

# Euler Proof Mechanism

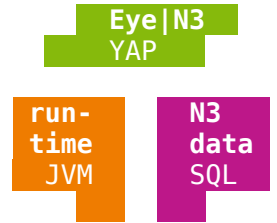
Euler is an inference engine supporting logic based proofs. It is a backward-forward-backward chaining reasoner enhanced with Euler path detection.

The backward-forward-backward chaining is realized via an underlying Prolog backward chaining, a forward meta-level reasoning and a backward proof construction.

The Euler path detection is roughly "don't step in your own steps" to avoid vicious circles so to speak and in that respect there is a similarity with what Leonhard Euler discovered in 1736 for the [Königsberg Bridge Problem](#).

Via [N3](#) the reasoner is interoperable with W3C [Cwm](#).

See also [README](#) and the software is maintained at [EulerSharp](#).



# Skolem Euler Machine (SEM)

The notation that is used is [N3](#) and the logic is [N3Logic](#).  
N3 builtins are described in [CwmBuiltins](#) and [log-rules](#).

Internally SEM is using Prolog Coherent Logic (PCL) intermediate code:

N3 data/proof output



Prolog Coherent Logic (PCL)

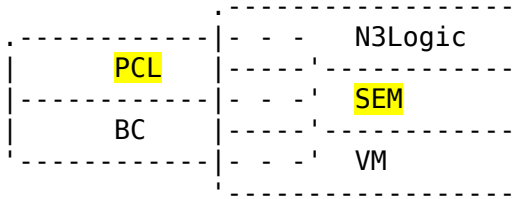


N3 data/rules input

Some [SEM test cases](#) and [results](#)

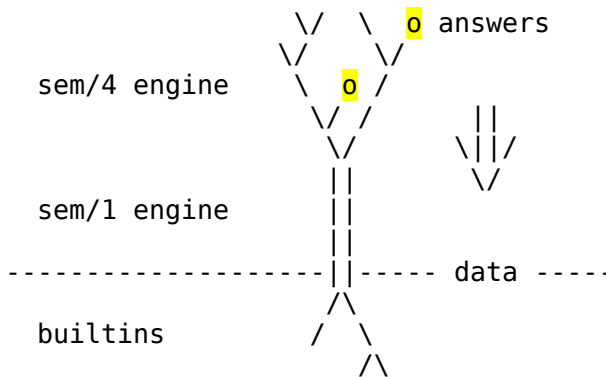
# Euler under the hood

Things are layered and cascaded as follows:



For proof tactics the main idea is to annotate the original N3 formulae instead of tampering with them. That way proof tactics can even be derived on the fly using N3 rules.

Euler is making use of builtins to support the sem/n engines:



# Prolog Coherent Logic (PCL)

N3 formulae are translated in Coherent Logic input clauses of the form

```
[<src>|variables] := <conjunction> => <disjunction>.
```

All the variables used in <conjunction> are universally quantified and all the other variables are existentially quantified.

Variables start with an upper case letter or the '\_' character. The conjunction operator is ',' and the disjunction operator is ';'. Comments start with '%' and end with the next end of line.

Strategies can be declared with the binary predicate

```
'e:tactic'('log:implies'(<conjunction>,<disjunction>),[<guard>,<new_guard>]).
```

The subject of e:tactic is an instance of <conjunction> => <disjunction> and the object of e:tactic is a list containing <guard> and <new\_guard>. For every reasoning step there is a <current\_guard> which is initially the empty list. Input clauses that have an e:tactic are only selected when their <guard> unifies with the <current\_guard> and when the input clause can be satisfied <current\_guard> becomes <new\_guard>.

Proving 'p and q' should be:

```
[<src>] := p, q => goal.
```

Proving 'p or q' should be:

```
[<src>] := p => goal.  
[<src>] := q => goal.
```

Proving 'there exists an x such that p(x)' should be:

```
[<src>] := p(X) => goal.
```

Proving 'for all x with p(x) there exists a y such that q(x,y)' should be:

```
dom(a).  
[<src>] := true => p(a).  
[<src>] := q(a,Y) => goal.
```

Inconsistency is proved without an input clause for goal.

## Using SQL

To cope with large amounts of triples, one can use `euler --sql` to translate triples into SQL and after adding e.g.

```
dbname.driver          = org.sqlite.JDBC
dbname.uri             = jdbc:sqlite:tripleStore/dbname
```

to [codd.properties](#) one can get e.g.

`http://host.domain/dbname?SQL=sql` where `sql` is an urlencoding of e.g.

```
SELECT '@prefix ', prefix, ' ', namespace, '.' FROM pfx;
SELECT '';
SELECT subject, ' ', predicate, ' ', object, '.' FROM rdf
  WHERE predicate == 'rdfs:subClassOf' AND object == ':Event';
```

Similar queries can be used to get triples out of the huge amount of existing relational data. For instance, given a database table `tbl1` with columns `one` and `two` the following query result is a set of RDF/N3 triples

```
SELECT '@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.';
SELECT '@prefix : <http://www.agfa.com/w3c/euler/dtP#>.';
SELECT '';
SELECT '"', one, '" :birthday "', two, '"^^xsd:gYear.' FROM tbl1
  WHERE two == 1956;
```



**End**

[Jos De Roo](#)

`$Id: GUIDE 3241 2009-12-23 19:42:33Z josd $`