COSC 410

The Resource Description Framework

RDF is basically triples

- subject predicate object .
- Strangely reminiscent of an old AI language called SAIL, and binary relations
- and description logics: $(x,y) \in r$ is written
  x r y .
- Another viewpoint: Directed Graphs. Hence the word "node" for subject/object.

Naming

- Subjects, predicates, and objects can be IRIs [URL special-case-of URI .
  IRI internationalised-version-of URI.]
- Subjects and objects can be “blank nodes”.
- Objects can be literals (number or strings; RDFS lets you tag literals with language or data type but not both).

IRIs

- For the most part, IRIs are just strings in namespaces. .
- Some IRIs have semantics defined in public documents, notably rdf itself and foaf.
- IRIs can be things like ISBNs too...
- They are *rigid designators*, always standing for the same thing (whatever that is).

Blank nodes

- Blank nodes are like existentially quantified variables. _:foobar will refer to the same node throughout an RDF graph, but it won't have an absolute identity that can be referred to elsewhere. For example,

  _:m mother_of simpsons:bart .

  _:m hair_colour "blue".

Three special prefixes

- rdf: is used for RDF special terms
- rdfs: is used for RDF Scheme terms
- xsd: is used for XML Schema datatypes
- Example: `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` might be written rdf:type

Literals

- “value”^^type
- data type is aligned with XML Schemas
- xsd:string, boolean, decimal, integer, double, float, date, time, dateTime, date TimeStamp, gYear, gMonth, gDay, ..., byte, short, long, ..., base64Binary, language, token, xsd:Name, ...
- + rdf:HTML and rdf:XMLLiteral

Plain RDF is just triples

- Except for blank nodes, it's just binary relations between entities (individuals, resources) and binary relations between entities and values.
- Two sets of triples are equivalent iff there is a bijection between the blank nodes of one and the blank nodes of the other making the two sets equal. That's it.

RDF Schema is a DL

- c rdf:type rdf:class.   c is a concept.
- r rdf:type rdf:property. r is a rôle.
- x rdf:type c. $x \in c$.
- c rdfs:subClassOf: d. $c \sqsubseteq d$
- p rdfs:subPropertyOf: q. $p \sqsubseteq q$
- p rdfs:domain c. $\exists r.\top \sqsubseteq c$ (range similar)

Why rdf:/rdfs:?

- "The fact that the constructs have two different prefixes is a somewhat annoying historical artefact, which is preserved for backward compatibility."
- NB: schema.org has lots of webby concepts you should use instead of reinventing.

Writing RDF data

- There are many ways to write RDF.
- You can use XML. You can embed RDF in HTML. You can even use JSON.
- The simplest method is N-Triples.
- <subj> <pred> <obj> .   or
  <subj> <pred> "literal".
  IRIs are written between < ... > brackets.

Turtle

- IRIs may be relative. BASE <iri> says what they are relative to.
- PREFIX pfx: <iri> says that pfx:*name* is to be interpreted as <iri*name*>
- s p1 o1 ; p2 o2 ; p3 o3 . lets you avoid repeating a subject. o4, o5, o6 same pred.
- "a" stands for "rdf:type"

Example

- @base <http://example.org/> .
- @prefix foaf: <http://xmlns.com/foaf/0.1/> .
  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
  @prefix schema: <http://schema.org/> .
  @prefix dcterms: <http://purl.org/dc/terms/> .
  @prefix wd: <http://www.wikidata.org/entity/> .
- wd:Q12418
     dcterms:title "Mona Lisa" ;
     dcterms:creator <http://dbpedia.org/resource/
  Leonardo_da_Vinci> .

Example (2)

- ```
  <bob#me>
      a foaf:Person ;
      foaf:knows <alice#me> ;
      schema:birthDate "1990-07-04"^^xsd:date ;
      foaf:topic_interest wd:Q12418 .
  ```

- ```
  <http://data.europeana.eu/item/04802/243FA>
      dcterms:subject wd:Q12418 .
  ```

- ```
  [] foaf:topic_interest [
      dcterms:title "Mona Lisa" ;
      dcterms:creator <http://dbpedia.org/resource/
  Leonardo_da_Vinci> ] .
  ```
  ```

Triple stores

- A triple store accepts (s,p,o) and (s,p,v) triples. Lots of them, up to milliards.
- You can enumerate matches for partially specified triples, e.g., in SWI Prolog, rdf(wd:'Q12418', dcterms:title, Title)
- Issues: storage bulk, speed of loading, speed of retrieval, kinds of match allowed, ability to hold multiple graphs and query across them.

Inference

- With rdf:type, rdfs:domain, and so on, RDF is a description logic.
- We would like a query to succeed if it is *true*, whether it was explicitly stored or not.
- Some triple stores do this, *e.g.*, ClioPatria

Higher level triples

- It's not enough to find matches, present or implied, for partial patterns.
- We want to write queries above the level of the DL.

SPARQL

- Start with Turtle.
- Add logical variables ?*name*.
- Add case-insensitive keywords.
- Yearn for the respectability of SQL.
- Stir and bake.

Simple Query

- SELECT *vars* WHERE { *triples* }
- SELECT DISTINCT *vars* WHERE { *triples* }

Beware!

- Language tagging is essential in a world with 6,000 living languages
- But “barn” is an xsd:string and “barn”@en is an rdf:langString and the two are *not* equal! I can’t find any way to supply a default language tag in Turtle or SPARQL.
- Results can contain blank nodes.

Beware!

- Turtle uses @base and @prefix and lets you put them anywhere.
- SPARQL uses BASE and PREFIX (without a dot after the IRI) and only allows them at the beginning.
- Turtle picked up BASE and PREFIX from SPARQL, but Turtle is case sensitive.

expressions

- SELECT may use (*expression* AS *?var*)
- The body of WHERE may use BIND (*expression* AS *?var*)
- The body of WHERE may use FILTER *expression* — this can do comparisons and regular expression matching amongst other things

Returning a new graph

- CONSTRUCT { *triples* } WHERE { *triples* }
- blank nodes in the WHERE part are logical variables, new blank nodes in the CONSTRUCT part are really blank nodes.

OPTIONAL

- In relational algebra, r ⟕ s (the left outer join) joins tuples from r and s like r ⋈ s, but when a tuple in r has no match in s it is included anyway.
- { *pattern0* OPTIONAL *pattern1* ...} is like that. For a match of *pattern0,* information will be added from *pattern1* if possible; if not, *pattern0* won't fail.

UNION

- A simple tuple list is an AND.
- { *pattern0* UNION *pattern1* ...} is an OR.
- These can be nested in each other.

Negation

- Negation is done with FILTER, *e.g.*, FILTER (?x > ?y)
- FILTER NOT EXISTS { *pattern* }
- There is also FILTER EXISTS { *pattern* } where the nested pattern does not provide bindings for variables.
- { *pattern0* MINUS *pattern1* } is AND NOT.

Beware!

- Imitating SQL leads to a world of pain.
- Given :a :b :c,
  SELECT * WHERE { ?s ?p ?o
  FILTER NOT EXISTS {?x ?y ?z}} ⇨ nothing

  SELECT * WHERE { ?s ?p ?o
  MINUS {?x ?y ?z}} ⇨ [(:a,:b,:c)]

Compound rôles

- A rôle in SPARQL can be r, ^r (inverse), r1/r2 (composition), r1|r2 (or), r*, r+, r?, (r), and some other possibilities.
- :richard (:father|:mother)/:brother ?unc asks for my uncles.
- You can't use these in CONSTRUCT, only in WHERE.

More SQL-like stuff

- Aggregate expressions in SELECT: COUNT, SUM, MIN, MAX, AVG, SAMPLE
- Groups are defined using GROUP BY *vars*
- and filtered using HAVING (*expression*)
- You can sort with ORDER BY *vars*

SPARQL is not a logic

- SPARQL is a query language that sits on top of a description logic. While there is obviously some sort of subsumption relationship between some parts of queries, we don't expect any algorithm to find it. SPARQL queries make no assertion.
- This is how SPARQL escapes the complexity of inference trap.