PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
MAESTRÍA EN INFORMÁTICA

INGENIERÍA Y GESTIÓN DEL CONOCIMIENTO
Laboratorio 3 - RDFS
(Segundo Semestre del 2020)

Objetivos de aprendizaje:

- Implementar inferencias usando `subPropertyOf` en RDFS.
- Implementar inferencias usando `domain` y `range` en RDFS.
- Modelamiento de patrones en RDFS para representar intersección y unión

---

1. Introducción

En este laboratorio se busca que el alumno implemente inferencias a partir de estructuras RDFS para resolver problemas de la Ingeniería del Conocimiento. Se busca que el alumno pueda crear, salvar, leer y consultar modelos RDFS usando un lenguaje de programación, aplicando de esta manera los conceptos vistos en la sesión teórica.

2. Métodos y procedimientos

Se utilizarán las siguientes herramientas:

- Como lenguaje de programación se utilizará Java.
- Como Entorno de Desarrollo Integrado (IDE) se utilizará Netbeans 8.2 el cual puede ser descargado a partir de la URL `https://netbeans.org/downloads/8.2/`.
- Para la manipulación del RDF y RDFS se utilizará el API de Apache Jena, el cual se encuentra en la URL `https://jena.apache.org/`. El alumno podrá encontrar un tutorial de RDF usando Jena en la URL `https://jena.apache.org/tutorials/rdf_api.html`.
- Los ejemplos están basados en el libro *"Semantic web for the working ontologist: effective modeling in RDFS and OWL"* de Allemang y Hendler. Asimismo todas las imágenes mostradas en este documento han sido tomadas de este libro. El libro se puede encontrar en la URL `https://www.sciencedirect.com/book/9780123859655/semantic-web-for-the-working-ontologist`.

Parte I
Inferencia en RDFS - `subPropertyOf`

3. Creación de jerarquía de propiedades RDFS

Esta parte tiene por objetivo la implementación de inferencias usando propiedades. En RDFS las propiedades pueden tener subpropiedades las cuales se implementan usando la propiedad RDFS `subPropertyOf`. Esta propiedad se usa para indicar que todos los recursos relacionados con una propiedad también están relacionados por otra.

En la figura 2 se encuentra una jerarquía de propiedades que se usará para demostrar cómo funcionan las inferencias. En ella se puede apreciar que en la categoría más alta se encuentra la propiedad `worksFor`. Esto significa que las propiedades que se encuentra en el segundo nivel de jerarquía, `contractsTo` y `isEmployedBy`, son también `worksFor`.

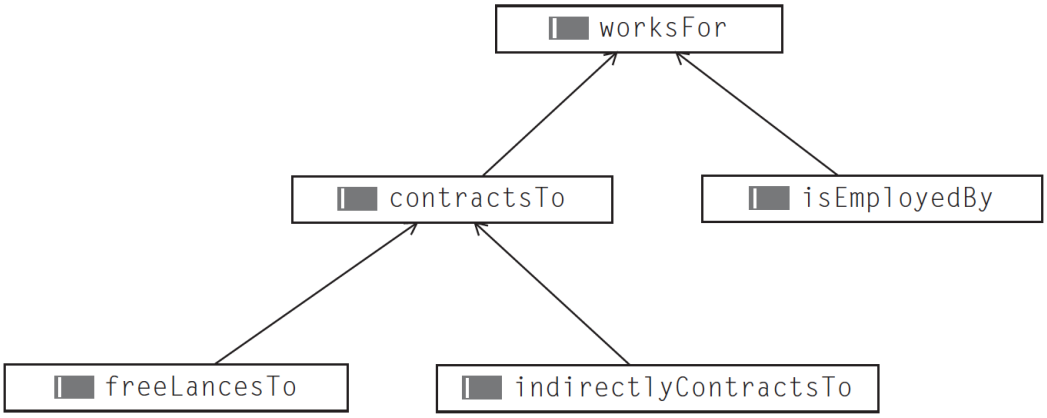

Figura 1: Jerarquía de propiedades en RDFS.

Dado que se crearán varias propiedades, se va a utilizar un método que facilitará este propósito. En el programa 1 se aprecia el método `crearPropiedad`. Este método, tal como su nombre lo indica, permite crear una propiedad (`Property`) en el modelo (`Model`). En este caso `uri_base` representa el URI base del modelo a utilizar y el parámetro `id`, el identificador de la propiedad. El método de modelo que permite crear la propiedad es `createProperty`.

Programa 1: Creando propiedades en RDF en Jena

```
...
    private static Property crearPropiedad(String uri_base, String id, Model
        model) {
        return model.createProperty(uri_base, id);
    }
...
```

En el programa 2 se utiliza el método `crearPropiedad` para crear todas las propiedades requeridas. Como URI base se usará `http://www.pucp.edu.pe/`. Este es un URI de referencia y usted la podrá cambiar a su conveniencia. Con el código del programa 2 se crean las propiedades, todavía no se realiza ninguna relación entre ellas.

Programa 2: Creando propiedades en RDF en Jena

```
...
    Model model = ModelFactory.createDefaultModel();
        String uri = "http://www.pucp.edu.pe/";
        String prefijo = "pucp";
        model.setNsPrefix(prefijo, uri);

        Property worksFor = crearPropiedad(uri, "worksFor", model);
        Property contractsTo = crearPropiedad(uri, "contractsTo", model);
        Property isEmployedBy = crearPropiedad(uri, "isEmployedBy", model);
        Property freeLancesTo = crearPropiedad(uri, "freeLancesTo", model);
        Property indirectlyContractsTo = crearPropiedad(uri,
            "indirectlyContractsTo", model);
...
```

Para crear la jerarquía de propiedades lo único que se requiere hacer es agregar estas declaraciones como tripleta. Para agregar una triplela en RDF se utiliza el método `add` que se encuentra en la clase `Model`. Recuerde que una tripleta en RDF está formada por un sujeto, un predicado y un objeto. El predicado en este caso es la propiedad RDFS `subPropertyOf`. En esta caso en particular, no es necesario crear la propiedad pues Jena facilita esta propiedad a través de `RDFS.subPropertyOf`. En el programa 3 se puede apreciar el código que permite la creación de la jerarquía de propiedades que se presenta en la figura 2.

Programa 3: Creando la jerarquía de propiedades en RDFS en Jena

```
...
        model.add(contractsTo, RDFS.subPropertyOf, worksFor);
        model.add(isEmployedBy, RDFS.subPropertyOf, worksFor);
        model.add(freeLancesTo, RDFS.subPropertyOf, contractsTo);
        model.add(indirectlyContractsTo, RDFS.subPropertyOf, contractsTo);
...
```

4. Creación instancias de datos usando las propiedades RDFS

En la sección 3 se ha realizado la creación de la jerarquía de propiedades. Para poder realizar inferencias se requiere de instancias vinculados a dichas propiedades. Para probar las inferencias, se usarán las declaraciones definidas en la figura 2. En dicha figura tanto `Goldman`, `Spence`, `Long` así como `TheFirm` son recursos RDF.

```
:Goldman :isEmployedBy :TheFirm.
:Spence :freeLancesTo :TheFirm.
:Long :indirectlyContractsTo :TheFirm.
```

Figura 2: Instancia de datos sobre trabajadores en RDFS.

Dado que se crearán varios recursos, se va a utilizar un método que facilitará este propósito. En el programa 4 se aprecia el método `crearRecurso`. Este método, tal como su nombre lo indica, permite crear un recurso (`Resource`) en el modelo (`Model`). En este caso se le pasa el URI del recurso a crear en el parámetro `id`.

Programa 4: Creando recursos en RDF en Jena

```
...
    private static Resource crearRecurso(String id, Model model) {
        return model.createResource(id);
    }
...
```

En el programa 5 se utiliza el método `crearRecurso` para crear todos los recursos requeridos. Como URI base se usará `http://www.pucp.edu.pe/`. Esta es un URI de referencia y usted la podrá cambiar a su conveniencia. Con el código del programa 5 se crean los recursos, todavía no se realiza ninguna relación a ninguna propiedad.

Programa 5: Creando recursos en RDF en Jena

```
...
        Resource TheFirm = crearRecurso(uri + "TheFirm", model);
        Resource Goldman = crearRecurso(uri + "Goldman", model);
        Resource Spence = crearRecurso(uri + "Spence", model);
        Resource Long = crearRecurso(uri + "Long", model);
...
```

Para relacionar un recurso con otro, al igual que con todas las afirmaciones en RDF, lo único que se requiere hacer es agregar estas declaraciones como tripleta. El predicado en este caso serán las propiedades solicitadas en la figura 2. En el programa 6 se puede apreciar el código que permite la inclusión de las declaraciones solicitadas.

Programa 6: Creando instancia de datos sobre trabajadores en RDFS en Jena

```
...
        model.add(Goldman, isEmployedBy, TheFirm);
        model.add(Spence, freeLancesTo, TheFirm);
```

```
        model.add(Long, indirectlyContractsTo, TheFirm);
...
```

Finalmente se procede a salvar el archivo con el nombre de `relaciones_trabajadores.rdf`. Este archivo será consultado posteriormente.

Programa 7: Grabando el esquema RDFS en Jena

```
...
        FileOutputStream output = null;
        try {
            output = new FileOutputStream("relaciones_trabajadores.rdf");
        } catch (FileNotFoundException e) {
            System.out.println("Ocurrió un error al crear el archivo.");
        }
        model.write(output, "RDF/XML-ABBREV");
...
```

5. Consultando relaciones de trabajadores en RDFS

En esta sección se realizarán las consulta para verificar si es que se puede inferir información a partir de la semántica de las relaciones de las propiedades. En el programa 8 se encuentra el código que permite consultar si `Goldman worksFor TheFirm`. En teoría como `Goldman isEmployedBy TheFirm` y dado que `isEmployedBy` es subpropiedad de `worksFor`, la afirmación en cuestión debería ser verdadera. Para la implementación de la consulta se usa el método `listStatements` el cual recibe como parámetro un selector que contiene la tripleta a consultar. Para recuperar un recurso (`Resource`) a partir del modelo (`Model`) se usa el método `getResource`. Para recuperar una propiedad (`Property`) a partir del modelo (`Model`) se usa el método `getProperty`. En ambos caso se le debe pasar la URI como parámetro.

Programa 8: Consultando relaciones de trabajadores en RDFS usando Jena

```
...
        String inputFileName = "relaciones_trabajadores.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);

        String resourceURI = model.expandPrefix("pucp:Goldman");
        Resource goldman = model.getResource(resourceURI);

        resourceURI  = model.expandPrefix("pucp:TheFirm");
        Resource TheFirm = model.getResource(resourceURI);
        String propertyURI = model.expandPrefix("pucp:worksFor");
        Property worksFor = model.getProperty(propertyURI);

        Selector selector = new SimpleSelector(goldman, worksFor, TheFirm);
        StmtIterator iter = model.listStatements(selector);
        while (iter.hasNext()) {
            System.out.println(iter.nextStatement().toString());
        }
...
```

Para poner en práctica

- ¿Qué se obtiene como resultado?
- ¿El resultado es consistente con la inferencia que se esperaba?

Para que Jena produzca las inferencias estudiadas, es necesario realizar un pequeño cambio en el programa. En el programa 9 usted podrá ver en la línea 5 la creación de un modelo de inferencia, el cual está implementado por la clase `InfModel`. Esta clase en su creación (`ModelFactory.createRDFSModel`) recibe

el modelo leído del archivo RDF. Al momento de realizar la consulta a través del método `listStatements`, se utiliza el modelo de inferencia (vea la línea 16 del programa 9).

Programa 9: Consultando relaciones de trabajadores en RDFS usando Jena

```
1  ...
2          String inputFileName = "relaciones_trabajadores.rdf";
3          Model model = ModelFactory.createDefaultModel();
4          model.read(inputFileName);
5          InfModel inf = ModelFactory.createRDFSModel(model);
6
7          String resourceURI = model.expandPrefix("pucp:Goldman");
8          Resource goldman = model.getResource(resourceURI);
9
10         resourceURI  = model.expandPrefix("pucp:TheFirm");
11         Resource TheFirm = model.getResource(resourceURI);
12         String propertyURI = model.expandPrefix("pucp:worksFor");
13         Property worksFor = model.getProperty(propertyURI);
14
15         Selector selector = new SimpleSelector(goldman, worksFor, TheFirm);
16         StmtIterator iter = inf.listStatements(selector);
17         while (iter.hasNext()) {
18             System.out.println(iter.nextStatement().toString());
19         }
20 ...
```

Para poner en práctica

Verifique si las siguientes afirmaciones (tripletas) se cumplen:

- `Goldman worksFor TheFirm`.
- `Spence contractsTo TheFirm`.
- `Long contractsTo TheFirm`.
- `Spence worksFor TheFirm`.
- `Long worksFor TheFirm`.

Para poner en práctica

- Consulte todas las personas que trabajan (`worksFor`) para la organización (`TheFirm`).
- Consulte todas las personas que son contratadas con (`contractsTo`) la organización (`TheFirm`).
- Consulte todas las personas que son empleadas por (`isEmployedBy`) la organización (`TheFirm`).

Parte II
Inferencia en RDFS - `domain` y `range`

6. Ontología en RDFS

Esta parte tiene por objetivo la implementación de inferencias usando el dominio y rango. En RDFS el dominio y rango proporcionan información sobre cómo se usará determinada propiedad. Para especi-

ficarlas en RDFS se utilizan las propiedades `domain` y `range`. Estas propiedades se usan para indicar el dominio y rango respectivamente de determinada propiedad.

En la figura 3 se puede apreciar una pequeña ontología que se usará para demostrar cómo funcionan las inferencias usando dominio y rango. En ella se pueden apreciar dos clases (`Woman` y `MarriedWoman`) y una propiedad (`hasMaidenName`). Se especifica que el dominio de la propiedad `hasMaidenName` es la clase `MarriedWoman`. Además la clase `MarriedWoman` es subclase de `Woman`.

Figura 3: Ontología representada en RDFS.

En el programa 10 (ver líneas 8–10), se utilizan los métodos `crearRecurso` y `crearPropiedad`, especificados previamente, para crear los recursos y las propiedades requeridas. Como URI base se usará `http://www.pucp.edu.pe/`. Este es un URI de referencia y usted la podrá cambiar a su conveniencia.

Para crear la ontología solicitada lo único que se requiere hacer es agregar las declaraciones que la definen. Estas declaraciones nuevamente serán tripletas RDF. En la línea 12 del programa 10 se define la subclase de forma muy similar a como se realizó en la sesión anterior. Se utiliza el método `RDFS.subClassOf` que ya retorna la propiedad RDFS requerida. En la línea 13 se define el dominio de la propiedad `hasMaidenName`. Para esto se utiliza el método `RDFS.domain` que ya retorna la propiedad RDFS requerida.

Programa 10: Creando la ontología en RDFS usando Jena

```
1  ...
2      public static void main(String[] args) {
3          Model model = ModelFactory.createDefaultModel();
4          String uri = "http://www.pucp.edu.pe/";
5          String prefijo = "pucp";
6          model.setNsPrefix(prefijo, uri);
7
8          Resource Woman = crearRecurso(uri + "Woman", model);
9          Resource MarriedWoman = crearRecurso(uri + "MarriedWoman", model);
10         Property hasMaidenName = crearPropiedad(uri, "hasMaidenName", model);
11
12         model.add(MarriedWoman, RDFS.subClassOf, Woman);
13         model.add(hasMaidenName, RDFS.domain, MarriedWoman);
14 ...
```

7. Afirmación en RDFS

Para probar las inferencias, se usarán las afirmación definidas en la figura 4. En dicha figura `Karen` y `Stephens` son recursos RDF. `hasMaidenName` es la propiedad creada previamente.

```
:Karen :hasMaidenName "Stephens".
```

Figura 4: Afirmación en RDFS.

En el programa 11 se utiliza el método `crearRecurso` para crear todos los recursos requeridos. La afirmación se realiza nuevamente a través de una triplea. Para agregar una triplela en RDF se utiliza el método `add` que se encuentra en la clase `Model`.

Programa 11: Realizando afirmación en RDF usando Jena

```
...
        Resource Karen = crearRecurso(uri + "Karen", model);
        Resource Stephens = crearRecurso(uri + "Stephens", model);
        model.add(Karen, hasMaidenName, Stephens);
...
```

8. Grabando archivo RDF

Debido a que se grabarán varios esquemas RDF en los ejemplos siguientes, se va a utilizar un método que facilitará este propósito. En el programa 12 se aprecia el método `grabarRDF`. Este método, tal como su nombre lo indica, permite grabar una esquema RDF en un archivo a partir de un modelo (`Model`). En este caso `nmRDFFile` representa al nombre del archivo de salida.

Programa 12: Creando método para salvar archivos RDF en Jena

```
...
    public static void grabarRDF(String nmRDFFile, Model model) {
        FileOutputStream output = null;
        try {
            output = new FileOutputStream(nmRDFFile);
        } catch (FileNotFoundException e) {
            System.out.println("Ocurrió un error al crear el archivo.");
        }
        model.write(output, "RDF/XML-ABBREV");
    }
...
```

En el programa 13 se presenta el código completo que crea la ontología, realiza la afirmación anteriormente mencionada y graba el esquema RDF en un archivo.

Programa 13: Salvando RDF en un archivo usando Jena

```
...
    public static void main(String[] args) {
        Model model = ModelFactory.createDefaultModel();
        String uri = "http://www.pucp.edu.pe/";
        String prefijo = "pucp";
        model.setNsPrefix(prefijo, uri);

        Resource Woman = crearRecurso(uri + "Woman", model);
        Resource MarriedWoman = crearRecurso(uri + "MarriedWoman", model);
        Property hasMaidenName = crearPropiedad(uri, "hasMaidenName", model);

        model.add(MarriedWoman, RDFS.subClassOf, Woman);
        model.add(hasMaidenName, RDFS.domain, MarriedWoman);

        Resource Karen = crearRecurso(uri + "Karen", model);
        Resource Stephens = crearRecurso(uri + "Stephens", model);
        model.add(Karen, hasMaidenName, Stephens);

        grabarRDF("nombre_soltera.rdf", model);
    }
...
```

9. Consultando si determinada mujer es casada RDFS

Lo único que se sabe de Karen es que tiene un nombre de soltera (`hasMaidenName`). ¿Es esto suficiente para inferir que Karen es una mujer casada?, ¿es esto suficiente para inferir que Karen es una mujer? El programa 14 intenta realizar estas consultas. Note que para saber que si un elemento es parte de un conjunto, se utiliza la propiedad `RDF.type`. En caso se requiera preguntar si `Karen` es un elemento de la clase `MarriedWoman`, se debe usar la tripleta `Karen RDF.type MarriedWoman` en el selector.

Programa 14: Consultando si una determinada mujer es casada en Jena

```
...
        String inputFileName = "nombre_soltera.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);

        String resourceURI = model.expandPrefix("pucp:Karen");
        Resource Karen = model.getResource(resourceURI);
        resourceURI = model.expandPrefix("pucp:MarriedWoman");
        Resource MarriedWoman = model.getResource(resourceURI);

        Selector selector = new SimpleSelector(Karen, RDF.type, MarriedWoman);
        StmtIterator iter = inf.listStatements(selector);
        while (iter.hasNext()) {
            System.out.println(iter.nextStatement().toString());
        }
...
```

Como simplemente queremos verificar si una afirmación es cierta, crearemos un método que nos ayude a reconocer si una afirmación existe en un modelo inferido. En el programa 15 se presenta el método `existenAfirmaciones` que encapsula esto. Recibe como parámetros el modelo de inferencia (`InfModel`), el sujeto (`Resource`), predicado (`Property`) y objeto a consultar (`Resource`).

Programa 15: Método para verificar si existen afirmaciones en un esquema RDFS

```
...
    public static Boolean existenAfirmaciones(InfModel inf,  Resource Sujeto,
       Property predicado, Resource objeto){
        Boolean hayAfirmaciones;
        Selector selector = new SimpleSelector(Sujeto, predicado, objeto);
        StmtIterator iter = inf.listStatements(selector);
        hayAfirmaciones = iter.hasNext();
        return hayAfirmaciones;
    }
...
```

Usando el método `existenAfirmaciones` verificamos si la afirmación Karen (`Karen`) pertenece (`RDF.type`) a la clase Mujer Casada (`MarriedWoman`) es cierta. Esto se puede apreciar en el programa 16.

Programa 16: Consultando si una determinada mujer es casada en Jena

```
...
        String inputFileName = "nombre_soltera.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);

        String resourceURI = model.expandPrefix("pucp:Karen");
        Resource Karen = model.getResource(resourceURI);
        resourceURI = model.expandPrefix("pucp:MarriedWoman");
        Resource MarriedWoman = model.getResource(resourceURI);

        if (existenAfirmaciones(inf, Karen, RDF.type, MarriedWoman)) {
```

```
            System.out.println("La afirmación es cierta");
        } else {
            System.out.println("La afirmación no es cierta");
        }
    }
...
```

Para poner en práctica

- Verifique si es verdad que `Karen` es reconocida como `Woman`. Analice su respuesta y justifique en términos de la Web Semántica y RDFS.

En el contexto de la Web Semántica no solo basta con hacer buenas inferencias sino demostrar la validez de éstas. Muchos de los sistemas en la Web Semántica poseen mecanismos que permiten verificar el "razonamiento" del motor de inferencia. Jena no es ajena a este mecanismo y es posible mostrar las derivaciones que se hace al momento de hacer una inferencia. En el programa 17 se ha creado un método que presenta todas las derivaciones de una inferencia. Recibe como parámetros el modelo de inferencia (`InfModel`), el sujeto (`Resource`), predicado (`Property`) y objeto a consultar (`Resource`). A través del método `getDerivation` del modelo de inferencia se obtiene las derivaciones que son mostradas en este caso en la salida estándar pero las clases de Jena están preparadas para enviarlas a un archivo también.

Programa 17: Método para mostrar derivaciones en Jena

```
...
    public static void mostrarDerivaciones(InfModel inf,  Resource Sujeto,
       Property predicado, Resource objeto){
        PrintWriter out = new PrintWriter(System.out);
        for (StmtIterator i = inf.listStatements(Sujeto, predicado, objeto);
           i.hasNext();) {
            Statement s = i.nextStatement();
            System.out.println("Statement is " + s);
            for (Iterator id = inf.getDerivation(s); id.hasNext();) {
                Derivation deriv = (Derivation) id.next();
                deriv.printTrace(out, true);
            }
        }
        out.flush();
    }
...
```

Para usar el método `mostrarDerivaciones` solo se debe tener un pequeño cuidado. Hay que informarle al modelo de inferencia que guarde las derivaciones. Esto evidentemente hará el razonamiento de motor se demoré un poco más, dependiendo de las reglas de inferencia que guardará. En el programa 18 se puede apreciar en la línea 7 como se define con `true` en el método `setDerivationLogging`. Sin esta configuración las derivaciones no serán guargadas.

Programa 18: Mostrando las justificaciones para una afirmación en Jena

```
...
    public static void main(String[] args) {
        String inputFileName = "nombre_soltera.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);
        inf.setDerivationLogging(true);

        String resourceURI = model.expandPrefix("pucp:Karen");
        Resource Karen = model.getResource(resourceURI);
        resourceURI = model.expandPrefix("pucp:MarriedWoman");
        Resource MarriedWoman = model.getResource(resourceURI);

```

```
14          if (existenAfirmaciones(inf, Karen, RDF.type, MarriedWoman)) {
15              System.out.println("La afirmación es cierta");
16              mostrarDerivaciones(inf, Karen, RDF.type, MarriedWoman);
17          } else {
18              System.out.println("La afirmación no es cierta");
19          }
20      }
21  ...
```

```
CONSTRUCT {?P rdfs:domain ?C .}
WHERE {?P rdfs:domain ?D .
       ?D rdfs:subClassof ?C .}
```

Figura 5: Patrón de Inferencia en RDFS usando `domain` y `subClassOf`.

Para poner en práctica

- Analice los resultados de programa 18 y verifique si el constructo presentado en la figura 5 se cumple.

Parte III

Patrones en RDFS - Intersección de conjuntos

Esta parte tiene por objetivo la implementación patrones en RDFS para intersección de conjuntos. En RDFS no existen primitivas para modelar la intersección de conjuntos pero se pueden simular. Este patrón es muy simple, basta hacer que una misma clase A sea subclase de otras dos B y C, de esta manera se podrá simular que A es la intersección de los conjuntos B y C.

En la figura 6 se puede apreciar el uso de este patrón para simular la intersección. En ella se pueden apreciar tres clases: `Surgeon`, `Staff` y `Physician`. En este caso el patrón muestra que `Surgeon` es la intersección de los conjuntos `Staff` y `Physician`.

```
:Surgeon rdfs:subClassOf :Staff.
:Surgeon rdfs:subClassOf :Physician.
:Kildare rdf:type :Surgeon.
```

Figura 6: Estructura RDFS usando intersección de conjuntos.

En el programa 19 se utiliza el método `crearRecurso` para crear todos los recursos requeridos. Como URI base se usará `http://www.pucp.edu.pe/`. Este es un URI de referencia y usted la podrá cambiar a su conveniencia.

Programa 19: Creando recursos en RDF en Jena

```
1   ...
2           Model model = ModelFactory.createDefaultModel();
3           String uri = "http://www.pucp.edu.pe/";
4           String prefijo = "pucp";
5           model.setNsPrefix(prefijo, uri);
6
```

```
        Resource Staff = crearRecurso(uri + "Staff", model);
        Resource Physician = crearRecurso(uri + "Physician", model);
        Resource Surgeon = crearRecurso(uri + "Surgeon", model);
        Resource Kildare = crearRecurso(uri + "Kildare", model);
...
```

Para relacionar un recurso con otro, al igual que con todas las afirmaciones en RDF, lo único que se requiere hacer es agregar estas declaraciones como tripleta. El predicado en este caso serán las propiedades solicitadas en la figura 6. En el programa 20 se puede apreciar el código que permite la inclusión de las declaraciones solicitadas y graba el RDF en el archivo `cirujanos.rdf`.

Programa 20: Creando y salvando el modelo en Jena

```
...
        model.add(Surgeon, RDFS.subClassOf, Staff);
        model.add(Surgeon, RDFS.subClassOf, Physician);
        model.add(Kildare, RDF.type, Surgeon);

        grabarRDF("cirujanos.rdf", model);
...
```

Para mostrar las declaraciones de una consulta, nuevamente crearemos un método que nos ayude con esto. En el programa 21 se presenta el método `mostrarDeclaraciones`. Recibe como parámetros el modelo de inferencia (`InfModel`), el sujeto (`Resource`), predicado (`Property`) y objeto a consultar (`Resource`).

Programa 21: Método para mostrar declaraciones RDF en Jena

```
...
    public static void mostrarDeclaraciones(InfModel inf, Resource Sujeto,
        Property predicado, Resource objeto) {
        Selector selector = new SimpleSelector(Sujeto, predicado, objeto);
        StmtIterator iter = inf.listStatements(selector);
        while (iter.hasNext()) {
            System.out.println(iter.nextStatement().toString());
        }
    }
...
```

En esta sección se realizarán las consulta para verificar a qué clases pertenece el recurso `Kildare`. En teoría como `Kildare rdf:type Surgeon` y dado que `Surgeon` es el conjunto intersección de `Staff` y `Physician`, RFS debería inferir que `Kildare` también es un elemento de `Staff` y `Physician` respectivamente. El programa 22 realiza dicha consulta. Verifique si se cumple.

Programa 22: Consultando a qué conjuntos pertenece un recurso usando Jena

```
...
        String inputFileName = "cirujanos.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);

        String resourceURI = model.expandPrefix("pucp:Kildare");
        Resource Kildare = model.getResource(resourceURI);

        mostrarDeclaraciones(inf, Kildare, RDF.type, null);
...
```

Para poner en práctica

Verifique si las siguientes afirmaciones (tripletas) se cumplen:

- `Kildare rdf:type Staff.`
- `Kildare rdf:type Physician.`

Se hará ahora un experimento para determinar si RDFS puede hacer la inferencia de la intersección si se definen las instancias de otra manera. En programa 23 se hace la implementación de un modelo en RDFS esquema similar al de la figura 6 y lo graba en el archivo `cirujanos2.rdf`, ¿cuál es la diferencia?

Programa 23: Creando recursos en RDF en Jena

```
...
    public static void main(String[] args) {
        String NS = "http://www.pucp.edu.pe/informatica#";
        Model model = ModelFactory.createDefaultModel();

        Resource Staff = crearRecurso(NS, "Staff", model);
        Resource Physician = crearRecurso(NS, "Physician", model);
        Resource Surgeon = crearRecurso(NS, "Surgeon", model);
        Resource Kildare = crearRecurso(NS, "Kildare", model);

        model.add(Surgeon, RDFS.subClassOf, Staff);
        model.add(Surgeon, RDFS.subClassOf, Physician);
        model.add(Kildare, RDF.type, Staff);
        model.add(Kildare, RDF.type, Physician);

        grabarRDF("cirujanos2.rdf", model);
    }
...
```

En el programa 24 se hace la misma inferencia pero ahora usando el archivo `cirujanos2.rdf`. ¿Se llega al mismo resultado?

Programa 24: Consultando a qué conjuntos pertenece un recurso usando Jena

```
...
        String inputFileName = "cirujanos2.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);

        String resourceURI = model.expandPrefix("pucp:Kildare");
        Resource Kildare = model.getResource(resourceURI);

        mostrarDeclaraciones(inf, Kildare, RDF.type, null);
...
```

Para poner en práctica

- Analice los resultados de programa 24 y verifique si es verdad que se cumple la siguiente afirmación `Kildare rdf:type Surgeon`.
- Muestre las derivaciones para justificar su análisis.

Parte IV
Patrones en RDFS - Intersección de propiedades

Esta parte tiene por objetivo la implementación patrones en RDFS para intersección de propiedades. En RDFS no existen primitivas para modelar la intersección de propiedades pero se pueden simular. Este patrón es muy simple, basta hacer que una misma propiedades P_A sea subpropiedades de otras dos P_B y P_C, de esta manera se podrá simular que P_A es la intersección de los conjuntos P_B y P_C.

En la figura 7 se puede apreciar el uso de este patrón para simular la intersección. En ella se pueden apreciar tres propiedades: `lodgedIn`, `billedFor` y `assignedTo`. En este caso el patrón muestra que `lodgedIn` es la intersección de loas propiedades `billedFor` y `assignedTo`.

```
:lodgedIn rdfs:subPropertyOf :billedFor.
:logdedIn rdfs:subPropertyOf :assignedTo.
```

Figura 7: Estructura RDFS usando intersección de propiedades.

En el programa 25 se utiliza el método `crearPropiedad` para crear todos las propiedades requeridas. Como URI base se usará `http://www.pucp.edu.pe/`. Este es un URI de referencia y usted la podrá cambiar a su conveniencia.

Para relacionar una propiedad con otra, al igual que con todas las afirmaciones en RDF, lo único que se requiere hacer es agregar estas declaraciones como tripleta. El predicado en este caso serán las propiedades solicitadas en la figura 7. En el programa 25 se puede apreciar el código que permite la inclusión de las declaraciones solicitadas.

Programa 25: Creando recursos en RDF en Jena

```
1  ...
2          Model model = ModelFactory.createDefaultModel();
3          String uri = "http://www.pucp.edu.pe/";
4          String prefijo = "pucp";
5          model.setNsPrefix(prefijo, uri);
6
7          Property billedFor = crearPropiedad(uri, "billedFor", model);
8          Property assignedTo = crearPropiedad(uri, "assignedTo", model);
9          Property lodgedIn = crearPropiedad(uri, "lodgedIn", model);
10
11         model.add(lodgedIn, RDFS.subPropertyOf, billedFor);
12         model.add(lodgedIn, RDFS.subPropertyOf, assignedTo);
13 ...
```

Para probar las inferencias, se usarán las afirmación definidas en la figura 8. En dicha figura `Marcus lodgedIn Room101`. En la figura `Marcus` y `Room101` son recursos mientras que `lodgedIn` es la propiedad creada previamente.

```
:Marcus :lodgedIn :Room101.
```

Figura 8: Afirmación en RDFS.

En el programa se implementa la afirmación realizada en el figura 8 en el modelo y se graba este con el nombre `habitacion.RDF`.

Programa 26: Creando y salvando el modelo en Jena

```
1  ...
2          Resource Marcus =crearRecurso(NS, "Marcus", model);
```

```
        Resource Room101 =crearRecurso(NS, "Room101", model);
        model.add(Marcus, lodgedIn, Room101);

        grabarRDF("habitacion.RDF", model);
...
```

En el programa 27 se verifica si que se cumple la siguiente afirmación `Marcus billedFor Room101`. ¿Es coherente el resultado con lo que se espera del modelo de inferencia de RDFS?

Programa 27: Verificando una afirmación RDF en Jena

```
...
        String inputFileName = "habitacion.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);
        inf.setDerivationLogging(true);

        String resourceURI = model.expandPrefix("pucp:Marcus");
        Resource Marcus = model.getResource(resourceURI);
        resourceURI = model.expandPrefix("pucp:Room101");
        Resource Room101 = model.getResource(resourceURI);
        String propertyURI = model.expandPrefix("pucp:billedFor");
        Property billedFor = model.getProperty(propertyURI);

        if (existenAfirmaciones(inf, Marcus, billedFor, Room101)) {
            System.out.println("La afirmación es cierta");
            mostrarDerivaciones(inf, Marcus, billedFor, Room101);
        } else {
            System.out.println("La afirmación no es cierta");
        }
...
```

Para poner en práctica

- Verifique si es verdad que se cumple la siguiente afirmación `Marcus assignedTo Room101`.
- Muestre las derivaciones y analice el resultado.

Parte V

Patrones en RDFS - Unión de conjuntos

Esta parte tiene por objetivo la implementación patrones en RDFS para unión de conjuntos. En RDFS no existen primitivas para modelar la unión de conjuntos pero se pueden simular. Este patrón es muy simple, basta hacer que una misma clase A sea superclase de otras dos B y C, de esta manera se podrá simular que A es la unión de los conjuntos B y C.

En la figura 10 se puede apreciar el uso de este patrón para simular la intersección. En ella se pueden apreciar tres clases: `AllStarCandidate`, `MVP` y `TopScorer`. En este caso el patrón muestra que `AllStarCandidate` es la unión de los conjuntos `MVP` y `TopScorer`.

```
:MVP rdfs:subClassOf :AllStarCandidate.
:TopScorer rdfs:subClassOf :AllStarCandidate.
```

Figura 9: Estructura RDFS usando unión de conjuntos.

En la figura 10 se presenta la afirmación que se usará para realizar la inferencia posteriormente.

```
:Reilly rdf:type :MVP.
:Kaneda rdf:type :TopScorer.
```

Figura 10: Afirmación en RDFS.

En el programa 28 se utiliza el método `crearRecurso` para crear todos los recursos requeridos. Como URI base se usará `http://www.pucp.edu.pe/`. Este es un URI de referencia y usted la podrá cambiar a su conveniencia.

Para relacionar un recurso con otro, al igual que con todas las afirmaciones en RDF, lo único que se requiere hacer es agregar estas declaraciones como tripleta. El predicado en este caso serán las propiedades solicitadas en la figura 10. En el programa 28 se puede apreciar el código que permite la inclusión de las declaraciones solicitadas y graba el RDF en el archivo `deportes.rdf`.

Programa 28: Creando recursos en RDF en Jena

```
...
        Model model = ModelFactory.createDefaultModel();
        String uri = "http://www.pucp.edu.pe/";
        String prefijo = "pucp";
        model.setNsPrefix(prefijo, uri);

        Resource AllStarCandidate = crearRecurso(uri + "AllStarCandidate",
            model);
        Resource MVP = crearRecurso(uri + "MVP", model);
        Resource TopScorer = crearRecurso(uri + "TopScorer", model);

        model.add(MVP, RDFS.subClassOf, AllStarCandidate);
        model.add(TopScorer, RDFS.subClassOf, AllStarCandidate);

        Resource Reilly = crearRecurso(uri + "Reilly", model);
        Resource Kaneda = crearRecurso(uri + "Kaneda", model);

        model.add(Reilly, RDF.type, MVP);
        model.add(Kaneda, RDF.type, TopScorer);

        grabarRDF("deportes.rdf", model);
...
```

Se crearán dos métodos para encapsular la obtención de recursos y propiedades de un determinado modelo. En el programa 29 se presentan el método `obtenerRecurso` que permite obtener una instancia de la clase `Resource` dada una URI y un modelo y el método `obtenerPropiedad` que permite obtener una instancia de la clase `Property` dada también una URI y un modelo.

Programa 29: Métodos para obtener recursos y propiedades de un modelo RDF en Jena

```
...
    public static Resource obtenerRecurso(String id, Model model) {
        String uri = model.expandPrefix("pucp:" + id);
        return model.getResource(uri);
    }

    public static Property obtenerPropiedad(String id, Model model) {
        String uri = model.expandPrefix("pucp:" + id);
        return model.getProperty(uri);
    }
...
```

En esta sección se realizarán las consulta para verificar a qué clases pertenece el recurso `Reilly` pertenece al conjunto `AllStarCandidate`. El programa 30 realiza dicha consulta. Verifique si se cumple.

Programa 30: Verificando una afirmación en RDF usando Jena

```
...
        String inputFileName = "deportes.rdf";
        Model model = ModelFactory.createDefaultModel();
        model.read(inputFileName);
        InfModel inf = ModelFactory.createRDFSModel(model);

        Resource Reilly = obtenerRecurso("Reilly", model);
        Resource AllStarCandidate = obtenerRecurso("AllStarCandidate", model);

        if (existenAfirmaciones(inf, Reilly, RDF.type, AllStarCandidate)) {
            System.out.println("La afirmación es cierta");
        } else {
            System.out.println("La afirmación no es cierta");
        }
...
```

Para poner en práctica

- Verifique si es verdad que se cumple la siguiente afirmación `Kaneda rdf:type AllStarCandidate`.
- Muestre las derivaciones y analice el resultado.

San Miguel, 30 de septiembre de 2020.