** a pour date un objet de type **Instants**. La hiérarchie des Object Properties est représentée dans la **Figure 2**.

Enfin, nous définissons les Data Properties. Ces propriétés permettent de relier certaines classes à une valeur donnée. Par exemple, la propriété "AUneDuree" indique qu'un objet de type **Phenomene** a une durée exprimée en minute, de type **float**. La hiérarchie des Data Properties est représentée dans la **Figure 3**.

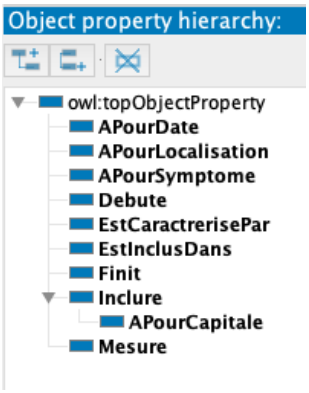

Figure 2: Définition des propriétés et concepts pour énoncer des caractéristiques avec les Object Properties

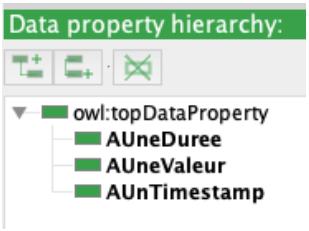

Figure 3: Définition des propriétés et concepts pour énoncer des caractéristiques avec les Data Properties

2.1.2 Peuplement

L'idée est ici de représenter des faits simples à partir de l'ébauche d'ontologie construite précédemment. Par exemple, pour décrire que *La force du vent est similaire à la vitesse du vent*, on ajoute à l'instance de paramètre 'Force du vent' une autre instance 'VitesseDuVent' via 'Same Individual As' comme sur la **Figure 4**.

Après avoir représenté la connaissance 'Toulouse est située en France' en créant Toulouse et France non pas comme une ville et un pays, mais comme des individus sans classe, on remarque que le raisonneur Hermit a déduit que Toulouse et France sont des lieux. Il s'appuie sur la propriété 'est inclus dans' qui relie forcément une instance de un lieu à une autre instance de lieu.

Un autre exemple, lorsqu'on indique que la France a pour capitale Paris, à l'aide de l'Object Property "APourCapitale", alors le raisonneur en déduit automatiquement que Paris est une ville. En effet, la propriété "APourCapitale" associe toujours une ville à un pays.

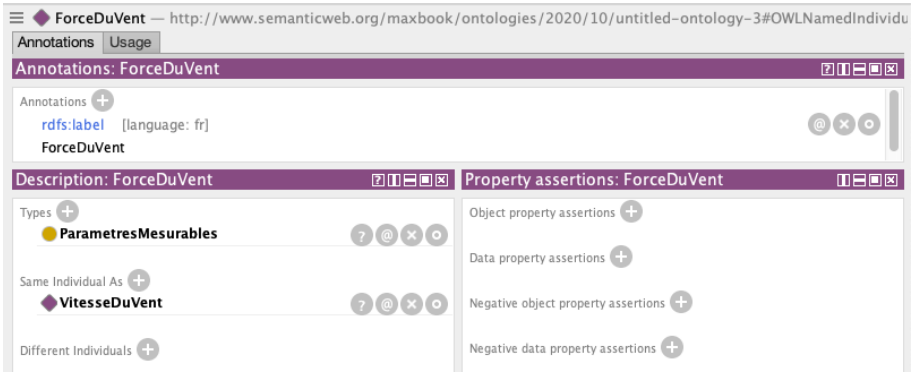

Figure 4: Définition de l'instance de paramètre 'ForceDuVent'

Enfin, si nous indiquons que P1 est une **Observation** qui a une mesuré la valeur 3 mm de pluviométrie à Toulouse à l'instant I1 et que le fait A1 a pour symptôme P1, alors le raisonneur en déduit que A1 est de type **Phénomène**. En effet, comme P1 est une observation et que l'Object Property 'APourSymptome' associe un phénomène à une observation, alors le raisonneur en déduit que A1 est de type Phénomène.

2.2 L'ontologie lourde

2.2.1 Conception

Désormais, nous allons concevoir une ontologie dite "lourde", c'est à dire une ontologie utilisant des restrictions, des inférences ou des constructions de classes par exemple. **Les ontologies lourdes, par rapport aux ontologies légères, incluent des axiomes logiques dans la définition des classes: si un individu remplit toutes les propriétés de la classe alors il appartient forcément à cette classe.**

La question 3.1 permet de comprendre l'utilité d'assigner des restrictions à des classes : dans notre cas, une ville ne peut pas être un pays et cette restriction est renseignée comme telle dans la **Figure 5**.

Figure 5: Restriction "Disjoint with"

Lorsque cette restriction est indiquée, le raisonneur va directement interpréter, qu'à fortiori, un pays ne peut pas être une ville.

De plus, afin de caractériser une classe définie dans Protégé, il est possible d'utiliser la syntaxe de Manchester. Par exemple, dans la question 3.3, la connaissance "un phénomène long est un phénomène dont la durée est au moins de 15 minutes" se traduit en Manchester comme sur la **Figure 6**.

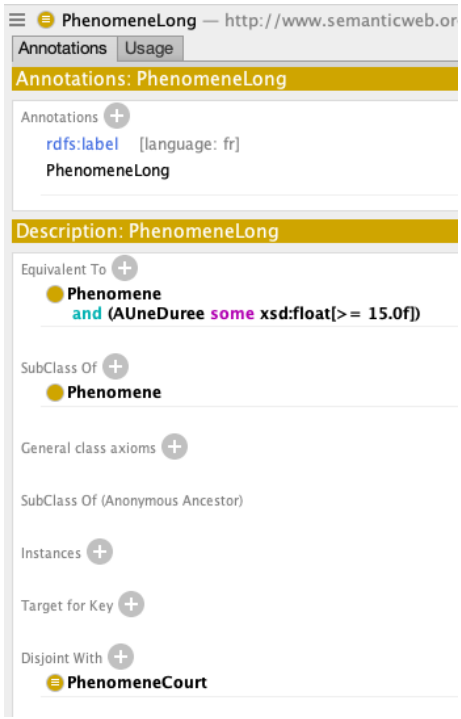

Figure 6: Caractéristiques et syntaxe de Manchester

Dans l'exemple de la **Figure 6**, la classe PhenomemeLong est donc une sous-classe de la classe Phénomène et qui est disjointe avec la classe PhenomeneCourt.

Par ailleurs, certaines caractéristiques sont directement renseignées dans les paramètres des Object Properties, comme le montre la **Figure 7**.

Par exemple, la caractéristique "Transitive" permet de renseigner, au sein de notre ontologie, la relation présentée dans la question 3.6 qui "si un lieu A est situé dans un lieu B, et que ce lieu B est situé dans un lieu C, alors A est situé dans C". De même, la caractéristique "Functional" est utile pour répondre à la question 3.7 qui dit qu'à tout pays correspond une et une seule capitale.

Pour la question 3.8, on utilise la notion de sous-propriété : si un pays a pour capitale une ville, alors ce pays contient cette ville. Pour cela, on indique que la propriété "APourCapitale"

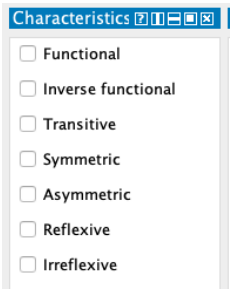

Figure 7: Caractéristiques au sein des Object Properties

est une sous-propriété ("SubProperty of") de la propriété "Inclure", tel que présenté dans la **Figure 2** illustrant la hiérarchie des Object Properties.

Enfin, il est possible de modéliser une classe de manière plus précise, toujours à l'aide de la syntaxe Manchester. Par exemple, la connaissance suivante " la Pluie est un Phénomène ayant pour symptôme une Observation de Pluviométrie dont la valeur est supérieure à 0" se traduit en Manchester par "Phénomène that 'a pour symptôme' some (Observation that *('mesure'value Pluviométrie)* and $('apourvaleur' somexsd : float[> 0]))$". L'utilité de cette caractéristique sera vue lors du peuplement de notre ontologie lourde.

2.2.2 Peuplement

L'intérêt de cette partie est de comprendre l'interprétation du raisonneur à la suite de la définition de notre ontologie lourde.

Par exemple, le raisonneur interprète désormais A1 différemment : pour rappel, il interprétait l'élément A1 comme un phénomène durant notre ontologie légère, alors qu'il interprète maintenant l'élément A1 comme du type **Pluie**. C'est la relation indiquée en syntaxe Manchester durant la conception de notre ontologie lourde qui permet au raisonneur d'interpréter A1 de manière plus précise : A1 a pour symptôme l'observation P1 qui a une valeur de pluviométrie égale à 3. Or, si un phénomène a pour symptôme une observation avec une valeur de pluviométrie supérieur à 0, alors ce phénomène est un phénomène Pluie. Donc A1 est de type Pluie comme l'indique la **Figure 8**.

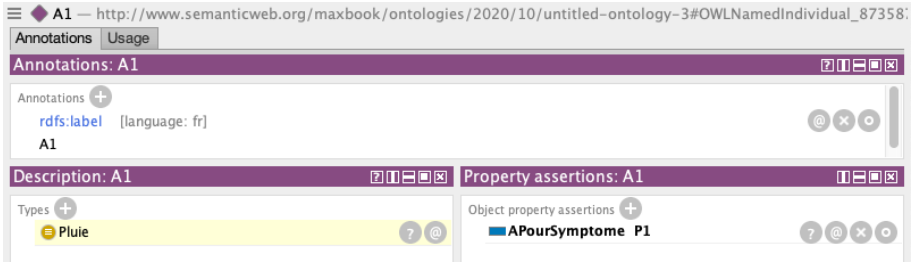

Figure 8: A1 de type Pluie après l'ontologie lourde

De même, quand on définit à la fois *Paris* et *La ville lumière* comme capitales de la France, le raisonneur interprète Paris comme étant à la fois inclus dans Europe et dans France, puisque France est inclus dans Europe (notion de transitivité) et interprète Paris comme équivalent à *La Ville Lumière*.

3 Exploitation de l'ontologie

3.1 Implémentation de IModelFunction

Dans cette partie, nous avons implémenté des fonctions qui puissent être utilisées sur notre ontologie. L'objectif est ici d'utiliser les propriétés définie précédemment dans Protégé. Voici les différentes fonctions à implémenter et leur rôle, ainsi que des commentaires sur la manière dont elles ont été implémentées. Notre code est présenté dans les **Figures 9 et 10**.

Ces fonctions ont été implémentées à partir d'un schéma existant, contenu dans 'I Convenience Interface'. Lorsqu'une référence est faite à 'this.model', cela renvoie à l'une des fonctions décrite dans cette classe.

On note la différence entre les instances créées et leur URI. Un URI peut être associé à tout ce qui appartient à l'ontologie: propriétés, classes ou instances. Nous avons donc très largement utilisé la fonction **getEntityURI** pour récupérer l'URI de n'importe quel élément de l'ontologie en lui spécifiant seulement le label de l'entité recherchée. Il faut cependant penser à utiliser *get(0)* après chaque utilisation de cette fonction, car elle retourne une liste de *string* dont on veut le premier élément.

Nous allons maintenant voir plus en détail le rôle de ces fonctions et la manière dont elles ont été implémentées.

- **createPlace** : créé une instance de classe 'Lieu' et retourne l'URI de l'instance créée. Prend en paramètre le nom du lieu à créer.
  Implémentation: cette fonction utilise la brique de base 'createInstance' qui permet de créer n'importe quelle instance à partir d'un nom et d'un URI.

- **createInstant** : similaire à 'createPlace' mais pour créer un instant. Cette fonction prend en paramètre un 'timestampEntity'. Une particularité de cette fonction est que l'instant demandé ne sera pas créé s'il existe déjà un 'instant' correspondant au timestamp passé en paramètre.
  Implémentation: encore une fois, la fonction 'createInstance' a été utilisée exactement comme pour 'createPlace'. Une nouvelle notion est introduite ici: *les propriétés*. Nous avons utilisé la brique de base 'addDataPropertyToIndividual' pour lier le timestamp associé à un nouvel 'instant'.

- **getInstantURI** : prend en entrée un instant et renvoie son URI.
  Implémentation: nous avons ici manipulé la propriété 'aPourTimestamp'. La fonction recherche l'instant dont le timestamp correspond à celui passé en paramètre, et en retourne son URI. Pour faire la comparaison, la fonction de base 'hasDataPropertyValue' est utilisée.

- **getInstantTimestamp** : prend en paramètre un instant, et retourne son timestamp. S'il n'existe pas, la fonction renvoie 'null'.
  Implémentation: cette fonction est l'inverse de la précédente en quelque sorte. Nous utilisons ici 'listProperties' qui renvoie toutes les propriétés relatives à une entité. Nous avons utilisé une boucle for pour parcourir la liste de tous les 'instants' et checker à chaque itération si l'URI actuel correspond à l'URI recherché.

- **createObs** : permet de créet une 'observation' à partir des paramètres qui la caractérisent: son 'instant', et le type de paramètre qui est observé. La fonction renvoie l'URI de

l'observation créée.
Implémentation: on note une différence avec les fonctions précédentes, car ici des observations sont des entités plus complexes à manipuler car elles comportent de nombreuses propriétés. La fonction se décompose en deux parties. La première permet d'avoir accès à tous les URI nécessaires pour créer une observation en utilisant 'getEntityURI' et 'getInstantTimestamp'. La deuxième partie permet de linker les URI de l'observation. Un capteur est aussi présent, avec la fonction 'whichSensorDidIt' pour pouvoir linker la nouvelle opération à son capteur avec 'addObservationToSensor'.

Une fois toutes nos fonctions codées, nous avons vérifié leur bon fonctionnement avec les tests qui étaient fournis dan le projet. L'objectif est de savoir si notre modèle est capable de créer différentes instances de différentes classes. Les résultats des tests étaient tous concluants, et sont détaillés en **Figure 11**.

```java
package semantic.model;

import java.util.List;

public class DoItYourselfModel implements IModelFunctions
{
    IConvenienceInterface model;

    public DoItYourselfModel(IConvenienceInterface m) {
        this.model = m;
    }

    @Override
    public String createPlace(String name) {
        List<String> PlaceURI = this.model.getEntityURI("Lieu");

        return this.model.createInstance(name, PlaceURI.get(0));
    }

    @Override
    public String createInstant(TimestampEntity instant) {
        List<String> InstantURI = this.model.getEntityURI("Instant");
        List<String> TimestampURI = this.model.getEntityURI("a pour timestamp");

        String instant1 = this.model.createInstance("I1", InstantURI.get(0));
        this.model.addDataPropertyToIndividual(instant1, TimestampURI.get(0), instant.getTimeStamp());

        return instant1;
    }

    @Override
    public String getInstantURI(TimestampEntity instant) {

        String URIClassInstant = this.model.getEntityURI("Instant").get(0);

        String URITimestamp = this.model.getEntityURI("a pour timestamp").get(0);

        for (String instantURI:this.model.getInstancesURI(URIClassInstant)) {
            if (this.model.hasDataPropertyValue(instantURI, URITimestamp, instant.getTimeStamp())) {
                return instantURI;
            }
        }
        return null;
    }
```

Figure 9: Code pour les fonctions

```java
    @Override
    public String getInstantTimestamp(String instantURI)
    {
        String timestampPropertyURI = this.model.getEntityURI("a pour timestamp").get(0);
        for(List<String> propTuple : this.model.listProperties(instantURI))
        {
            if(propTuple.get(0).equals(timestampPropertyURI))
            {
                return propTuple.get(1);
            }
        }
        return null;
    }


    @Override
    public String createObs(String value, String paramURI, String instantURI) {
        String observationClassURI = this.model.getEntityURI("Observation").get(0);
        String uri_instance = this.model.createInstance("obs_"+instantURI, observationClassURI);
        String uri_dataValue = this.model.getEntityURI("a pour valeur").get(0);
        String uri_instantProp = this.model.getEntityURI("a pour date").get(0);
        String uri_paramProp = this.model.getEntityURI("mesure").get(0);
        String timestamp = getInstantTimestamp(instantURI);
        String sensor = this.model.whichSensorDidIt(timestamp, paramURI);
        this.model.addObservationToSensor(uri_instance, sensor);
        this.model.addDataPropertyToIndividual(uri_instance, uri_dataValue, value);
        this.model.addObjectPropertyToIndividual(uri_instance, uri_instantProp, instantURI);
        this.model.addObjectPropertyToIndividual(uri_instance, uri_paramProp, paramURI);
        return uri_instance;
    }
}
```

Figure 10: Code pour les fonctions (suite)

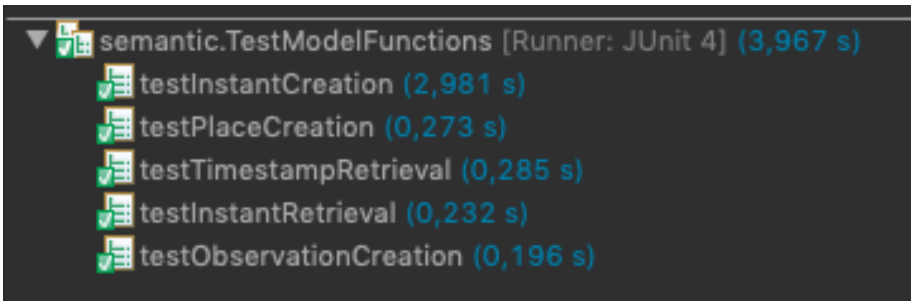

Figure 11: Résultat des tests

3.2 Implémentation de IControlFunctions

Pour cette partie, nous avons récupéré le code existant sur le dépôt Git par manque de temps. Le contrôleur utilise les fonctions du modèle, pour enrichir l'ensemble des données (dataset). Le but est ici d'instancier les observations. Le code est présenté en **Figure 12**.

3.3 Exploitation dans Protégé

Nous avons importé le modèle généré dans Protégé. Nous voulons maintenant vérifier si les capteurs ont été choisis judicieusement. Le SSN (une ontologie des capteurs) contient des classes et des propriétés pour décrire les conditions dans lesquelles un capteur doit être utilisé. L'arborescence est présentée en **Figure 13**.

```java
package semantic.controler;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import semantic.model.IConvenienceInterface;
import semantic.model.IModelFunctions;
import semantic.model.ObservationEntity;

public class DoItYourselfControl implements IControlFunctions
{
	IConvenienceInterface model;
	IModelFunctions customModel;

	public DoItYourselfControl(IConvenienceInterface model, IModelFunctions customModel)
	{
		this.model = model;
		this.customModel = customModel;
	}

	@Override
	public void instantiateObservations(List<ObservationEntity> obsList,
			String paramURI) {
		// The key of this map is the timestamp of instants individuals, to avoid creating
		// multiple individuals for the same instant
		Map<String, String> instantsMap = new HashMap<String, String>();
		// For each element of the list, create its representation in the KB
		int i =0;
		for(ObservationEntity oe : obsList)
		{
			i++;
			System.out.println("Instantiating observation "+i);
			String instantURI;
			if(!instantsMap.containsKey(oe.getTimestamp().getTimeStamp()))
			{
				instantURI = this.customModel.createInstant(oe.getTimestamp());
				instantsMap.put(oe.getTimestamp().getTimeStamp(), instantURI);
			}
			else
			{
				instantURI = instantsMap.get(oe.getTimestamp().getTimeStamp());
			}
			this.customModel.createObs(oe.getValue().toString(), paramURI, instantURI);
		}

	}
}
```

Figure 12: Code de la partie IControlFunctions

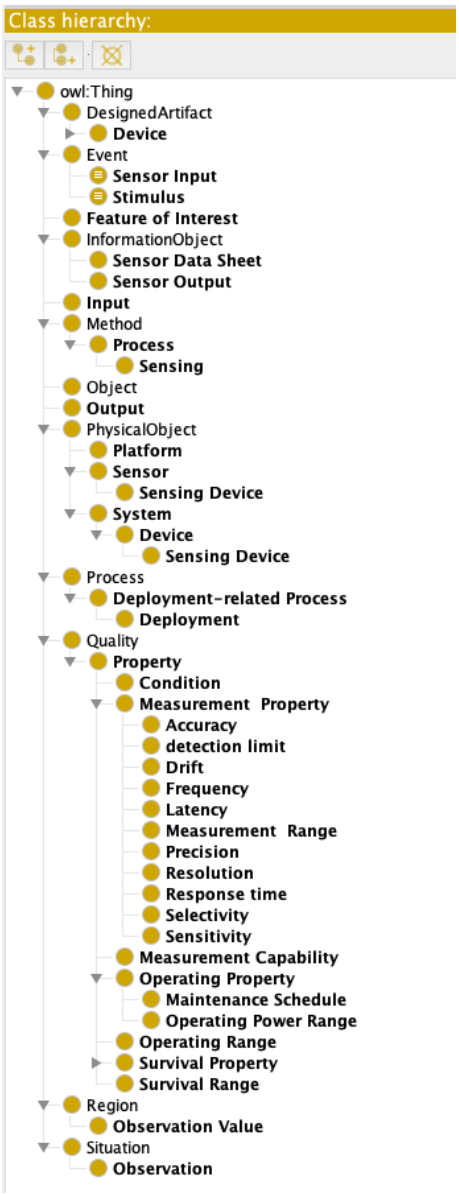

Figure 13: Arborescence dans Protégé

4 Conclusion

Ce TP nous a permis de comprendre la **création, l'utilisation et les spécificités d'une ontologie**. Avec le logiciel Protégé, nous avons pu voir l'interaction que la définition de propriétés peut causer au sein d'une ontologie. Le raisonneur est en charge de ces déductions logiques sur la base de ce qui a été défini comme propriétés.

Une notion fondamentale est la différence entre object property et data property. Dans Protégé, il existe différents onglets pour créer des object properties et des data properties. Si une propriété doit relier des **individus à des individus**, elle doit être une **object property**, et si elle relie des **individus à des valeurs**, elle doit être une **data property**.

La seconde partie nous a permis de voir un **exemple d'exploitation d'ontologie**. Nous avons vu avec la définition des différentes fonctions qu'il est nécessaire de créer un modèle d'ontologie pour l'exploiter. Nous n'avons pas eu le temps de finir l'implémentation, mais nous avons pu mener une réflexion à partir du code fourni.