Semantic Web
Tutorial Using N3

Tim Berners-Lee
Dan Connolly
Sandro Hawke

For Presentaton
May 20, 2003

http://www.w3.org/2000/10/swap/doc

Table of Contents

1 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 allow one to imagine what life will be like when it is widely deployed. 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.

The material in these notes may be deeper in parts than the tutorial itself, which is limited to 6 hours.

1. Writing data (using Statements, URIs, and Vocabularies)
    - Primer: Getting into RDF & Semantic Web using N3
    - Sidebar: Comparing with other data formats
    - Sidebar: Installing cwm (Install it during the break)
    - Sidebar: Cwm command line arguments
2. More Syntactic Sugar, More Ontological Power
    - Shorthand: Paths and Lists
    - Ontologies: More powerful information about vocabularies
    - Writing rules
    - Processing RDF data using rules
3. Procesing data with cwm/n3
    - Built-in functions in rules
    - Sidebar: List of built-in functions in cwm
    - Sidebar: Comparing with other rules systems
4. Semantics + Web = Semantic Web
    - Reaching out into the Web
    - Trust
    - application integration: travel tools

- Glossary

2 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.

2.1 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: the object (only) can be a literal, such as a string or integer:

```
<#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.

2.2 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.

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 owl:  <http://www.w3.org/2002/
                        07/owl#> .
```

These are the RDF, RDF schema, and OWL 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.

2.3 Making vocabularies

Things like dc:title above are RDF *Properties*. When you want to define a new vocabulary 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 the RDF Schema and OWL 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.

2.3.0.1 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.

2.3.0.2 Choosing a namespace and publishing your vocabulary

Good on-line documentation for vocabulary terms helps people read and write RDF data. Writers need to see how a term is supposed to be used; readers need to see what it is supposed to mean. People developing software which uses the terms need to know in particular detail exactly what each URI means.

If you document your vocabulary using the RDF Schema and OWL vocabularies, then your documentation will be machine-readable in a variety of interesting and useful ways, as mentioned above and covered in more detail in Vocabulary Documentation. This kind of RDF-documentation-in-RDF is sometimes called a "schema" or an "ontology."

The easiest way to help people find your documentation is to make the URIs you create as vocabulary terms also work in a web browser. This happens automatically if you follow the naming convention we use here, where the vocabulary definition document has a URI like `http://example.com/terms` and it refers to its terms like `<#Woman>`. With the @prefix declaration above, this gives the URI `http://example.com/terms#Woman` which should work in any browser to display the definition document.

Ideally, you should publish your documentation on the web using a server and portion of URI-space which are owned by an organization which can commit to maintaining them well into the future. That way, many years down the road, RDF data using your terms will still be documented and potentially understandable. The convention of putting the current year into the URI can help with stability; some day people may be tempted to re-use `http://example.com/food-vocabulary`, but they will probably only touch `http://example.com/2003/food-vocabulary`, when they really mean to upgrade the documentation there. In some circumstances you can also achieve increased stability by using a specialized domain name which may be insulated from possible organizational renaming and trademark issues.

Of course if you are just playing around, you can use a file (say `mydb.n3`) in the same directory as the rest of your work. When you do that, your can simply use `<mydb.n3#>` as your namespace identifier, because in N3 (as in HTML), the URIs can be specified relative to the current location.

3 Shorthand: 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.

3.1 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 an 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 necessarily 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.

3.2 Lists

A common need is to represent an ordered collection of things. This is done in RDF as a rdf: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 `rdf:first` and `rdf:rest`. `rdf:first` is the relationship betwen a list and its first component. `rdf:rest` if the relationship between a list and the list of everything except its first element. `rdf:nil` is the name for the empty list. Therefore,

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

is equivalent to

```
[ rdf:first "Monday";
  rdf:next [ rdf:first "Tuesday";
             rdf:rest [ rdf:first "Wednesday";
                        rdf:rest rdf: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.

4 Vocabulary Documentation

As we read and write N3, communicating in RDF, we need to share an understanding of what each URI means. We often pick URIs which offer clues about meaning, such as `http://www.w3.org/2000/10/swap/test/demo1/biology#Dog`, but the text of the URI still gives only a clue. Would a wolf qualify as a one of these? How about a Dingo? We can't tell just by looking at the name. It's even possible the URI text is misleading, and the intended meaning has nothing to do with dogs.

A good technique for addressing these issues is to publish a document which carefully describes the intended meaning of each term. If this description is done carefully enough, following a precise structure, software can help us understand what the terms mean, check for some kinds of errors, and sometimes even dynamically update itself to use and recognize new URIs.

There is a vocabulary of about fifty terms (URIs) being developed by W3C Working Groups to support this kind of precise documentation. About a dozen of the terms are part of the RDF Vocabulary Description Language 1.0: RDF Schema, for which we use the "rdfs:" namespace abbreviation. The others are part of the OWL Web Ontology Language. All fifty are listed and cross referenced in the OWL Guide Appendix .

@@@ Old Material

In RDF, URIs are used a lot like was use words and phrases in natural language. Each URI has something it stands in place of, represents, refers to, or denotes.

Each URI

An *ontology* or is a precise but sometimes incomplete specification of a vocabulary. It often consists of both human-readable documentation and machine-readable declarations. 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 (what is the class hierarchy?) and what they have in common (what are the properties of each instance?).

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

Historically, RDF has had a variety of ontology languages, including RDFS, DAML+OIL, and OWL with its three different "species", OWL Lite, OWL DL and OWL Full. However, with the exception of DAML+OIL which is obsolete, they all fit together in OWL Full. So use OWL Full, and if you use only a restricted subset of the language, then you will have additional performance guarantees.

4.1 Classes, Properties, Individuals, and Datatype Values

In designing an ontology, it can be useful to divide the world into four different kinds of things. (Occasionally something will fit into more than one of the areas; that's okay with OWL Full.)

- 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

4.2 What is it for?

An ontology can be used in several different ways:

- By ontology creators, to keep the ontology logically consistent.
- By data providers and consumers, to help communicate which information might be provided and what the terms mean.
- By data providers and consumers, to help "validate" the data, finding constructs which are definitely or probably errors
- By consumers, to license inferences, possibly from multiple data sources, supporting data "fusion"

4.3 Subclass Reasoning

```
:Man s:subclassOf :Human .
:YoungMan s:subclassOf :Man .
```

4.3.1 OWL Inference

```
:Joe a :YoungMan -->  :Joe a :Man. Joe a :Human.

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

4.3.2 "Lint" Processing

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

Joe is a YoungMMMan -->  Warning: "YoungMMMan" used as a Class but
                                  not mentioned in schema.
```

4.4 Domain, Range, Cardinality

```
:father s:domain :Human; s:range :Man.
:Sara :father :Alan.
```

```
:Sara a :Human.  :Alan a :Man.
```

```
:father owl:cardinality "1".
:Sara :father :Alan.
:Sara :father :MrFoster.
```

```
:Alan = :MrFoster.
```

4.5 Sameness and Difference

@@@ owl:sameAs, sameIndividualAs, ...

distinctFrom, disjointClass

5 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 some central heating vocabulary) "If the thermostat temperature is high, then the heating system power is zero", 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 syntax which is not standard.

5.1 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 formula above might be written

$$<mo></mo><mi>x</mi><mo>|</mo><mi>temp</mi><mo>|</mo><mi>thermostat</mi><mo>,</mo><mi>x</mi><mo>}</mo><mo></mo><mo></mo><mi>power</mi><mo>|</mo><mi>cooling</mi><mo>,</mo><mi>x</mi><mo>}</mo></mo>$$

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 actually 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 explicit variables.

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

5.2 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 pseudo-properties which are really special language 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 which don't have nested formulae inside the antecedent.

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 often used in ntriples but in N3, you can normally use the [] syntax to avoid using any variables at all.

5.3 Shorthand symbols for implies, etc.

When it is used as a predicate (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 14] .

6 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.

6.1 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. **Tip**: 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?)

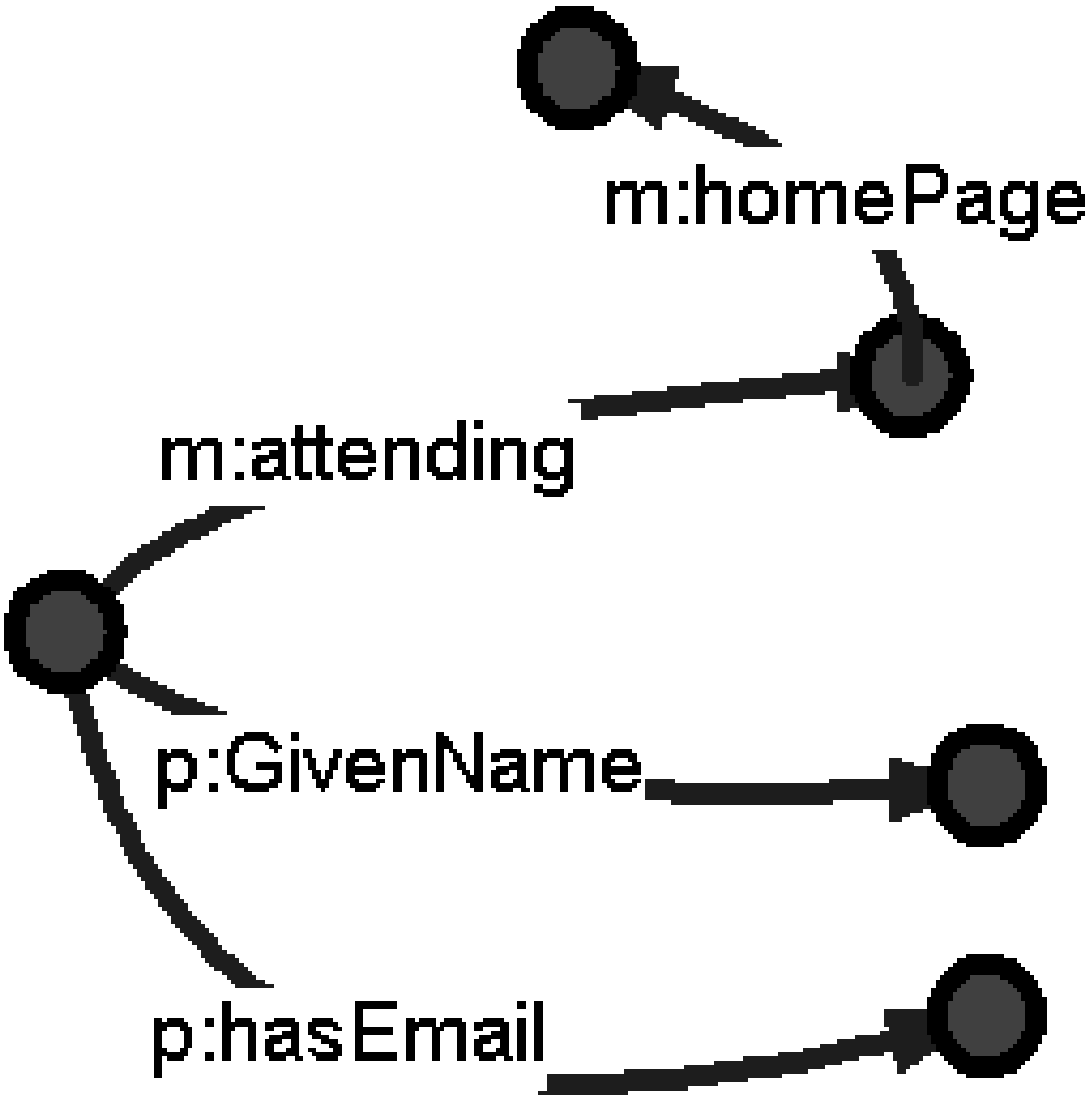
6.2 Merging data

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.

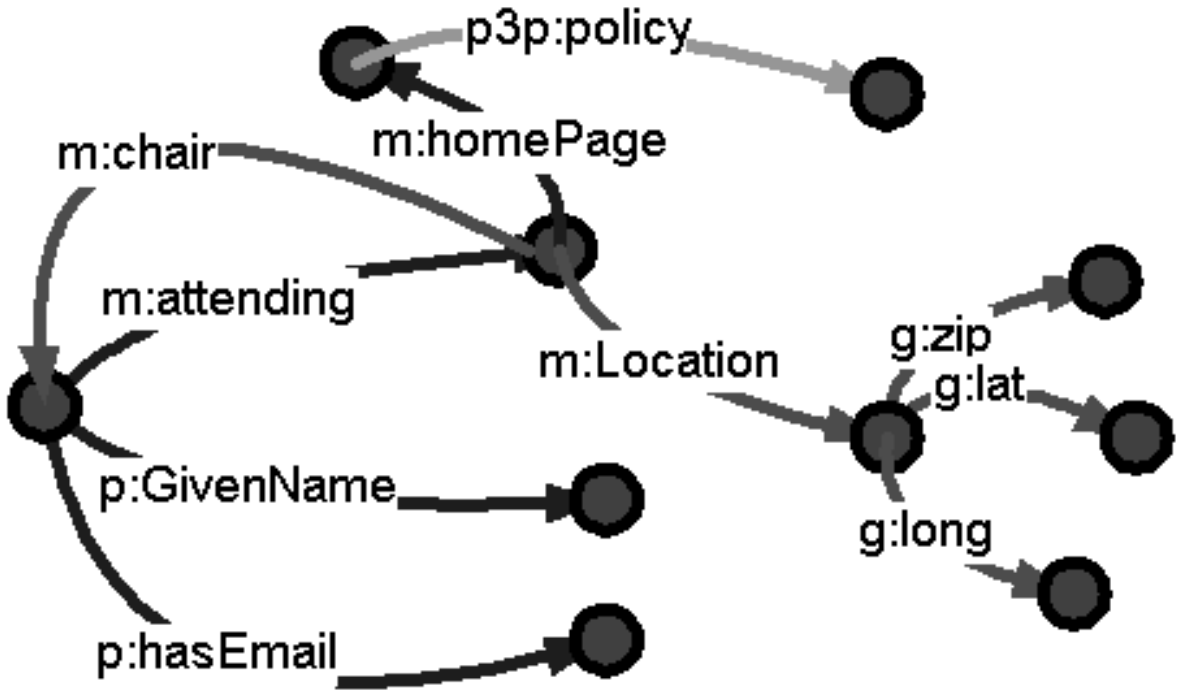
```
cwm blue.n3 red.n3 green.n3  > white.n3
```

reads three n3 files, and then outputs the results in N3 to white.n3.

This is particulaly useful if the files have used the same URIs for the same concepts. Suppose the first has that someone is attending a meeting which has a given home page.



and the second (red) has some information about the meeting, and a third (green) has some information about the meeting's home page. Merging the three gives us an interconnected web, including the fact that the attendee is actuallt chairing the meeting.



6.2.1 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

6.3 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.

6.3.1 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.

6.4 Report Generation

All the examples above process 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?

6.4.1 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, ...

6.4.2 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

6.5 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.

6.6 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 give files different file extensions depending on their role. You might want to leave the rule files as .n3, but make the sensor data `.sen` 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.

6.7 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!

7 Built-in functions in 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, string operations, and so on.

This is done by some magic "built-in" properties of which cwm knows the meaning. It automatically test the truth of a statement, or calculate the rest of the statement given part of it.

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

Built-in *inverse functions* are properties for 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 where you can't calculate either side from the other, but you can test for truth if both sides have been resolved. 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.

---

8 Trust

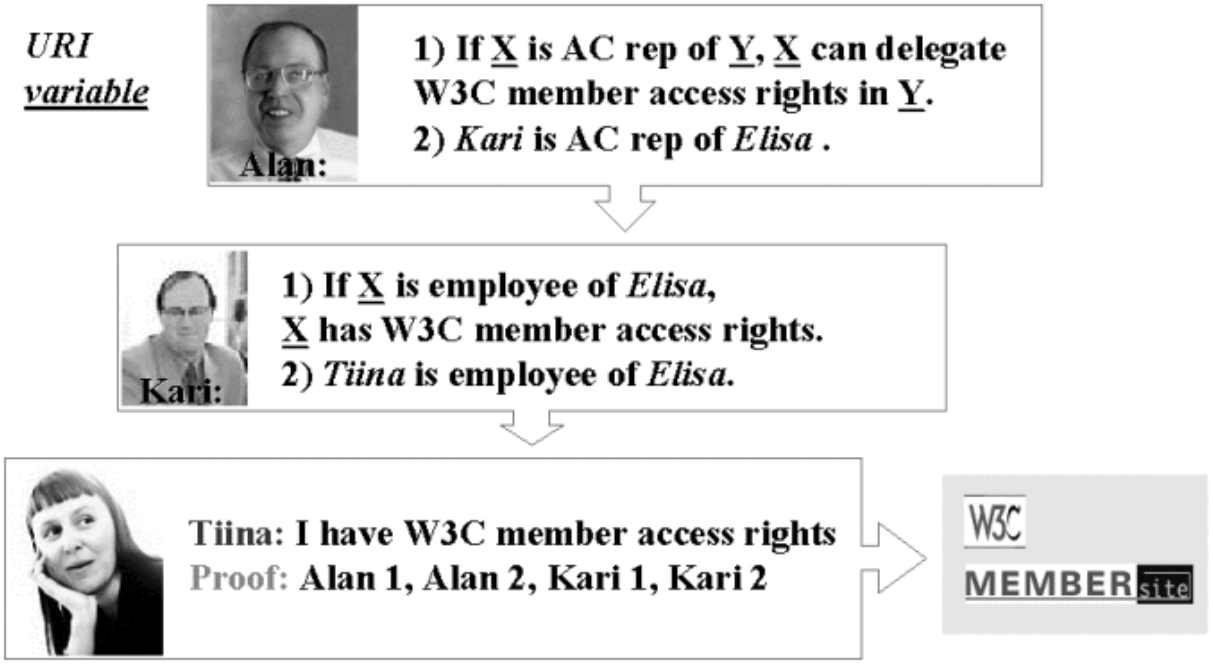
In the real world, you don’t want a program doing things without being careful. One of the more difficult things is figuring out what rules you really use for deciding what data to trust. The Semantic Web doesn’t make that social problem much easier. When you have figured out a trust model, the Semantic Web allows you to write it down. Not only that -- if we introduce cryptographic built-in functions, we can actually make a system which will use public key cryptography to check the authenticity of data.

This can of course get very big, and as complicated as your social or business environment. However, we can give a taste with a simple example. The idea here is not to learn this particular cryptographic library, but to get the feel of what can be done.

8.1 Delegated authority

Suppose we want to set up an access control system for the W3C *Member site*. (Yes, even though companies support W3C just because is a good thing, many quite reasonably need this "what’s in it for me?" justification).

We will formalize the social arrangements and then we will write the rules which encapsulate those arrangements so that they can be followed by a machine. Let’s just take the case of a web page which is only accessible to W3C members. What does that mean? Well, when a company or organization joins the W3C Advisory Committee representative is chosen as the liaison between everyone at the organization and everything happening at W3C. Currently (2003) when someone says they belong to a company and want to access the member site in that role, we check, by email, with the AC rep. Let’s design a system to do it with digital signature.



In this picture, Alan Kotok is the W3C associate chairman who deals with membership. In this case, Kari is the AC rep for W3C member *Elisa Communications*. Alan delegates to Kari the right to say who, as a employee of Elisa, get access to the W3C Member Site.

8.1.1 Master Key

Alan's authority to define who is a W3C is represented by a digital key. he generates it using the command

```
cwm access-gen-master.n3 --think --purge > access-master.private
```

where the access-gen-master.n3 contains:

```
#  Generate master key

@prefix : <#> .
@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@prefix crypto: <http://www.w3.org/2000/10/swap/crypto#> .
@prefix string: <http://www.w3.org/2000/10/swap/string#> .
@prefix acc: <http://www.w3.org/2000/10/swap/test/crypto/acc.n3#>.

this log:forAll :x , :y.


{ :x crypto:keyLength "1024";
     crypto:publicKey :y } log:implies {
 :x a acc:MasterKeyPair; a acc:Secret.  :y a acc:MasterKey } .

log:forAll a log:Chaff.
log:implies a log:Chaff.
```

The `crypto:keyLength` built-in generates :x, which a string which actually contains both the private and the public keys. This is convenient, as anyone who knows a private key needs to keep track of which public key goes with it. This is not a very semantic web function, as it isn't repeatable rule you can reuse - it is just a trick for generating a key. (It will generate a different key each time, although the library at one stage didn't do that to make debugging easier).

The `crypto:publicKey` builtin is a function which strips out the private parts of a key :x, leaving only the public part :y. Note we have class here `:Secret` which we just use for labelling things which we don't want another person to know.

The log:Chaff part is to label those bits which are for deletion by the --purge command, just to leave the key file clean of stuff we don't want.

Kari also generates a key, which represents the authority has as Elisa's representative. he does this in an identical fashion, using `access-gen-elisa.n3`. The actual passing of authority happens when Kari gives the public part of his key to Alan, in some way secure enough for Alan to be sure enough it , and Alan then signs something to the effect that it is indeed Kari's key as AC rep for Elisa.

```
cwm access-elisa.public access-master.private access-sign-member-cert.n3 \
 --think --purge --with "Elisa" > access-eliza.cert
```

Here, Alan uses the private key which he (alone) has access to and the public key which Kari gave him. `access-sign-member-cert.n3` is a rule file which does the signing. It contains, in essence:

```
{ :n is os:argv of "1".
  :tk a acc:MemberKey.
  { :tk a acc:MemberKey;
        acc:authorityName :n;
        acc:junk "327462sjsdfjsakdhfkjsafd32164321"
   } log:n3String :str.
   :kp a acc:MasterKeyPair.
   :k a acc:MasterKey.
   ([is crypto:md5 of :str] :kp) crypto:sign :z
} => {
   :str acc:endorsement[acc:signature :z; acc:key :k
} .
```

:n resolves to be the name "Elisa", :tk resolves to be Elisa's public key. The string str is string form of the certificate formula, as the digital signatures work on strings. The MD5 checksum of the string is formed, and that is signed with the master key pair (:kp). All this happens as the left hand side of the rule is resolved, and when it has, then the conclusion is that that string has an endorsement which has a given signature and a given public key. This endorsement information is the certificate.

In exactly the same fashion, Tiina, the employee, makes a key, and gets a certificate from Kari about that key.

Then the time comes that Tiina wants to access the member site. She makes up a request for the web page, and signs it with her private key.

```
cwm access-tiina.private access-sign-request.n3 \
 --think --purge --with http://www.w3.org/Member > access-1.request
```

The with command passes in the URI of the page she would like to access.

The crux of this whole system is the file `access-rule.n3` which tells the web site how to trust a request. And the crux of that is its first rule. It is a bit simple but you should get the idea. It says that:

> if a request is supported by a key, and there is a certificate -- signed itself with k2 -- which says k is a good request key, and that there is some other certificate, signed with the master key, that says that k2 is a member key, then the request is a good request.

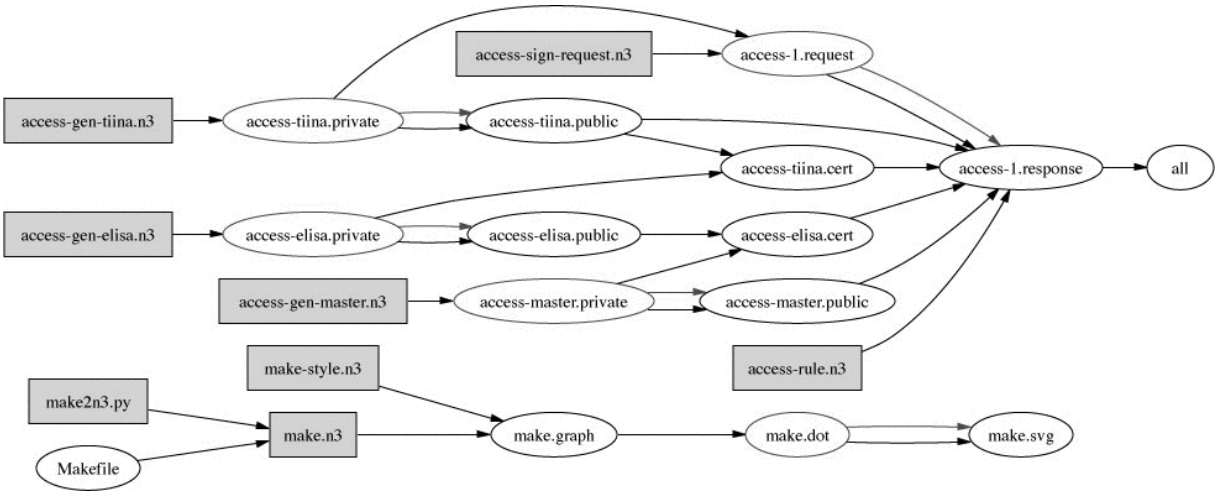
The other details of what "supported" means are below.

```
this log:forAll :d, :F, :G, :k, :k2, :k3, :kp, :x, :request, :sig, :str, :y , :z , :q .

# The rule of access.
#
```

```
#  acc:requestSupportedBy means that it correctly claimed to be
#        signed by the given key.

{ :request a acc:GoodRequest } is log:implies of
{
    :request acc:forDocument :d;
             acc:requestSupportedBy :k.

    []       acc:certSupportedBy :k2;    # Certificate
             log:includes { :k a acc:RequestKey }.

    []       acc:certSupportedBy [a acc:MasterKey];    # Certificate
             log:includes { :k2 a acc:MemberKey }.

}.

#  What is a Master key?
#
#  (we could just put in the text here)

{ <access-master.public> log:semantics [
       log:includes {:x a acc:MasterKey}]
} log:implies {:x a acc:MasterKey}.


#  What do we belive is a request?
#  We trust the command line in defining what is a request.

{  "1"!os:argv!os:baseAbsolute^log:uri log:semantics :F.
   :F log:includes { :str acc:endorsement[acc:signature :sig; acc:key :k]}.
   :k crypto:verify ([is crypto:md5 of :str] :sig).
   :str log:parsedAsN3:G } log:implies { :G acc:requestSupportedBy :k }.

#  What do we believe from a signed request?
#   - what it says it is asking for.
#   - what it quotes as credentials
#  It could actually enclose a copy of the credentials inline,
#  but here we use the web.  A credential is a document which
#  provides evidence in support of the request.

{:G acc:requestSupportedBy :k; log:includes {:G acc:forDocument :d}} =>
   {:G acc:forDocument :d}.

{:G acc:requestSupportedBy :k; log:includes {:G acc:credential :d}} =>
   {:G acc:credential :d}.

# What do we belive from a signed credential.
#
# In this case, just note that a key supports the signed formula.
# The fact of this support is used in the access rule above.
# We don't actually beleive everything the certificate says.

{  [] acc:credential [ log:semantics :F ].
   :F log:includes { :str acc:endorsement[acc:signature :sig; acc:key :k]}.
   :k crypto:verify ([is crypto:md5 of :str] :sig).
   :str log:parsedAsN3 :G } log:implies { :G acc:certSupportedBy :k }.
```

The important thing is that we are really trusting very specific information from different sources.

Here is the complete data flow in summary.



The bit at the bottom shows how the picture itself, `make.svg`, was generated from the Makefile - via RDF of course. Some people have actually started to use RDF just because of the diagram drawing facility, but we hope by this stage that you will have a longer term and wider vision of the value of Semantic Web!

8.2 Conclusion

You will notice that in this example we didn't use a PKI certificates looked up in a x.509 hierarchy. We could have. We would have used an RDF interface to PKI.

There are two interesting reasons not to. One is that in designing the system from scratch we defined exactly how it works. We defined what information Alan signed about Kari, without involving any other information actually identifying Kari as an individual person in the world. We don't need to know Kari's social security number with the Finnish government. The key represents Kari in the role he plays in this system. Because we are using RDF, we can put whatever information into the certificate we like. We don't have to worry about extending the certificate format. Because what we are doing concerns the relationships between the social entities involved without bringing in any superfluous external authority, it is more weblike in nature. We are freer to model the actual social trust system we use.

The other interesting point is that the amount of software for working out whether a request is valid is in fact quite small. The access rule is the secure part which is specific to this system. Apart from that the trusted code base is a general-purpose engine and a cryptographic library.

One can make the code base even smaller. Tiina can herself run the access rule herself, and generate a proof - an output of the bits of data used and rules used step by step. The web server then only has to check this proof. Although it wasn't very difficult, it did involve running a simple inference engine. Checking a proof is in fact simpler. (Cwm is being adapted to generate proofs, but it can't do it yet for this case) . We assumed that Tiina had a program to generate the request which know exactly how to do it. You could imagine a more difficult situation in which her access agent had to search for some connection between her and W3C, and find a way of justifying the request. In this case, finding the credentials could have taken a long time, or used special knowledge -- things which the web server itself wouldn't want to worry about. The proof sent would then contain just the essential information actually used to justify entry. The web server's trusted code base would be a proof checker and a digital signature validator. That's the sort of thing it every processor could have in ROM when it is made.

The ability to write rules about what document actually says what allows us to make systems which behave in predicatable and appropriate ways in the real world. Cryptography gives use the ability to build secure systems, which in a weblike way allow us to express what the real trust situation.

9 Semantic Web Application Integration: Travel Tools

The bane of my existence is doing things I know the computer could do for me. When I got my proposed July 2001 travel itinerary in email, I just couldn't bear the thought of manually copying and pasting each field from the iterary into my calendar. I started putting the Semantic Web approach to application integration to work.

The Semantic Web approach to application integration emphasizes data about real-world things like people, places, and events over document structure. Documents are important real-world things too, of course. And Semantic Web data formats benefit from the internationalization support in XML and the growing infrastructure of tools. But most XML schemas are too constrained, syntactically, and not constrained enough, semantically, to accomplish these integration tasks:

- plot an itinerary on a map [p 30]
- import travel itineraries into my iCalendar-happy desktop PIM [p 33]
- produce a brief summary of an itinerary for use in plain text email [p 36]
- check proposed work travel itineraries against family constraints [p 36]
- import travel itineraries into my PDA calendar [p 37]
- tell me when my travel schedule brings me unusually near a friend/colleague [p 37]
- produce animated views of my travel schedule or past trips
- find conflicts between teleconferences and flights [p 37]

9.1 Working with legacy data

While more and more of the data in our lives is available in the Semantic Web, there will always be a place for mechanisms that extract the statements implicit in legacy data.

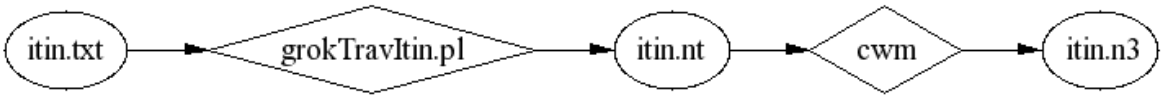
The data comes from the travel agency like this, probably dumped from their database system:

```
07 APR 03 - MONDAY
AIR AMERICAN AIRLINES FLT:3199 ECONOMY
OPERATED BY AMERICAN AIRLINES
LV KANSAS CITY INTL 940A EQP: MD-80
DEPART: TERMINAL BUILDING B 01HR 36MIN
AR DALLAS FT WORTH 1116A NON-STOP
```

I hope that before too long they'll dump it from their database directly into RDF or perhaps in XML using some travel industry vocabulary, but

- before they'll do so, somebody will have to show them why it's valuable
- sometimes, for the short term, reverse-engineering the structure of their data is cheaper than getting them to change their processes

So I wrote a perl script (grokNavItin.pl [p 37]) to extract statements from the data:



The output of the perl script, `itin.nt`, is in n-triples, a line-oriented serialization developed in the RDF Core working group for testing parsers. For visual inspection and debugging, we use cwm to pretty-print it in N3. The result look like this:

```
:_gflt3199_3     a :_gECONOMY_5;
     k:endingDate :_gdayMONDAY07_2;
     k:fromLocation <http://www.daml.org/cgi-bin/airport?MCI>;
     k:startingDate :_gdayMONDAY07_2;
     k:toLocation <http://www.daml.org/cgi-bin/airport?DFW>;
```

```
        t:arrivalTime "11:16";
        t:carrier :_gAMERICANAIRLINES_4;
        t:departureTime "09:40";
        t:flightNumber "3199" .

    :_gAMERICANAIRLINES_4     a k:AirlineCompany;
        k:nameOfAgent "AMERICAN AIRLINES" .

    :_gECONOMY_5     r:value "ECONOMY" .

    :_gdayMONDAY07_2     a k:Monday;
        dt:date "2003-04-07" .
```

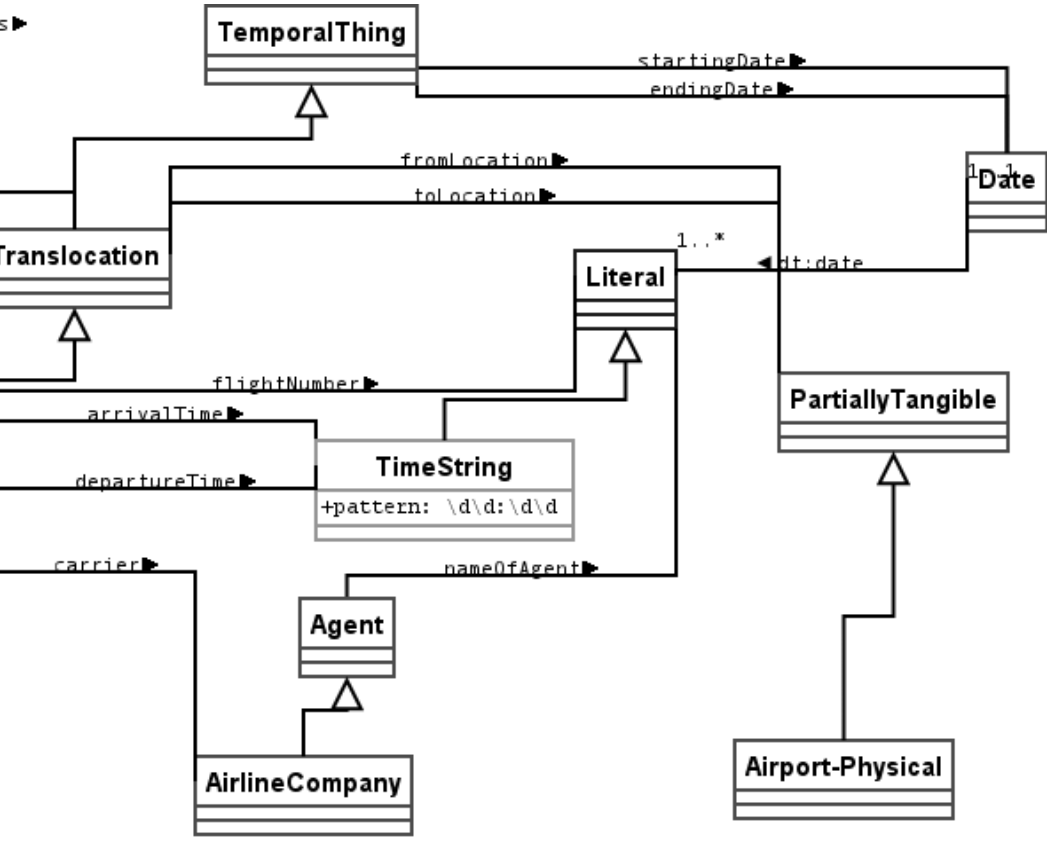
9.1.1 Choosing a Vocabulary: Build Or Buy?

The import script not only bridges the syntactic gap between the legacy data and RDF, but it also translates the vocabulary of terms used in the data into URI space. This raises the classic build-or-buy choice:

use (buy) an existing, general-purpose vocabulary
: If we can accept the risk of putting word's into the source's mouth, we can benefit from an economy of scale of shared vocabulary such as the cyc ontology of common sense transportation terms.

build a vocabulary just for this purpose
: A special-purpose vocabulary isolates the data from risks of version skew and such.

Early versions of the import script used a special-purpose vocabulary; rules to relate this vocabulary to other vocabularies were developed one at a time. But eventually a pattern of using the general purpose cyc ontology emerged, and the expected benefit of maintaining the special-purpose ontology was dominated by the cost. More recent versions convert directly to terms in shared ontologies, except in the case where custom terms were needed:



- cyc terms (prefix `k:`) in red
- DAML airport ontology terms in purple
- custom travelTerms (prefix `t:`) in green. e.g. `departureTime`, `flightNumber`, ...; see travelTerms, in RDF/xml, RDF/n3.
- RDF standard terms (prefix `r:`)in blue
- XML Schema terms (prefix `dt:`)in orange

Note that **mixing vocabularies in RDF is easy**; so easy, compared with the general problem of mixing XML namespaces, that I hardly notice it at all. Within the basic subject/predicate/object abstract syntax, terms can be combined freely. Migrating to more specialized or more generalized terms is cheap, using `rdfs:subPropertyOf` and the like.

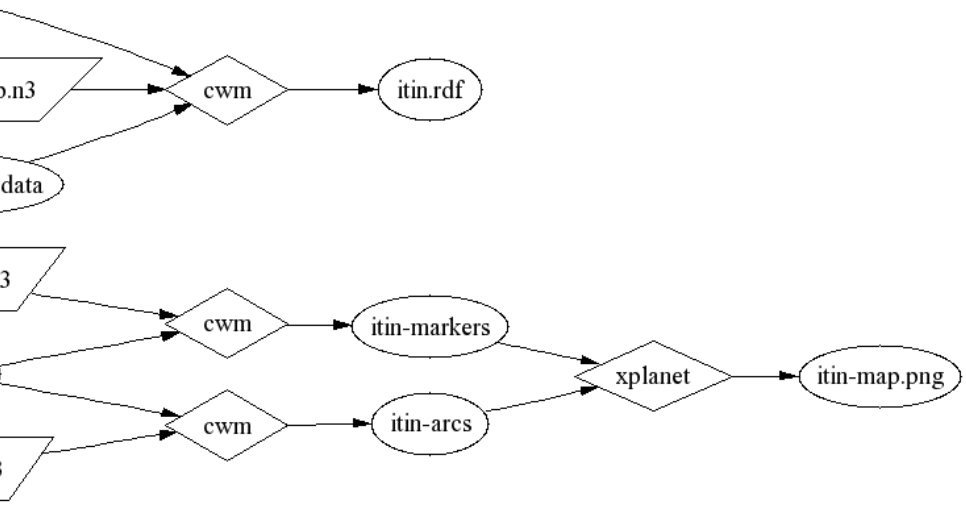
9.2 Integration with mapping tools

Let's exploit the effort we have put into going beyond formalized document structure into formalized data about the real world. Folks in the DAML project have imported airport lat/long data into the semantic web; we can use log:semantics to reach out and get it with rules like these, excerpted from airportLookup.n3:

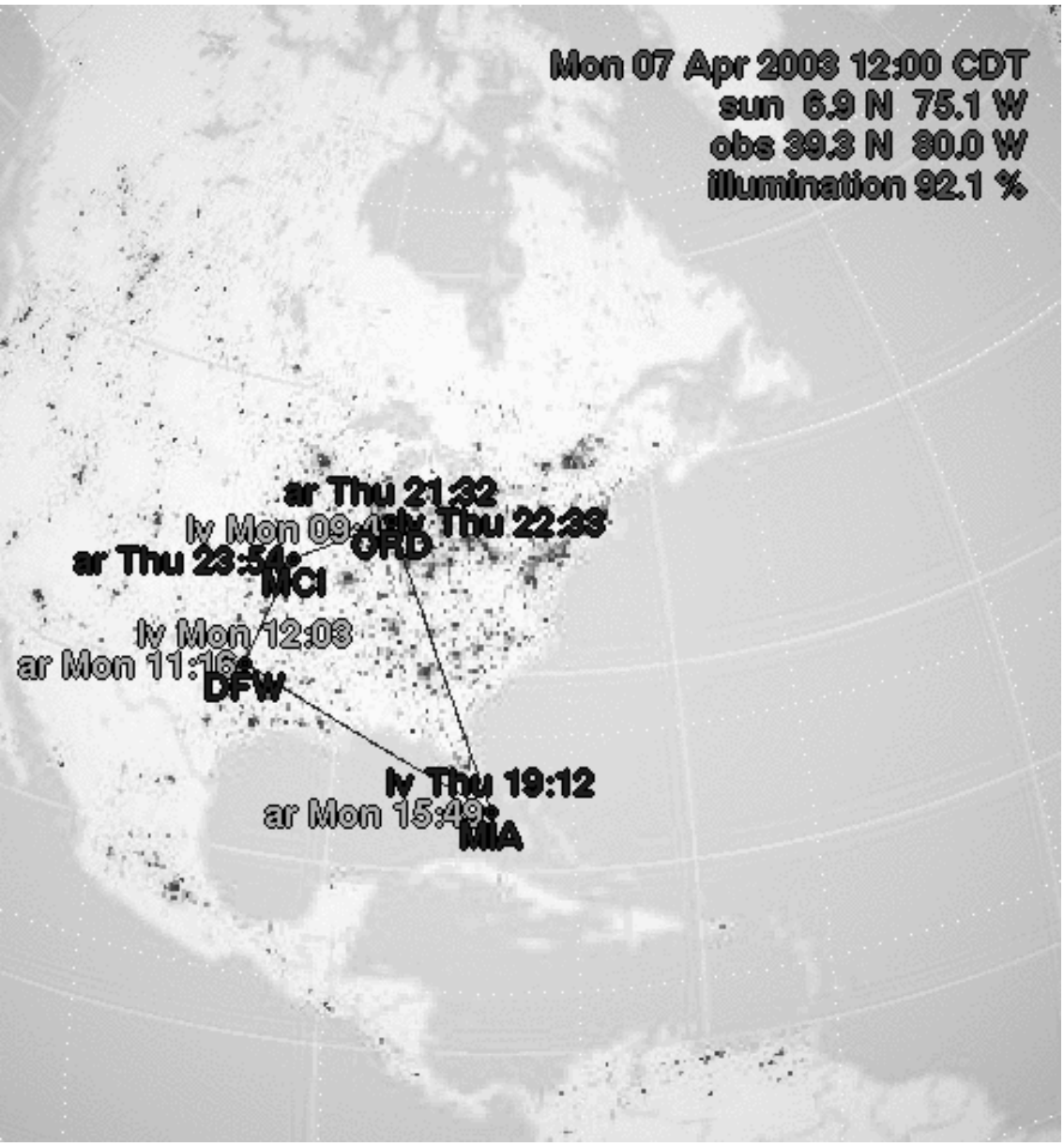
```
# well-known airports...
{ :X a :Y; #@@kludge...
    log:uri [ str:startsWith "http://www.daml.org/cgi-bin/airport?" ] }
 log:implies { :X a :AirportKnownToDAML }.

{ :X apt:iataCode :K.
  :Y log:uri [ is str:concatenation of
               ("http://www.daml.org/cgi-bin/airport?" :K) ];
}
  log:implies { :Y a apt:Airport; apt:iataCode :K; = :X }.

# we only want to look up certain airports...
```

```
{ [ k:toLocation :X ]. }
 log:implies { :X a :InterestingPlace }.
{ [ k:fromLocation :X ]. }
 log:implies { :X a :InterestingPlace }.


# believe what daml.org says about airport latitutde/longitudes...
:AirportProperty is rdf:type of
  apt:latitude,
  apt:name,
  apt:iataCode,
  apt:icaoCode,
  apt:location,
  apt:latitude,
  apt:longitude,
  apt:elevation.

{
  :P a :AirportProperty.
  [ a :AirportKnownToDAML, :InterestingPlace;
    log:semantics [
      log:includes {
        :IT :P :X.
      }
    ] ].
} log:implies {
  :IT a apt:Airport; :P :X.
}.
```

For the convenience of consumers (including ourselves), we publish in RDF/XML the results of reaching out with the rules; i.e. the itinerary including the lat/long info. Then we use the (*little documented*) `cwm --strings` output mode to generate two files, `itin-arcs` and `itin-markers`, as input to xplanet:



The resulting map shows that we have given the machine a fairly deep understanding of the itinerary:



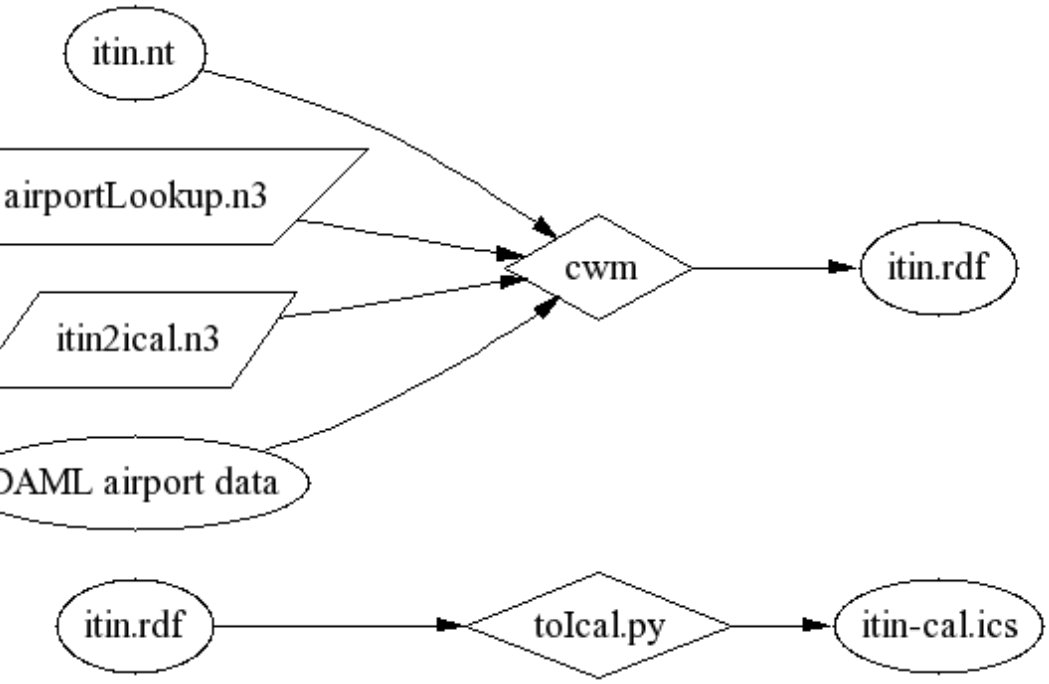
9.3 Integration with iCalendar Tools

In fact, the published RDF/XML version of the itinerary is joined not only with latitude/longitude data, but also timezone data, and elaborated via itin2ical.n3 rules into an RDF representation of the standard iCalendar syntax.

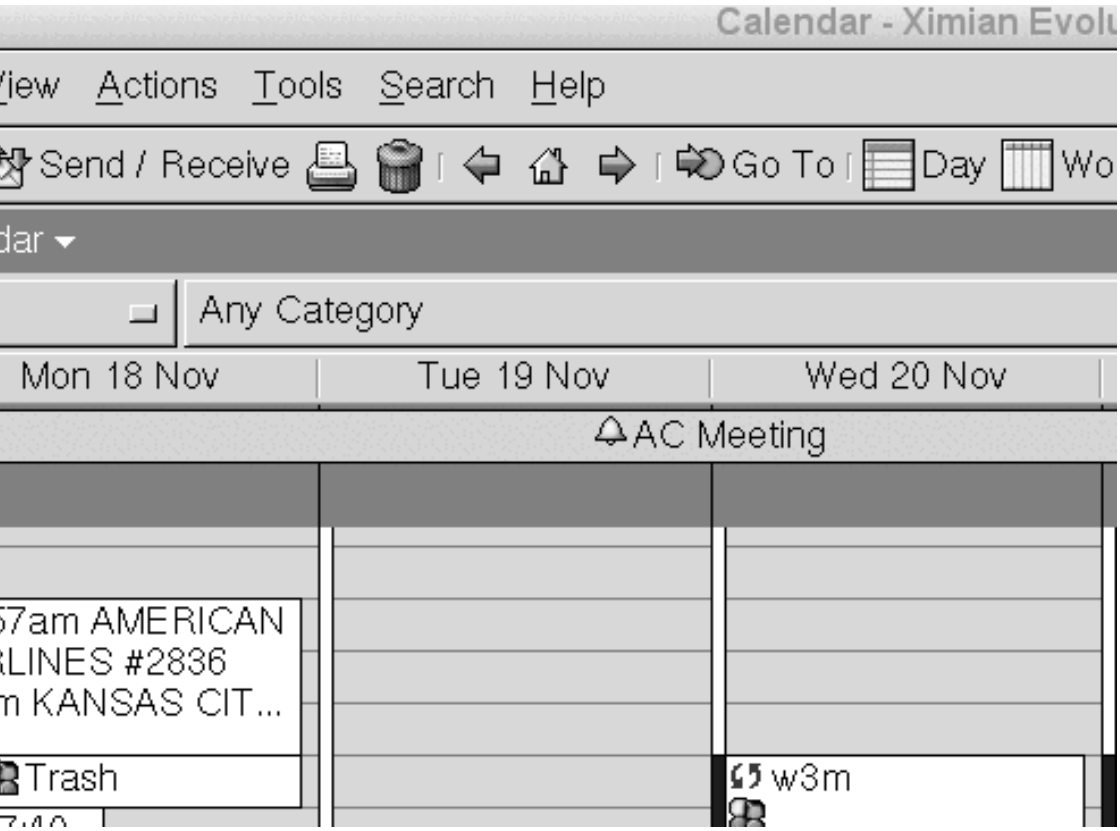
```
{ :FLT
    k:startingDate [ dt:date :YYMMDD];
    k:endingDate [ dt:date :YYMMDD2];
    t:departureTime :HH_MM;
    k:fromLocation [ :timeZone [ cal:tzid :TZ] ];
    t:arrivalTime :HH_MM2;
    k:toLocation [ :timeZone [ cal:tzid :TZ2] ].
  :DTSTART is str:concatenation of
    (:YYMMDD "T" :HH_MM ":00"). #@@ extra punct in dates
  :DTEND is str:concatenation of
    (:YYMMDD2 "T" :HH_MM2 ":00").

  ( :FLT!log:rawUri "@uri-2-mid.w3.org") str:concatenation :UID. #@@hmm... kludge?
```

```
}
 log:implies {
  :FLT a cal:Vevent;
    cal:uid :UID;
    cal:dtstart [ cal:tzid :TZ; cal:dateTime :DTSTART ];
    cal:dtend [ cal:tzid :TZ2; cal:dateTime :DTEND ].
}.
```

The final syntactic export is more complex than the markers/arcs case, so we wrote a python program, `toIcal.py`, using the cwm API, to generate iCalendar syntax.



We can import the resulting iCalendar file into an of a number of interoperable tools, such as Ximian Evolution:



9.4 Plain Text Summaries

All this rich integration is great when the tools are all working and you have plenty of bandwidth and all that, but sometimes, plain text is necessary and sufficient for the task at hand. For example, if I get mail asking when I arrive at the meeting site, mailing back a map is probably overkill, and I can't be 100% sure their desktop is iCalendar-happy.

The `cwm --strings` output mode can be really handy in these cases; we can use a few itinBrief.n3 rules ala...

```
python cwm.py itinBrief.n3 itin.nt --think --strings
```

to get a summary ala...

```
2003-04-07 09:40 - 11:16 MCI->DFW Monday AMERICAN AIRLINES #3199
2003-04-07 12:03 - 15:49 DFW->MIA Monday AMERICAN AIRLINES #68
2003-04-10 19:12 - 21:32 MIA->ORD Thursday AMERICAN AIRLINES #1477
2003-04-10 22:33 - 23:54 ORD->MCI Thursday AMERICAN AIRLINES #1081
```

9.5 Checking Constraints

Now that I have the proposed itinerary formalized, I can automatically check it against various constraints before I accept it and before I copy it to my PDA and to all the other peers that need to know about it.

Rules like "itineraries that have me leaving before 30 July are no good" are a bit tedious to formalize, but my confidence in the results is higher than my confidence in eyeballing it:

```
{
 ?D a k:ItineraryDocument; k:containsInformationAbout-Focally ?TRIP.
 ?TRIP k:subEvents
    [ k:startingDate [ dt:date ?D1 ];
      k:fromLocation [ apt:iataCode "MCI" ];
      t:departureTime ?T1;
    ].
  ?D1 str:lessThan "2001-07-30".
} => {
   ?TRIP <#leavesDaysTooSoon> ?D1;
         <#at> ?T1.
}.
```

These constraints can be checked with cwm ala:

```
$ python cwm.py proposed-itinerary.nt --think=constraints.n3
```

... and look for `<#leavesDaysTooSoon>` in the output.

9.6 Conversion for PDA import

Having already developed palmagent, an HTTP/RDF interface to my PDA, it was almost trivial to write some rules to relate this itinerary vocabulary to the RDF vocabulary for the palmpilot datebook:

9.7 Conclusions and Future Work

For the first few integration tasks, it might have been less work to just manually copy the data, field by field. But the return on investment increases with each trip I take, each system we integrate with, and each collaborator who develops and interoperable tool.

@@future work

10 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 39] ; a one-parameter predicate; a unary relation.

DAML
: Darpa Agent Markup Langauge. This is DARPA's name for the US govenment funded projects which led to, among other things, OWL. The DAML site has useful pointers and registries of useful ontologies and data in RDF.

domain [p ??]
: For a Property [p 38] , a class of things which any subject of the Property [p 38] must be in.

Formula [p ??]
: An (unordered) set of statements [p 38] . 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.

existential variable
: A term in a language (such as N3) which stands in place of a normal symbol, allowing one to consider a formula being true for some symbol being put consistently in place of the variable.

filter [p ??]
: A set of rules [p 38] which are used to select certain data from a larger amount of information.

Formula
: A set of statements [p 38] , with a list of universally quanitified variables and a list of existentially quantified variables. In N3, a literal formula if representated by braces {}.

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.

OWL
: The Web Ontology Language standard from W3C. Currently (2003/04) on the Recommensation track. An RDF vocabulary.

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 38] in a statement [p 38] .

range [p ??]
: For a Property [p 38] , its range is a class which any object [p 38] of that Property [p 38] 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 38] rules are statements [p 38] 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 OWL, 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. **Tip:** Saying something is a Thing doesn't tell anyone anything, which is why you don't see it much.

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.

universal variable
: A term in a language (such as N3) which stands in place of a normal symbol, allowing one to consider a formula being true for any symbol being put consistently in place of the variable.

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.

variable
: Either a universal variable [p 39] or an existential variable [p 38] .