Semantic Web Tutorial using N3

This is an introduction to semantic web ideas aimed at someone with experience in programming, perhaps with web sites and scripting, who wants to understand how RDF is useful in practice. The aim is to give a feel for what the Semantic Web is and how it will be useful in practice. This is illustrated using the N3 language, which is easy to read and write, and `cwm` which is an experimental general purpose program for semantic web stuff.

The tutorial is in the making: places linked below have text. This material will be presented as a tutorial at WWW2003 in Budapest, 2003-05. There may be notes about the presentation here, like who will present that section and how long it will take; they are just our working notes and should not be taken as any sort of committment.

Primer: Getting into RDF & Semantic Web using N3

The world of the semantic web, as based on RDF, is really simple at the base. This article shows you how to get started. It uses a simplified teaching language -- Notation 3 or N3 -- which is basically equivalent to RDF in its XML syntax, but easier to scribble when getting started.

Subject, verb and object

In RDF, information is simply a collection of statements, each with a subject, verb and object - and nothing else. In N3, you can write an RDF triple just like that, with a period:

```
<#pat> <#knows> <#jo> .
```

Everything, be it subject, verb, or object, is identified with a Universal Resource Identifier. This is something like <http://www.w3.org/> or <http://www.w3.org/2000/10/swap/test/s1.n3#includes>, but when everything is missed out before the "#" it identifies <#pat> in the current document whatever it is.

There is one exception that the object (only) can be a string representing a value:

```
<#pat> <#knows> <#jo> .
<#pat> <#age> "24" .
```

The verb "knows" is in RDF called a "property" and thought of as a noun expressing a relation between the two. In fact you can write

```
<#pat> <#child> <#al> .
```

alternatively, to make it more readable, as either

```
<#pat> has <#child> <#al> .
```

or

```
<#al> is <#child> of <#pat> .
```

There are two shortcuts for when you have several statements about the same subject: a semicolon ";" introduces another property of the same subject, and a comma introduces another object with the same predicate and subject.

```
<#pat> <#child>  <#al>, <#chaz>, <#mo> ;
       <#age>    "24" ;
       <#eyecolor> "blue" .
```

So, for example, the data in the table

	age	eyecolor
pat	24	blue
al	3	green
jo	5	green

could be written

```
  <#pat>   <#age> "24";  <#eyecolor> "blue" .
  <#al>    <#age>  "3";  <#eyecolor> "green" .
  <#jo>    <#age>  "5";  <#eyecolor> "green" .
```

Sometimes there are things involved in a statement don’t actually have any identifier you want to give them - you know one exists but you only want to give the properties . You represent this by square brackets with the properties inside.

```
<#pat> <#child> [ <#age> "4" ] , [ <#age> "3" ].
```

You could read this as #pat has a #child which has #age of "4" and a #child which has an #age of "3". There are two important things to remember

- The identifiers are just identifiers - the fact that the letters p a t are used doesn't tell anyone or any machine that we are talking about anyone whose name is "Pat" -- unless we say <#pat> <#name> "Pat". The same applies to the verbs - never take the actual letters c h i l d as telling you what it means - we will find out how to do that later.
- The square brackets declare that something exists with the given properties, but don't give you a way to refer to it elsewhere in this or another document.

If we actually want to use a name, we could have written the table above as

```
  [ <#name> "Pat"; <#age> "24";  <#eyecolor> "blue"  ].
  [ <#name> "Al" ; <#age>  "3";  <#eyecolor> "green" ].
  [ <#name> "Jo" ; <#age>  "5";  <#eyecolor> "green" ].
```

There are many ways of combining square brackets - but you can figure that out from the examples later on. There is not much left learn about using N3 to express data, so let us move on.

Sharing concepts

The semantic web can't define in one document what something means. That's something you can do in english (or occasionally in math) but when we really communicate using the concept "title", (such in a library of congress catalog card or a web page), we rely on a shared concept of "title". On the semantic web, we share quite precisely by using exactly the same URI for the concept of title.

I could try to give the title of an N3 document by

```
<> <#title>  "A simple example of N3".
```

(The <> being an empty URI reference always refers to the document it is written in.) The <#title> refers to the concept of #title as defined by the document itself. This won't mean much to the reader. However, a group of people created a list of properties called the Dublin Core, among which is their idea of title, which they gave the identifier

<http://purl.org/dc/elements/1.1/title>. So we can make a much better defined statement if we say

```
<> <http://purl.org/dc/elements/1.1/title>
 "Primer - Getting into the Semantic Web and RDF using N3".
```

That of course would be a bit verbose - imagine using such long identifiers for everything like #age and #eyecolor above. So N3 allows you to set up a shorthand prefix for the long part - the part we call the *namespace*. You set it up using "@prefix" like this:

```
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
<> dc:title  "Primer - Getting into the semantic web and RDF using N3".
```

Note that when you use a prefix, you use a colon instead of a hash between dc and title, and you don't use the <angle brackets> around the whole thing. This is much quicker. This is how you will see and write almost all your predicates in N3. Once set up, a prefix can be used for the rest of the file (* may change).

There are an increasingly large number of RDF vocabularies for you to refer to - check the RDF home page and things linked from it - and you can build your own for your own applications very simply.

From now, on we are going to use some well known namespaces, and so to save space, I will just assume the prefixes

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ont:  <http://www.daml.org/2001/03/daml-ont#> .
```

These are the RDF, RDF schema, and experimental DAML ontology namespaces, respectively. They give us the core terms which we can bootstrap ourselves into the semantic web. I am also going to assume that the empty prefix stands for the document we are writing, which we can say in N3 as

```
@prefix : <#> .
```

This means we could have the example above as

```
:pat :child [ :age "4" ] , [ :age "3" ].
```

which is slightly less characters to type. Now you understand how to write data in N3, you can start making up your own vocabularies, because they are just data themselves.

Making vocabularies

Things like dc:title above are RDF *Properties*. When you want to define a new vocabulary, or ontology, you define new classes of things and new properties. When you say what type of thing something is, you say a *Class* it belongs to.

The property which tells you what type something is is `rdf:type` which can be abbreviated to N3 to just `a`. So we can define a class of person

```
:Person a rdfs:Class.
```

In the same document, we could introduce an actual person

```
:Pat a :Person.
```

Classes just tell you about the thing which is in them. An object can be in many classes. There doesn't have to be any hierarchical relationship -- think of Person, AnimateObject, Animal, TallPerson, Friend, and so on. If there is a relationship between two classes you can state it - check out the properties (of classes) in RDF Schema and the the DAML ontology vocabularies.

```
:Woman a rdfs:Class; rdfs:subClassOf :Person .
```

A property is something which is used to declare a relationship between two things.

```
:sister a rdf:Property.
```

Sometimes when a relationship exists between two things, you immediately know something about them, which you can express as a class. When the subject of any property must be in a class, that class is a *domain* of the property. When the object must be in a class, that class is called the *range* of a property. A property can have many domains and ranges, but typically one specifies one.

```
:sister rdfs:domain :Person; rdfs:range :Woman.
```

Note the class identifiers start with capitals and properties with lower case letters. This is not a rule, but it is a good convention to stick to. Note also that because the domain of rdfs:range and rdfs:domain themselves is rdf:Property, it follows that :sister is a rdf:Property without it being stated explicitly.

Equivalence

Often, you define a vocabulary where one or more of the terms, whether or not you realized it when you started, is in fact exactly the same as one in another vocabulary. This is a really useful titbit of information for any machine or person dealing with the information! The property of equivalence between two terms is so useful and fundamental that N3 has a special shorthand for it, "=".

```
:Woman = foo:FemaleAdult .
:Title a rdf:Property; = dc:title .
```

Tip: Use other people's vocabularies when you can - it helps interchange of data. When you define your own vocabulary which includes synonyms, do record the equivalence because this, likewise, will help present and future processors process your and others' data in meaningful ways.

Choosing a namespace and publishing a schema

A document which defines properties and classes is called a schema. It defines the meaning of identifiers in a particular namespace. The namespace URI can be anything which you control: which you can guarantee no one will reuse for something else. One way is just to send an email about the namespace. Then look at the headers, and find the `Message-Id:` header which gives the unique identifier he mailer created for the mail message, and use that. Another way is to put a document in a bit of very well maintained web space which no one can take away from you. Or of course if you are just playing with it, use a file (say `mydb.n3`) in the same directory as the rest of your work, and the namespace identifier you can use is just `<mydb.n3#>`.

When use a web document to contain information in N3 about the terms you define (good practice!) then the namespace will end in a hash.

You don’t have to make your schema available to the world, but it helps machine process documents written with your vocabulary if you do. It especially helps if the email message or web page is written in RDF (in XML or in N3) and contains the schema information, as a program can associate the two easily, if it finds the mail or looks up the web page. If you do this, the namespace will be the document URI followed by a `#.`

This is a big topic - see for example further reading

- W3C Namespace policy
- W3C draft persistence policy
- Cool URIs don’t change

Now you know all you need to start creating your own vocabularies, or ontologies, and you have pointers to where to look for the richer ways of defining them.You don’t have to go any further, as what you have now will allow you to create new applications, and create schemas, data files, and programs which interchange and manipulate data for the semantic web.

More

At this point, you should be getting the hang of it and be writing stuff. To give you some more ideas, though, there is a longer list of more complex and varied examples. These come with less tutorial explanation.

Or, you can continue with a tutorial which goes into mroe features of the language, explaining how to process you data and involve other data on the Web. In that case, next bit is about: Shortcuts and long cuts

---

References

- Many More Examples
- Notation3 - Design Issues article

Tim BL, with his director hat off

$Id: Primer.html,v 1.41 2003/03/19 21:42:28 timbl Exp $

Comparing Formats

There are a many languages in use today for exchanging RDF-structured information. Here’s a simple example rendered in many different ways. Converters to between some of these formats, but not others (yet).

@@@ TODO: Make give him another name, ’Patrick Smith’ to show off cardinality issues.

@@@ change names to about-pat and friends-ontology (clear, but maybe not practical)....

@@@ make a demo 2 about either auto-repair shops or late-night restaurants. Use it for n-ary via Go-Meta (reify).

English (Very Informal)

There is person, Pat, known as "Pat Smith" and "Patrick Smith". Pat has a pet dog named "Rover".

English Hypertext (Informal)

http://www.w3.org/2000/10/swap/test/demo1/

Here the ambiguity of terms is addressed by making the words be hypertext links. The links may or may not work, depending on the servers involved.

Pat is a human with the names "Pat Smith" and "Patrick Smith". Pat has a pet, a dog, with the name "Rover".

N3

```
@prefix : <http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#> .
@prefix bio: <http://www.w3.org/2000/10/swap/test/demo1/biology#> .
@prefix per: <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#> .

:pat     a bio:Human;
     per:name "Pat Smith",
              "Patrick Smith";
     per:pet  [
         a bio:Dog;
         per:name "Rover" ] .
```

Directed Labeled Graph

[IMAGE]

RDF/XML

```
<rdf:RDF xmlns="http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#"
    xmlns:bio="http://www.w3.org/2000/10/swap/test/demo1/biology#"
    xmlns:per="http://www.w3.org/2000/10/swap/test/demo1/personal-notes#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <bio:Human rdf:about="#pat">
        <per:name>Pat Smith</per:name>
        <per:name>Patrick Smith</per:name>
        <per:pet>
            <bio:Dog>
                <per:name>Rover</per:name>
            </bio:Dog>
        </per:pet>
    </bio:Human>
</rdf:RDF>
```

N-Triples

With @prefix:

```
@prefix : <http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#> .
@prefix bio: <http://www.w3.org/2000/10/swap/test/demo1/biology#> .
@prefix per: <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

:pat rdf:type bio:Human.
:pat per:name "Pat Smith".
:pat per:name "Patrick Smith".
:pat per:pat _:genid1.
_:genid1 rdf:type bio:Dog.
_:genid1 per:name "Rover".
```

In standard form:

```
<http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2000/10/swap/test/demo1/biology#Human> .
<http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat> <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name> "Pat Smith" .
<http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat> <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name> "Patrick Smith" .
<http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat> <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet> _:genid1 .
_:genid1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2000/10/swap/test/demo1/biology#Dog> .
_:genid1 <http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name> "Rover" .
```

Prolog

Without namespaces, this is very terse:

```
human(pat).
dog(rover).            % we have to assign a name
name(pat, "Pat Smith").
name(pat, "Patrick Smith").
name(rover, "Rover").
pet(pat, rover).
```

One approach to namespaces:

```
ns(ns1_, "http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog").
ns(bio_, "http://www.w3.org/2000/10/swap/test/demo1/biology#").
ns(per_, "http://www.w3.org/2000/10/swap/test/demo1/personal-notes#").
bio_Human(ns1_pat).
bio_Dog(rover).      # unprefix could be be NodeIDs...
per_name(ns1_pat, "Pat Smith").
per_name(ns1_pat, "Patrick Smith").
per_name(rover, "Rover").
per_pet(ns1_pat, rover).
```

SQL

Table Relating URIs to Internal IDs

URIs are big, variable-length keys; it’s much more efficient to translate them into an internal id. If the "uri" is NULL, this resource is anonymous (a bNode, like a NodeID in RDF/XML).

```
CREATE TABLE uri (
  id INT AUTO_INCREMENT PRIMARY KEY,    # PRIMARY = UNIQUE and NOT NULL
  uri BLOB,   # BLOB is also called LONGVARBINARY
  UNIQUE KEY uri (uri(64))  # length is just a tuning knob
);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/1999/02/22-rdf-syntax-ns#type’);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/biology#Human’);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/biology#Dog’);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name’);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet’);
INSERT INTO uri (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat’);
INSERT INTO uri (uri) VALUES (NULL);   # this is rover, who has no URI

mysql> select * from uri;
+----+--------------------------------------------------+
| id | uri                                              |
+----+--------------------------------------------------+
|  1 | http://www.w3.org/1999/02/22-rdf-syntax-ns#type  |
|  2 | http://www.w3.org/2000/10/swap/test/demo1/biology#Human |
|  3 | http://www.w3.org/2000/10/swap/test/demo1/biology#Dog   |
|  4 | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name        |
|  5 | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet         |
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat                                     |
|  7 | NULL                                             |
+----+--------------------------------------------------+
```

Option 1: One Table, One Column Per Predicate

This is a simple, intuitive approach, but...

- You need to add a new column with each new predicate you use
- Predicates must be known to be individual-valued or data-valued; and if data-valued, then what type?
- You cannot have multivalued (cardinality > 1) properties

```
CREATE TABLE resource (
  id INT PRIMARY KEY,
  type INT,           # http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  name varchar(255),  # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name
  pet INT             # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet
);
INSERT INTO resource (id, type, name, pet) VALUES (6, 2, ’Pat Smith’, 7);
INSERT INTO resource (id, type, name) VALUES (7, 3, ’Rover’);

mysql> select * from resource;
+----+------+-----------+------+
| id | type | name      | pet  |
+----+------+-----------+------+
|  6 |    2 | Pat Smith |    7 |
|  7 |    3 | Rover     | NULL |
+----+------+-----------+------+
```

Option 2: One Table Per Class, One Column Per Predicate

Here we avoid having such wide tables, and our modeling better matches the normal database modeling. On the other hand, we have some conceptual redundancy between human.name and dog.name, because there is no support for inheritance.

```
CREATE TABLE human (  # http://www.w3.org/2000/10/swap/test/demo1/biology#Human
  id INT PRIMARY KEY,
  name varchar(255),  # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name
  pet INT             # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet
);
CREATE TABLE dog (  # http://www.w3.org/2000/10/swap/test/demo1/biology#Dog
  id INT PRIMARY KEY,
  name varchar(255)   # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name
);
INSERT INTO human VALUES (6, ’Pat Smith’, 7);
INSERT INTO dog   VALUES (7, ’Rover’);

mysql> select * from human, dog where human.pet=dog.id;
+----+-----------+------+----+-------+
| id | name      | pet  | id | name  |
+----+-----------+------+----+-------+
|  6 | Pat Smith |    7 |  7 | Rover |
+----+-----------+------+----+-------+
```

Option 3: One Table Per Predicate

Here we can support duplicate values.

```
CREATE TABLE name (  # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name
  subject INT NOT NULL,
  object varchar(255),
  INDEX(subject),
  UNIQUE INDEX(subject, object)
);
CREATE TABLE pet (   # http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet
  subject INT NOT NULL,
  object INT,
  INDEX(subject),
  UNIQUE INDEX(subject, object)
);
CREATE TABLE type (   # http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  subject INT NOT NULL,
  object INT,
  INDEX(subject),
  UNIQUE INDEX(subject, object)
);
INSERT INTO name VALUES (6, ’Pat Smith’);
INSERT INTO name VALUES (6, ’Patrick Smith’);
INSERT INTO name VALUES (7, ’Rover’);
INSERT INTO pet VALUES (6, 7);
INSERT INTO type VALUES (6, 2);
INSERT INTO type VALUES (7, 3);

mysql> select * from name;
+---------+---------------+
| subject | object        |
+---------+---------------+
|       6 | Pat Smith     |
|       6 | Patrick Smith |
|       7 | Rover         |
+---------+---------------+

mysql> select * from pet;
+---------+--------+
| subject | object |
+---------+--------+
|       6 |      7 |
+---------+--------+

mysql> select * from type;
+---------+--------+
| subject | object |
+---------+--------+
|       6 |      2 |
|       7 |      3 |
+---------+--------+
```

Option 4: One Table of Triples, One Tal

```
If we redo our URIs table as a "resources" table, with literals as


well as URIs, we have some more options.



CREATE TABLE resources (
  id INT AUTO_INCREMENT PRIMARY KEY,    # PRIMARY = UNIQUE and NOT NULL
  # either provide a uri
  uri BLOB,
  # or a literal_value, which might have a datatype and language
  literal_value BLOB,
  datatype INT,
  language VARCHAR(5),
  UNIQUE KEY (uri(64))  # length is just a tuning knob
);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/1999/02/22-rdf-syntax-ns#type’);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/biology#Human’);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/biology#Dog’);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name’);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet’);
INSERT INTO resources (uri) VALUES (’http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat’);
INSERT INTO resources (uri) VALUES (NULL);   # this is rover, who has no URI
INSERT INTO resources (literal_value) VALUES (’Pat Smith’);
INSERT INTO resources (literal_value) VALUES (’Patrick Smith’);
INSERT INTO resources (literal_value) VALUES (’Rover’);

mysql> select * from resources;
+----+--------------------------------------------------+---------------+----------+----------+
| id | uri                                              | literal_value | datatype | language |
+----+--------------------------------------------------+---------------+----------+----------+
|  1 | http://www.w3.org/1999/02/22-rdf-syntax-ns#type  | NULL          |     NULL | NULL     |
|  2 | http://www.w3.org/2000/10/swap/test/demo1/biology#Human | NULL          |     NULL | NULL     |
|  3 | http://www.w3.org/2000/10/swap/test/demo1/biology#Dog   | NULL          |     NULL | NULL     |
|  4 | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name        | NULL          |     NULL | NULL     |
|  5 | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet         | NULL          |     NULL | NULL     |
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat                                     | NULL          |     NULL | NULL     |
|  7 | NULL                                             | NULL          |     NULL | NULL     |
|  8 | NULL                                             | Pat Smith     |     NULL | NULL     |
|  9 | NULL                                             | Patrick Smith |     NULL | NULL     |
| 10 | NULL                                             | Rover         |     NULL | NULL     |
+----+--------------------------------------------------+---------------+----------+----------+
```

Then we can have a simple table of triples:

```
CREATE TABLE triples (
  subject INT NOT NULL,
  predicate INT NOT NULL,
  object INT NOT NULL,
  UNIQUE INDEX(subject, predicate, object),
  INDEX(predicate, object),
  INDEX(object, predicate)
);
INSERT INTO triples VALUES (6, 4, 8);
INSERT INTO triples VALUES (6, 4, 9);
INSERT INTO triples VALUES (7, 4, 10);
INSERT INTO triples VALUES (6, 1, 2);
INSERT INTO triples VALUES (7, 1, 3);
INSERT INTO triples VALUES (6, 5, 7);

mysql> select * from triples;
+---------+-----------+--------+
| subject | predicate | object |
+---------+-----------+--------+
|       6 |         1 |      2 |
|       6 |         4 |      8 |
|       6 |         4 |      9 |
|       6 |         5 |      7 |
|       7 |         1 |      3 |
|       7 |         4 |     10 |
+---------+-----------+--------+

mysql> select s.id, s.uri, p.uri as "predicate",
       o.id, o.uri, o.literal_value as "lit"
       from triples, resources as s, resources as p, resources as o
       where s.id=triples.subject AND
             p.id=triples.predicate AND
             o.id=triples.object;
+----+------+-----------------------------------------------------+----+-----------------------------------------------------+---------------+
| id | uri  | predicate                                           | id | uri                                                 | lit           |
+----+------+-----------------------------------------------------+----+-----------------------------------------------------+---------------+
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat | http://www.w3.org/1999/02/22-rdf-syntax-ns#type |  2 | http://www.w3.org/2000/10/swap/test/demo1/biology#Human | NULL          |
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name       |  8 | NULL                                                | Pat Smith     |
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name       |  9 | NULL                                                | Patrick Smith |
|  6 | http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#pet        |  7 | NULL                                                | NULL          |
|  7 | NULL | http://www.w3.org/1999/02/22-rdf-syntax-ns#type |  3 | http://www.w3.org/2000/10/swap/test/demo1/biology#Dog   | NULL          |
|  7 | NULL | http://www.w3.org/2000/10/swap/test/demo1/personal-notes#name       | 10 | NULL                                                | Rover         |
+----+------+-----------------------------------------------------+----+-----------------------------------------------------+---------------+
```

XML (but not RDF/XML)

The **striped** or **alternating-normal form** approach uses a markup language designed for the application domain. Conversion to and from triples requires specialized software

```
<Human>
  <uri>http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#pat</uri>
  <name>Pat Smith</name>
  <pet>
    <Dog>
     <name>Rover</name>
    </Dog>
  </pet>
</Human>
```

An alternative is to use **XML Triples**. This does not involve domain-specific markup. Several candidate DTDs/Schemes have been proposed; this is just one strawman. For some applications, this kind of syntax may be easier than RDF/XML or a striped syntax.

```
<!DOCTYPE Graph [
 <!ENTITY rdf    "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <!ENTITY bio    "http://www.w3.org/2000/10/swap/test/demo1/biology#">
 <!ENTITY ns1    "http://www.w3.org/2000/10/swap/test/demo1/pat-and-dog#">
 <!ENTITY per    "http://www.w3.org/2000/10/swap/test/demo1/personal-notes#">
]>
<Graph>
  <Triple>
    <subject><uri>&ns1;pat</uri></subject>
    <predicate><uri>&rdf;type</uri></predicate>
    <object><uri>&bio;Human</uri></object>
  </Triple>
  <Triple>
    <subject><uri>&ns1;pat</uri></subject>
    <predicate><uri>&per;name</uri></predicate>
    <object><literal>Pat Smith</literal></object>
  </Triple>
  <Triple>
    <subject><uri>&ns1;pat</uri></subject>
    <predicate><uri>&per;pet</uri></predicate>
    <object><nodeID>genid1</nodeID></object>
  </Triple>
  <Triple>
    <subject><nodeID>genid1</nodeID></subject>
    <predicate><uri>&rdf;type</uri></predicate>
    <object><uri>&bio;Dog</uri></object>
  </Triple>
  <Triple>
    <subject><nodeID>genid1</nodeID></subject>
    <predicate><uri>&per;name</uri></predicate>
    <object><literal>Rover</literal></object>
  </Triple>
</Graph>
```

Javascript and Python

The RDF model maps fairly well to several common programming constructs, especially those found in interpreted object-oriented languages, like Javascript and Python. Still, this mapping is not perfect.

This simple approach ignores URI and cardinality issues; it only allows pat to have one name and one pet:

```
pat = Human()
rover = Dog()
pat.name = "Pat Smith"
rover.name = "Rover"
pat.pet = rover
```

We can begin to consider cardinality:

```
pat = Human()
rover = Dog()
pat.name.append("Pat Smith")
rover.name.append("Rover")
pat.pet.append(rover)
```

But to be more complete, we need something like this which loses the simplicity of the built-in model:

```
pat = Resource()
rover = Resource()
pat.addProperty(  ns.rdf.type, ns.bio.Human)
rover.addProperty(ns.rdf.type, ns.bio.Dog)
pat.addProperty(  ns.per.name, "Pat Smith")
rover.addProperty(ns.per.name, "Rover")
pat.addProperty(  ns.perpet, rover)
```

$Id: formats.html,v 1.8 2003/04/11 05:03:40 sandro Exp $

Tutorial: Paths and lists

You don’t need to know this. It just makes life easier, and of course helps when reading other people’s work.

Paths

Often it turns out that you need to refer to something indirectly through a string of properties, such as "George’s mother’s assistant’s home’s address’ zipcode". This is traversal of the graph. In the N3 we have dealt with up till now, this would look something like

```
[is con:zipcode of [
    is con:address of [
        is con:home of [
            is off:assistant of [
                is rel:mother of :George]]]]]
```

which reads "that which is the zipcode of that which is the address of that which is the home of that which is the assistant of that which is the mother of :George", which isn’t very conventient to read or write. And this is when in ant object-oriented langauge, you would just cascade methods or attributes on properties using ".".

To make it easier in N3 there is a shortcut for the above would just be written

```
:George.rel:mother.off:assistant.con:home.con:address.con:zipcode
```

The dot must be immediately followed by the next thing, with no whitespace.

This is forward traversal of the graph, where with each "." you move from something to its property. So ?x.con:mailbox is x’s mailbox, and in fact in english you can read the "." as " ’s".

You can do backward traversal, using "^" instead of "." as punctuation. So if :me.con:mailbox means my mailbox, then <mailto:ora@lassila.com>^con:mailbox is that which has <mailto:ora@lassila.com> as a mailbox. This backward traversal is less usual - you can’t do it object-oriented programming langauges -- but sometimes its what you need.

Note that there isn’t necessraily only one match for a path - often there are many, when it is used on the left side of a rule.

```
:me.rel:parent
```

would be "Some x such that I have parent x", and

```
:me.rel:parent^rel:parent
```

would mean "Some y such that for some x, I had parent x and y had parent x", or loosely, some sibling of mine. In this imaginary ontology, ".rel:child" is equivalent to "^rel:parent".

Whatever the sequence of "." and "^", they always are read left to right across the page.

These are known as "paths". If you are used to XML, think: Xpath but simpler. If you think of the circles and arrows graph of data, think of a path from node to node.

Cwm doesn’t currently use paths on output, so getting cwm to input and output a file will turn it into the form of N3 you already know.

Lists

A common need is to represent an ordered collection of things. This is done in OWL as a owl:Collection. In N3, the list is represented by sepetating the objects with whitespace and surrounding them with parenthases. Examples are:

```
( "Monday" "Tuesday" "Wednesday" )

(:x :y)

( :cust.bus:order.bus:number  :cust.bus:order.bus:referncedPurchaseOrder.bus:number )
```

These lists are actually shorthand for statements which knit blank nodes together using owl:first and owl:rest. owl:first is the relationship betwen a list and its foirst component. owl:rest if the relationship between a list and the list of everything except its first element. owl:nil is the name for the empty list. Therefore,

```
( "Monday" "Tuesday" "Wednesday" )
```

is equivalent to

```
[ owl:first "Monday";
  owl:next [ owl:first "Tuesday";
             owl:rest [ owl:first "Wednesday";
                        owl:rest owl:nil ]]]
```

One of the common uses of lists is as parameter to a relationship which ihas more than one argment.

```
(  "Dear " ?name " "," )   string:conatenation   ?salutation.
```

for example, indicates that the salutation is the string concatenation of the three strings "Dear", whatever ?name is, and a comma. This will be used with built-in functions which we will discuss later.

Next: Rules [p 22]

Tim BL, with his director hat off

$Id: Shortcuts.html,v 1.4 2003/02/13 23:47:37 timbl Exp $

Ontologies

An *ontology* is a precise specification of a vocabulary. It typically consists of the set of vocabulary terms along with human-readable definitions of each term and machine-readable declarations about how the terms relate to each other. An ontology embodies a conceptualization of some subject area; to understand someone, you need to understand both their grammar and all the ontologies they are using.

The process of developing an ontology is usually similar to the process of object-oriented analysis and design: you figure out the classes of objects in some subject area, then consider how instances of those classes relate to each other and what they have in common.

With RDF ontologies, however, this analysis usually remains separated from any programming language. The features of inheritence and type-checking remain, however, and can be of great value.

With RDF, we currently have four ontology languages. In order of increasing expressive power:

- RDFS is quite simple, but may be all you need.
- OWL Lite is very strict and precise, but not very expressive. For many modeling tasks, it’s a good place to start.
- OWL DL is strict like OWL DL, but offers all the standard Description Logic (DL) features.
- OWL Full removes many restrictions of OWL DL, making reasoning potentially intractable. (This replaces DAML+OIL.)

Classes, Properties, Individuals, and Datatype Values

In ontology development, it helps to divide the world into four different kinds of things:

- An **individual** is something in the domain of discourse, like a specific person, an inventory item, or a city. This roughly corresponds to "objects" in object-oriented programming.
  - Arnold Schwarzenegger
  - Budapest
  - The Sun
  - The French Revolution
  - http://www.w3.org/ (the W3C front page)
- A **data value** is something which can be fully represented by a sequence of characters. These correspond to values of primitive types in many programming languages. There is some flexibility about whether to model some things as individuals or a data values, especially when you have the option of defining a new datatype.
  - the integer 17
  - the string "Hello, World"
  - the floating point number 3.771
  - the date January 17, 1985
- A **class** is a grouping of individuals to reflect something they have in common. Classes may overlap arbitrarily. In some ontology languages classes may contain data values and other classes (or even themselves!), but it usually makes sense to restrict ourselves to classes of individuals as required by OWL DL and OWL Lite.
  - Human
  - Valid XHTML Document
  - Member of the United Nations
  - ActiveCustomer

- A **property** relates an individual to something else, the value of that property for that individual. In the more restrictive languages, we have "data-valued properties" which have values which are datatype values, and "individual-valued properties" which have values which are individuals.
    - date_of_birth
    - lastName
    - mayor
    - purchaseMade

Subclass Reasoning

```
Man subclassOf Human
YoungMan subclassOf Man
```

OWL Inference

```
Joe is a YoungMan -->  Joe is a Man, Joe is a Human
```

```
Joe is a YoungMMMan -->  [okay]
```

"Lint" Processing

```
Joe is a YoungMan -->  [okay]
```

```
Joe is a YoungMMMan -->  Warning: No Such Class
```

Domain, Range, Cardinality

Sameness

Difference

Hrm. testing.

$Id: ontologies.html,v 1.3 2003/03/28 17:15:18 sandro Exp $

Primer: Rules and Formulae

We are going to learn how to express rules in N3, which will allow us to do all kinds of things. A simple rule might say something like (in a mechanical vocabulry, say) "If one this includes another, then the second is a part of the first", or

```
{ :thermostat :temp :high } log:implies { :heating :power "0" } .
```

The curly brackets here enclose a set of statements. Until this point, all statements in each file have been effectively in one bucket. They are all generated when the file is parsed, and stored in a store on an equal footing. In a rule, however, we need to talk about hypothetical statements: *If* this *were* the case *then* that *would* be the case. The statements within the brackets are not asserted like the other ones in the file. In the case above, the file does not say that the thermostat temperature is high. It just says that the left-hand formula implies the right-hand formula.

You see that, apart from the fact that the subject and object of the statement are formulae, the line above is just a single statement. `log:implies` is a special predicate which links formulae. The statement is a fact. Rules are in general facts, and on the semantic web they may or not be used by various programs at various times to figure things out.

The curly brackets are important. They do take us out of the things we can represent using the current RDF/XML OWL specifications. ((Formulae are (2003/2) a longer-term issue for the RDF Core group). Cwm will in most cases serialize formulae using an XML sysntax which is not standard.

Variables

In fact, formulae in N3 are more than just sets of statements. They also allow declarations of variables. A variable is like just a symbol (such as `:x` or `var:Y`) except it is used to stand for other symbols. There are two types. A "`forAll`" or *universal* variable, declared like this,

```
this log:forAll :x.
{ :thermostat :temp :x } log:implies { :cooling :power :x } .
```

means that all this information is true even if the symbol ":x" is replaced consistently by something else. This is of course what you want for a rule which implies something generally about anything. The `"this"` indicates the scope of the variable - it in fact stands for the formula in which it is, in this case the outermost formula of the document. In most cases for simple rules, the variables are quantified in the scope of the document, but when things get nested, it is wise to think about what you mean. We don't deal with it in detail here. Mathematically, the

&ForAll; x (temp (thermostat , x) &DoubleRightArrow; power (cooling , x))

A formula can also have "`forSome`", or *existential*, variables. They are declared as in

```
this log:forSome :a.
:Joe :home :a.
:a  :phone "555-1212".
```

This means that there is something which is Joe's home and has the given phone number. We've been doing this all along, though, surely, with blank nodes. Indeed - most times that an existential variable is used it is actually implicit in the [bracket] syntax.

```
:Joe :home [ :phone "555-1212" ].
```

Each pair of square brackets actully stands for an unnamed existential variable. Cwm will generally use square brackets on output when it can - if there are loops then one has to use explict variables.

```
this log:forSome :a.
:Joe :home :a.
:a   :owner :Joe.
:a  :phone "555-1212".
```

Variables with ? and _:

So far we have introduce variables for rules using *log:forAll*. We have introduced existential variables (blank nodes) with `log:forSome`. These are both special pseudoProperies which are really special langauge things. There are shortcuts under certain circumstances, which avoid you having to type them.

When a variable is universally quantified in not the formula itself but the surrounding formula, then you can just use ?x. This is in fact what you want for rules whcih don't have nested formulae inside the antecentent.

So instead of

```
this log:forAll :x, :y.

{ :x :parent :y } =>  { :y :child :x }.
```

you can write

```
{ ?x parent ?y } =>  { ?y :child ?x }.
```

which means just the same. We will use this all the time when writing simple rules. If you need the same variable to appear at several levels, then you can't use this form.

The other shortcut is that if you have existentially qualified variables in the scope of the whole document, then you can use _:a and so on. (This is the same as the ntriples notation)

So instead of writing

```
this log:forSome :a, :b.
```

```
:a :hates :b.   :b :loves :a.
```

you could just write

```
_:a :hates _:b.   _:b :loves _:a.
```

In practice this is very ofetn used in ntriples but in N3, you can normally use the [] syntax to avoid usiing any variables at all.

Shorthand symbols for implies, etc.

When it is used as a predciate (verb) in N3, `log:implies` and => are the same; `owl:equivalentTo` and = are the same.

So, having gone into a few details of universal and existential identifiers, what we are left with is a pretty simple rule language, which is still a fairly straightforward extension of RDF.

```
{?x family:parent ?y. ?y family:brother ?z} => {?x family:uncle ?z}.
```

(Always remember the dot after the rule.)

Now let’s learn how to process them [p 26] .

References

- Many More Examples
- Notation3 - Design Issues article
- Other rule languages for comparison eg Prolog, RuleML , Jess

---

Thanks to contributors including Joseph Reagle who added the uncle example to the original primer.

Tim BL, with his director hat off

$Id: Rules.html,v 1.6 2003/03/19 22:12:08 timbl Exp $

Primer: Processing your data using N3 and Cwm

Now that we know how to make a statements in N3, what can we do with them?

You know about how to write your data in N3, and also how to describe the terms you are using in a schema, or ontology. Why is that so useful? Because of all the things you can then do with it. Here we’ll learn some basic ways to use cwm, a command line python program. There are lots of other processors for RDF information, and cwm is just one we’ll use here. It was designed to show the feasibility of everything in the Semantic Web layer cake, so we can go quite a long way with it. It isn’t optimized, though, so you may find it too slow to use for large datasets. This is going to be a completely practical how-to, rather than a theoretical analysis of what is going on. You might like to keep around for reference:

- The cwm manual

  Examples - rather random order supplement

Cwm is a python program, so on most systems you can run it as `python wherever/cwm.py`, depending on where you have installed it. From now on, though, we’ll assume you have set up an alias or whatever your system uses to make it available by simply by the command `cwm`. You can always use the long form if you don’t have the short form.

Cwm uses the command line as a sequence of operations to perform from left to right. You can input data, process it and output it. The default is to input and output from the standard input and output. So you can read

```
cwm --rdf foo.rdf --think --n3
```

as "switch to RDF/XML format, read in foo.rdf, think about it, and then switch to N3 format (for output)".

Any filename is actually a relative URI, so you can suck data off the web just by giving its URI:

```
cwm http://www.w3.org/2000/10/swap/log.n3
```

will read in the remote file, and then output it to the terminal.

Converting data format

Converting data formats is simple:

```
cwm --rdf foo.xml --n3 > foo.n3
```

converts the RDF/XML file foo.xml into n3, and

```
cwm bar.n3 --ntriples > bar.nt
```

converts bar.n3 to ntriples format. The default format at the beginning of the command line is N3. We’ll mostly use RDF/N3 from now on, but all the data could equally well be in RDF/XML.

(All these examples involve cwm reading the data into a store, and then reading it out. This means that the order of the results will be different (sorted) and the comments will be lost. There is actually a`--pipe` option which preserves comments and order, but it only works with flat RDF files, not with rules and other things which need formulae. **Hint**: if the comment you are about to write is about one of the things your RDF file is about, make it a rdfs:comment property and it will be carried through the system -- who knows who will find it useful later?)

Merging data

The great thing about RDF which you will soon just assume (but you can’t do with plain XML documents) is that merging data is trivial.

```
cwm --rdf foo.rdf --n3 bar.n3 > both.n3
```

reads one xml file and one n3 file, and then outputs the results in N3 to both.n3.

- see also - color example@@

Deducing more data

Often, you have data in a raw form and the information you want can be deduced from it, and you would like it added to the data set.

In uncle.n3 we state that Fred is the father Joe, and Bob is the brother of Fred; we also describe the logical rule for the uncle relationship:

```
@prefix : <uncle#>.
:Joe has :father :Fred.
:Fred has :brother :Bob.

@prefix log: <http://www.w3.org/2000/10/swap/log#> .
this log:forAll :who1, :who2.
{ :who1 :father [ :brother :who2 ] } log:implies { :who1 :uncle :who2 }.
```

This rule means "whenever someone’s father has a brother, then the latter is their uncle". This rule, once is cwm’s store, will cause *cwm* to deduce the uncle information when it runs with the command line option `--think.`

@@- adding effective focal length to a photograph

Delivery of packet in the USA has often one price within the *contiguous United States*. How could we find those? Well, the test data set USRegionState.n3 has a list of states of the USA. This includes the `borderstate` property which gives a state’s neighbors. Contiguous means that you can get there from here by going from state to neighboring US state. Suppose you are starting in Boston.so you can get to Massachusetts. Also, any state bordering a contiguous state must also be contiguous. That’s all the rules you need:

```
{?x us:code "MA"} => { ?x a :ContiguousState }.
{?x a :ContiguousState.  ?x us:borderstate ?y} => {?y a :ContiguousState}.
```

These rules need to be applied more than once to find all the contiguous states. This is where you can see the difference between --rules and --think. Try it with --rules. Try it with --rules --rules. Then try it with --think.s

Filtering: when you have too much data

The result of the above search for contiguous state is too much information. How can we cut it down? Sometimes all we want from the mass of data at our disposal is a single statement.

One way is to decorate the data by marking all the uninteresting bits as being in class log:Chaff. Then, the --purge option of cwm will remove from the store any statement which mentions anything which is in that class. This doesn’t need much more discussion.

A more interesting way is to compute just the things which are interesting. This is done with a *filter*.

In uncleF.n3 we use the uncle example above, but as a filter. When a filter runs (unlike --think) *only* the information gathered by the rules is preserved: everything else is discarded. We use a filter to select the logical relationships that we want from the mass of what is already known:

```
@prefix : <uncle#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#> .

this log:forAll :p.

# What is the relationship between Joe and Bob

{ :Joe :p :Bob } log:implies { :p a :RelationshipBetweeJoeAndBob }.

# Is Bob an Uncle of Joe?

{ :Joe :uncle :Bob } log:implies { :Joe :uncle :Bob }.
```

When we ask cwm to consider the implication it concludes:

```
> python cwm.py uncle.n3 --think --filter=uncleF.n3
    :Joe     :uncle :Bob .
    :uncle     a :RelationshipBetweeJoeAndBob .
```

You can read the command line as: *read uncle.n3 and the deduce any new information you can given any rules you have. Now just tell me the information selected by the filter uncleF.n3*.

Note that any data in the filter is **not** used. It is easy to imagine that the machine knows something because you can see it in the filter file. However, the filter file is only searched for rules. If you want to include the data, you can put it into a separate input file, or you can even add the filter file as an input file as well as using it as the filter.

Combining cwm steps

In a lot of cases, one wants to take input, decorate the information with new stuff inferred using rules (with `--think`) and then filter out the essence of what is needed. Commands like

```
cwm income.n3 expenses.n3  categorize.n3 --think --filter=taxable.n3
```

are common ways of using cwm.

Report Generation

All the examples above proves data and leave the result in N3. It is easy of course to generate RDF/XML, too.

What do you do to generate something else, maybe an XHTML page, or an SVG diagram?

Using RDF/XML and XSLT

If you are used to XML tools such as XSLT, then you can generate the RDF/XML of your data, and then use XSLT to transform it into a report. When you do this, you may want to use the rdf output control flags to tell cwm how you like your output. There is also the `--bySubject` output method which prevents the "pretty-printing" of XML.

- See examples: Making .dot files, ...

Using --strings to output text

Another way is to output strings from cwm. This may work well or seem a bit weird, depending on your application. Remembering that all the data in cwm is stored in a big unordered pot, the trick is to tell cwm where you want things on the output stream. This is done by giving each output string a key using log:outputString relationship from key to string.. The --strings option then outputs all the strings which have keys, in order of key. You can use string:concatenation to build the strings out of data. You can also use it to build the key.

```
{   ?x.context:familyName  ?k.

    (?x.context:givenName " " ?x.context:familyName "has been invited\n" )
string:concatenation  ?s

} => {

    ?k  log:outputString ?s.

}.
```

This says if k is someone's family name, and a string is made by concatenating their given name, a space, their family name and " has been invited\n", then that string is output in order of key. In other words, in order of family name.

- See examples: tax report

Debugging

This is all very well, but what happens if it doesn't work? There are a number of ways of looking at problems. They are not in order.

Checking the syntax of files when you have finished editing them can save both later. You can just load the file with the `--no` option so that the syntax is checked by no output is done.

If a rule isn't firing, try commenting out one of the conditions to see which isn't firing.

Try calculating intermediate results, dividing a big rule into more than one step.

You may have misspelled a term. If you do that, it just won't match but it will be perfectly good syntax. To catch this, validate your file using the DAML validator or the cwm validator. These will check that the terms you use are indeed declared in a schema. It'll check a few other things too.

If you think it would help to know what cwm is doing, you can run cwm with `--chatty=50`, or any value between 0 and 99. It is often useful to change the `--chatty` flag at various times in the command line, setting it back to zero when you don't need it. You will find 15 tells you when files are being opened, 25 gives a list of the things cwm has deduced, then increasing levels give you more and more of the gorey details of what happens inside. If a rule isn't firing, look for "no way" in the debug output.

Get cwm to read your input file and output them again (`cwm foo.n3 > ,foo.n3`). Looking at the file in its new format sometimes shows up a bug: is that *really* what you meant?

Think.

Tips

When you use N3, you find that all your files, data and rules, can all be in the same language. Sometimes, when using tools like `make`, it is convenient to gibe files different file extensions depending on their role. You might want to leave the rule files as .n3, but make the sensor data .sense and the analyzed data as .ana. Then you can make makefile rules to map create .ana files from .sen files.

If you need to pass parameters to your rules, for example something to search for, or your name, then pass them as command line arguments, putting them at the very end of the cwm command line, after a --with. Then, the `os:argv` builtin function can be used to pick up the value of each argument, as `"1".os:argv` and so on.

More

At this point, you should be getting the hang of it and be writing stuff. To give you some more ideas, though, there is a longer list of more complex and varied examples. These come with less tutorial explanation.

Have fun!

References

- Many More Examples
- Notation3 - Design Issues article

Thanks to contributors including Joseph Reagle who added the uncle example to the original primer.

Tim BL, with his director hat off

$Id: Processing.html,v 1.3 2003/03/03 22:40:53 timbl Exp $

Tutorial: Built-in functions Cwm

The processing we have done so far involves matching existing data against a template, and where a match occurs, generating more inferred data. This is much like which you do with a database using SQL, and just as with SQL, in practice you need to be able to combine it with basic arithmetic and string operations, and so on.

This is done by some magic "built-in" properties for which cwm knows the meaning and can test the validity of a statement or calculate the rest of the statement given part of it.

Built-in *functions* are properties which cwm can calculate the object, given the subject.

Built-in *inverse functions* are properties which cwm can calculate the subject, given the object.

Some built-ins are both. Examples are log:uri, the relationship between a resource and its URI, and time:inSeconds, the relationship between a date-time in standard string form and the date-time as number seconds since the start of the era, One can work these either way.

Relational operators are bultins weher you can't calculate either side from th other, but you can test for truth if both sides have been realoved. Examples are comparison operations such as `math:equalTo` and `string:greaterThan`.

A complete list of of bultin- functions is available.

Builtin-in functions in cwm **only work** when they are used inside the **left-hand side of a rule**.

They are only used to figure out what things could take the place of a variable. If you just add data to the strore with a built-in function in it, it is just stored.

Let’s make a more complicated thermostat rule::

```
{ :thermostat :temp ?x.  ?x math:greaterThan "70" } log:implies { :cooling :power "high" } .
```

The first part, `:thermostat :temp ?x`, is statisfied by looking in the strore, where presumably the temperature at the thermostat is stored as, say, `:thermostat :temp "76.4"`. Under the hypothesis that ?x is is "76.4", then the second part, which now looks like "76.4" math:greaterThan "70" , is satified just by the built-in property of the math:greaterThan operator.

You can use path expressions to invoke builtins: this is of course especially useful for chaining them.

```
{ "1".os:argv^log:uri  log:semantics ?f } => { ?f a :InputFormula }.
```

You could read this along the lines of "If the first command line argument -- well, whatever has that as a URI -- has semantics f, then f is an input formula". Here it is in longhand:

```
{ "1".os:argv  ?x.
  ?d log:uri   ?x.
  ?d log:semantics ?f
} => {
  ?f a :InputFormula
}.
```

Some functions need more than one parameter. The convention we have used is that they take lists as the subject.

```
{   ((?x.tempInF "32").math:difference "0.5555") math:product ?c) } => { ?x tempInC ?c}.
```

Let’s look at that one without the path expressions.

```
{   (?x.tempInF "32") math:difference ?a.
    (?a "0.5555") math:product ?c.
} => {
     ?x  tempInC ?c.
}.
```

Tip: Its useful to think of where the rule engine is going to start with some data, which will alow it to find the values of variables.

{ ?x math:greaterThan ?y. } => { ?x :moreInterestingThan ?y }.

This doesn’t give cwm, a forward reasoner, much to go on. It won’t list all pairs of number where the first is greater than the second. A backward reasoner, such as Euler, will be able to use that. Future semantic web engines will get smarter about picking rules to use and algorithms to use them with.

These built-in functions give allow you to use basic properties of strings and numbers in your rules. They also allow you to do pragmatic things, such as pick up command line parameters and environment variables, which allow the whole rules system to be parameterized. They also allow you do do somthing else. They allow you to make rules which interrogate the Web, parse documents, and look objectively at the contents of documents. This opens up some very interesting possibilities -- so much so that it warrants a move to the next chapter.

References

- Many More Examples
- Notation3 - Design Issues article
- For comparison - the XQuery functions

Thanks to contrubters including Joseph Reagle who added the unclue example to the original primer.

Tim BL, with his director hat off

$Id: Built-In.html,v 1.4 2003/02/16 03:35:49 timbl Exp $

Comparing Rule-Based Systems

Cwm acts as a rules processor, using information written in N3 rules to guide it in manipulating the RDF/N3 information it has stored. While rules processors are not exactly commonplace, and understanding them is not manditory for the working programmer, they do have a long and solid history. Where does cwm fit into that history?

The field is sometimes called Knowledge-Based Systems or Expert Systems.

We can perhaps divide the territory into four camps, as follows.

Automated Theorem Provers

Automated reasoning using first-order became generally feasible in 1965 with Robinson’s resolution and hyperresolution algorithms. Today a raft of automated theorem provers continue this tradition, but they see little use in general computing.

A focal point for this research is Thousands of Problems for Theorem-Provers (TPTP), which includes links to provers and a conversion utility between logic languages. You can play a little with its web form (try problem ALG001-1).

In the RDF/Semantic Web community, people have used at least OTTER and SNARK.

Logic Programming (Prolog)

In 1970-1972, Prolog introduced Logic Programming, which took a restricted form of first-order logic (Horn clauses) and offered to prove things with them in a deterministic order, very much like running a program. Always chaining backward from a query.

(1970-1975 also saw the introduction of C, Pascal, Scheme, Smalltalk, and Microsoft BASIC.)

Production Systems (OPS5, CLIPS, JESS)

```
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?production+system
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?OPS5

Usually (but not always) chaining forward from the givens.
Same timeframe as awk.
```

Modern Hybrid

```
Euler?
cwm...
http://www.ksl.stanford.edu/software/IW/
http://belo.stanford.edu:8080/iwregistrar/Lookup?type=1&value=JTP&mode=full
```

$Id: rule-systems.html,v 1.1 2003/03/24 02:44:40 sandro Exp $

Glossary

These are not formal definitions - just phrases to help you get the hang of what these things mean. The definition terms are linked back to more information where available.

Class [p ??]	A set of Things [p 41] ; a one-parameter predicate; a unary relation.
domain [p ??]	For a Property [p 41] , a class of things which any subject of the Property [p 41] must be in.
Formula [p ??]	An (unordered) set of statements [p 41] . You can write one out in N3 using {braces}.
context	The relationship between a statement and a formula containing it.
cwm	(From: Closed world machine; valley.) A bit of code for playing with this stuff, as grep is for regular expressions. Sucks in RDF in XML or N3, processes rules, and spits it out again.
filter [p ??]	A set of rules [p 41] which are used to select certain data from a larger amount of information.
N3	Notation3, a quick notation for jotting down or reading RDF semantic web information, and experimenting with more advanced sematic web features.
object [p ??]	Of the three parts of a statement, the object is one of the two things related by the predicate. Often, it is the value of some property, such as the color of a car. See also: subject, predicate.

predicate
: Of the three parts of a statement, the predicate, or verb, is the resource, specifically the Property, which defines what the statement means. See also: subject, object.

Property [p ??]
: A sort of relationship between two things; a binary relation. A Property can be used as the predicate [p 41] in a statement [p 41] .

range [p ??]
: For a Property [p 41] , its range is a class which any object [p 40] of that Property [p 41] must be in.

rule
: A loose term for a Statement that an engine has been programmed to process. Different engines have different sets of rules. cwm [p 40] rules are statements [p 41] whose verb is `log:implies`.

Resource
: That identified by a Universal Resource Identifier (without a "#"). If the URI starts "http:", then the resource is some form of generic document.

Statement
: A subject, predicate and object which assert meaning defined by the particular predicate used.

subject
: Of the three parts of a statement, the subject is one of the two things related by the predicate. Often, it indicates the thing being described, such as a car whos color and length are being given. See also: object, predicate

Thing
: In DAML, a generic name for anything - abstract, animate, inanimate, whatever. The class which anything is in. (In RDF parlance, confusingly, rdf:Resource.) Identified by a URI with or without a "#" in it.

Truth
: In the log: namespace, a Class of all formulae which are true.

type
: A particular property used to assert that a thing is in a certain Class. The relationship between a thing and any Class it is in.

URI
: Universal Resource Identifier. The way of identifying anything (including Classes, Properties or individual things of any sort). Not everything has a URI, as you can talk about something by just using its properties. But using a URI allows other documents and systems to easily reuse your information.

References

- Many More Examples
- Notation3 - Design Issues article

Tim BL, with his director hat off

$Id: Glossary.html,v 1.4 2003/03/04 19:38:42 timbl Exp $