SWI-Prolog Semantic Web Library

Jan Wielemaker
University of Amsterdam/VU University Amsterdam
The Netherlands
E-mail: J.Wielemaker@cs.vu.nl

March 28, 2011

Abstract

This document describes a library for dealing with standards from the W3C standard for the *Semantic Web*. Like the standards themselves (RDF, RDFS and OWL) this infrastructure is modular. It consists of Prolog packages for reading, querying and storing semantic web documents as well as XPCE libraries that provide visualisation and editing. The Prolog libraries can be used without the XPCE GUI modules. The library has been actively used with upto 10 million triples, using approximately 1GB of memory. Its scalability is limited by memory only. The library can be used both on 32-bit and 64-bit platforms.

Contents

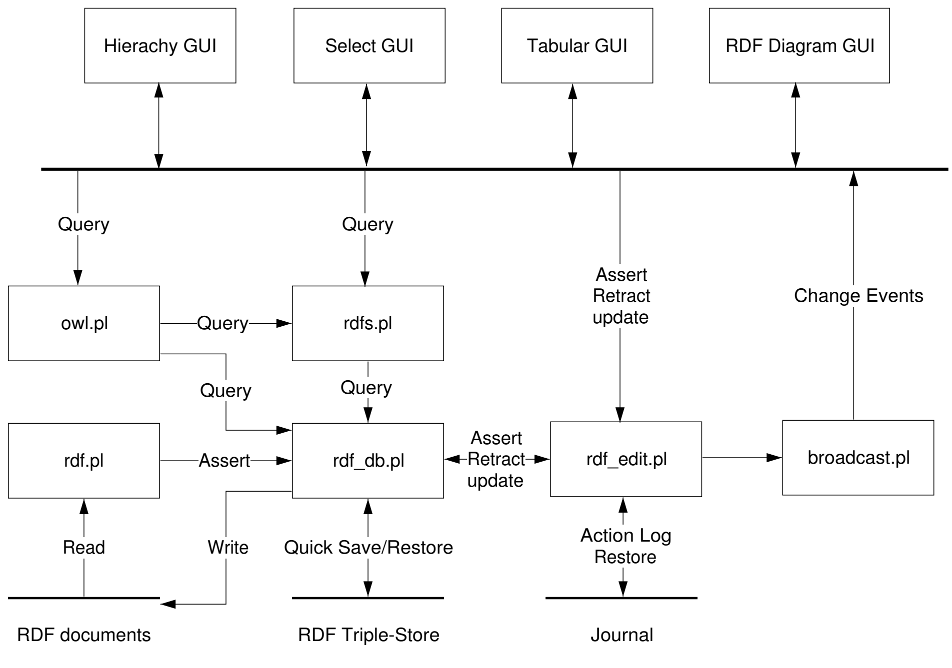

Figure 1: Modules for the Semantic Web library

1 Introduction

SWI-Prolog has started support for web-documents with the development of a small and fast SGML/XML parser, followed by an RDF parser (early 2000). With the `semweb` library we provide more high level support for manipulating semantic web documents. The semantic web is the likely point of orientation for knowledge representation in the future, making a library designed in its spirit promising.

2 Provided libraries

Central to this library is the module `semweb/rdf_db.pl`, providing storage and basic querying for RDF triples. This triple store is filled using the RDF parser realised by `rdf.pl`. The storage module can quickly save and load (partial) databases. The modules `semweb/rdfs.pl` and `semweb/owl.pl` add querying in terms of the more powerful RDFS and OWL languages. Module `semweb/rdf_edit.pl` adds editing, undo, journaling and change-forwarding. Finally, a variety of XPCE modules visualise and edit the database. Figure figure 1 summarised the modular design.

3 library(semweb/rdf_db): The RDF database

The central module is called `rdf_db`. It provides storage and indexed querying of RDF triples. Triples are stored as a quintuple. The first three elements denote the RDF triple. *File* and *Line* provide information about the origin of the triple.

$\{$*Subject Predicate Object File Line*$\}$

The actual storage is provided by the *foreign language (C)* module `rdf_db.c`. Using a dedicated C-based implementation we can reduced memory usage and improve indexing capabilities.[1] Currently the following indexing is provided.

- Any of the 3 fields of the triple
- *Subject* + *Predicate* and *Predicate* + *Object*
- *Predicates* are indexed on the *highest property*. In other words, if predicates are related through `subPropertyOf` predicates indexing happens on the most abstract predicate. This makes calls to `rdf_has/4` very efficient.
- String literal *Objects* are indexed case-insensitive to make case-insensitive queries fully indexed. See `rdf/3`.

3.1 Query the RDF database

rdf(*?Subject, ?Predicate, ?Object*)

Elementary query for triples. *Subject* and *Predicate* are atoms representing the fully qualified URL of the resource. *Object* is either an atom representing a resource or `literal`(*Value*) if the object is a literal value. If a value of the form *NameSpaceID* : *LocalName* is provided it is expanded to a ground atom using `expand_goal/2`. This implies you can use this construct in compiled code without paying a performance penalty. See also section 3.5. Literal values take one of the following forms:

Atom

If the value is a simple atom it is the textual representation of a string literal without explicit type or language (`xml:lang`) qualifier.

lang(*LangID, Atom*)

Atom represents the text of a string literal qualified with the given language.

type(*TypeID, Value*)

Used for attributes qualified using the `rdf:datatype` *TypeID*. The *Value* is either the textual representation or a natural Prolog representation. See the option `convert_typed_literal`(*:Convertor*) of the parser. The storage layer provides efficient handling of atoms, integers (64-bit) and floats (native C-doubles). All other data is represented as a Prolog record.

For string querying purposes, *Object* can be of the form `literal`(*+Query, -Value*), where *Query* is one of the terms below. Details of literal matching and indexing are described in section 3.1.1.

plain(*+Text*)

Perform exact match **and** demand the language or type qualifiers to match. This query is fully indexed.[2]

[1] The orginal implementation was in Prolog. This version was implemented in 3 hours, where the C-based implementation costed a full week. The C-based implementation requires about half the memory and provides about twice the performance.

[2] This should have been the default when using literal with one argument because it is logically consisent (i.e., (rdf(S,P,literal(X)), X == hello) would have been the same as rdf(S,P,literal(hello). In addition, this is consistent with SPARQL literal identity definition.

exact(+*Text*)
Perform exact, but case-insensitive match. This query is fully indexed.

substring(+*Text*)
Match any literal that contains *Text* as a case-insensitive substring. The query is not indexed on *Object*.

word(+*Text*)
Match any literal that contains *Text* delimited by a non alpha-numeric character, the start or end of the string. The query is not indexed on *Object*.

prefix(+*Text*)
Match any literal that starts with *Text*. This call is intended for *completion*. The query is indexed using the binary tree of literals. See section 3.1.1 for details.

ge(+*Literal*)
Match any literal that is equal or larger then *Literal* in the ordered set of literals.

le(+*Literal*)
Match any literal that is equal or smaller then *Literal* in the ordered set of literals.

between(+*Literal1*, +*Literal2*)
Match any literal that is between *Literal1* and *Literal2* in the ordered set of literals. This may include both *Literal1* and *Literal2*.

like(+*Pattern*)
Match any literal that matches *Pattern* case insensitively, where the '*' character in *Pattern* matches zero or more characters.

Backtracking never returns duplicate triples. Duplicates can be retrieved using `rdf/4`. The predicate `rdf/3` raises a type-error if called with improper arguments. If `rdf/3` is called with a term `literal(_)` as *Subject* or *Predicate* object it fails silently. This allows for graph matching goals like `rdf(S,P,O),rdf(O,P2,O2)` to proceed without errors.[3]

rdf(*?Subject, ?Predicate, ?Object, ?Source*)
As `rdf/3` but in addition return the source-location of the triple. The source is either a plain atom or a term of the format *Atom* : *Integer* where *Atom* is intended to be used as filename or URL and *Integer* for representing the line-number. Unlike `rdf/3`, this predicate does not remove duplicates from the result set.

rdf_has(*?Subject, ?Predicate, ?Object, -TriplePred*)
This query exploits the RDFS `subPropertyOf` relation. It returns any triple whose stored predicate equals *Predicate* or can reach this by following the recursive *subPropertyOf* relation. The actual stored predicate is returned in *TriplePred*. The example below gets all subclasses of an RDFS (or OWL) class, even if the relation used is not `rdfs:subClassOf`, but a user-defined sub-property thereof.[4]

```
subclasses(Class, SubClasses) :-
        findall(S, rdf_has(S, rdfs:subClassOf, Class), SubClasses).
```

[3]Discussion in the SPARQL community votes for allowing literal values as subject. Although we have no principal objections, we fear such an extension will promote poor modelling practice.

[4]This predicate realises semantics defined in RDF-Schema rather than RDF. It is part of the `rdf_db` module because the indexing of this module incorporates the `rdfs:subClassOf` predicate.

In addition, `rdf_has/4` handles the predicate property (see `rdf_predicate_property/2`) `symetric`(*true*) and `inverse_of`(*P2*).

Note that `rdf_has/4` and `rdf_has/3` can return duplicate answers if they use a different *TriplePred*.

rdf_has(*?Subject, ?Predicate, ?Object*)
Same as `rdf_has`(*Subject, Predicate, Object, _*).

rdf_reachable(*?Subject, +Predicate, ?Object*)
Is true if *Object* can be reached from *Subject* following the transitive predicate *Predicate* or a sub-property thereof, while repecting the `symetric`(*true*) or `inverse_of`(*P2*) properties.

If used with either *Subject* or *Object* unbound, it first returns the origin, followed by the reachable nodes in breath-first search-order. The implementation internally looks one solution ahead and succeeds deterministically on the last solution. This predicate never generates the same node twice and is robust against cycles in the transitive relation.

With all arguments instantiated, it succeeds deterministically if a path can be found from *Subject* to *Object*. Searching starts at *Subject*, assuming the branching factor is normally lower. A call with both *Subject* and *Object* unbound raises an instantiation error.[5] The following example generates all subclasses of `rdfs:Resource`:

```
?- rdf_reachable(X, rdfs:subClassOf, rdfs:'Resource').
X = 'http://www.w3.org/2000/01/rdf-schema#Resource' ;
X = 'http://www.w3.org/2000/01/rdf-schema#Class' ;
X = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Property' ;
...
```

rdf_reachable(*?Subject, +Predicate, ?Object, +MaxD, -D*)
Same as `rdf_reachable/3`, but in addition, *MaxD* limits the number of relations expanded and *D* is unified with the 'distance' between *Subject* and *Object*. Distance 0 means *Subject* and *Object* are the same resource. *MaxD* can be the constant `infinite` to impose no distance-limit.

rdf_subject(*?Subject*)
Enumerate resources appearing as a subject in a triple. The main reason for this predicate is to generate the known subjects *without duplicates* as one gets using `rdf`(*Subject, _, _*).

rdf_current_literal(*-Literal*)
Enumerate all known literals. Like `rdf_subject/1`, the motivation is to provide access to literals without generation duplicates. Otherwise the call is the same as `rdf`(*_,_,literal(Literal)*).

3.1.1 Literal matching and indexing

Starting with version 2.5.0 of this library, literal values are ordered and indexed using a balanced binary tree (AVL tree). The aim of this index is threefold.

[5] Should be using bi-directional search.

- Unlike hash-tables, binary trees allow for efficient *prefix* matching. Prefix matching is very useful in interactive applications to provide feedback while typing such as auto-completion.
- Having a table of unique literals we generate creation and destruction events (see `rdf_monitor/2`). These events can be used to maintain additional indexing on literals, such as 'by word'.
- A binary table allow for fast interval matching on typed numeric literals.[6]

As string literal matching is most frequently used for searching purposes, the match is executed case-insensitive and after removal of diacritics. Case matching and diacritics removal is based on Unicode character properties and independent from the current locale. Case conversion is based on the 'simple uppercase mapping' defined by Unicode and diacritic removal on the 'decomposition type'. The approach is lightweight, but somewhat simpleminded for some languages. The tables are generated for Unicode characters upto 0x7fff. For more information, please check the source-code of the mapping-table generator `unicode_map.pl` available in the sources of this package.

Currently the total order of literals is first based on the type of literal using the ordering

$$numeric < string < term$$

Numeric values (integer and float) are ordered by value, integers preceed floats if they represent the same value. strings are sorted alphabetically after case-mapping and diacritic removal as described above. If they match equal, uppercase preceeds lowercase and diacritics are ordered on their unicode value. If they still compare equal literals without any qualifier preceeds literals with a type qualifier which preceeds literals with a language qualifier. Same qualifiers (both type or both language) are sorted alphabetically.[7]

The ordered tree is used for indexed execution of `literal(prefix`*(Prefix), Literal*) as well as `literal(like`*(Like), Literal*) if *Like* does not start with a '*'. Note that results of queries that use the tree index are returned in alphabetical order.

3.2 Predicate properties

The predicates below form an experimental interface to provide more reasoning inside the kernel of the rdb_db engine. Note that `symetric`, `inverse_of` and `transitive` are not yet supported by the rest of the engine.

rdf_current_predicate(*?Predicate***)**

Enumerate all predicates that are used in at least one triple. Behaves as the code below, but much more efficient.

```
rdf_current_predicate(Predicate) :-
        findall(P, rdf(_,P,_), Ps),
        sort(Ps, S),
        member(Predicate, S).
```

[6] Not yet implemented

[7] The ordering defined above may change in future versions to deal with new queries for literals.

Note that there is no relation to defined RDF properties. Properties that have no triples are not reported by this predicate, while predicates that are involved in triples do not need to be defined as an instance of rdf:Property.

rdf_set_predicate(+*Predicate*, +*Property***)**
: Define a property of the predicate. This predicate currently supports the properties `symmetric`, `inverse_of` and `transitive` as defined with `rdf_predicate_property/2`. Adding an A inverse_of B also adds B inverse_of A. An inverse relation is deleted using `inverse_of`(*[]*). ‘

rdf_predicate_property(*?Predicate, -Property***)**
: Query properties of a defined predicate. Currently defined properties are given below.

symmetric(*Bool***)**
: True if the predicate is defined to be symetric. I.e., {A} P {B} implies {B} P {A}. Setting symmetric is equivalent to `inverse_of`(*Self*).

inverse_of(*Inverse***)**
: True if this predicate is the inverse of *Inverse*. This property is used by `rdf_has/3`, `rdf_has/4`, `rdf_reachable/3` and `rdf_reachable/5`.

transitive(*Bool***)**
: True if this predicate is transitive. This predicate is currently not used. It might be used to make `rdf_has/3` imply `rdf_reachable/3` for transitive predicates.

triples(*Triples***)**
: Unify *Triples* with the number of existing triples using this predicate as second argument. Reporting the number of triples is intended to support query optimization.

rdf_subject_branch_factor(-*Float***)**
: Unify *Float* with the average number of triples associated with each unique value for the subject-side of this relation. If there are no triples the value 0.0 is returned. This value is cached with the predicate and recomputed only after substantial changes to the triple set associated to this relation. This property is indented for path optimalisation when solving conjunctions of `rdf/3` goals.

rdf_object_branch_factor(-*Float***)**
: Unify *Float* with the average number of triples associated with each unique value for the object-side of this relation. In addition to the comments with the subject_branch_factor property, uniqueness of the object value is computed from the hash key rather than the actual values.

rdfs_subject_branch_factor(-*Float***)**
: Same as `rdf_subject_branch_factor/1`, but also considering triples of ‘subPropertyOf’ this relation. See also `rdf_has/3`.

rdfs_object_branch_factor(-*Float***)**
: Same as `rdf_object_branch_factor/1`, but also considering triples of ‘subPropertyOf’ this relation. See also `rdf_has/3`.

3.3 Modifying the database

As depicted in figure 1, there are two levels of modification. The `rdf_db` module simply modifies, where the `rdf_edit` library provides transactions and undo on top of this. Applications that wish to use the `rdf_edit` layer must *never* use the predicates from this section directly.

3.3.1 Modifying predicates

rdf_assert(+*Subject, +Predicate, +Object***)**

Assert a new triple into the database. This is equivalent to `rdf_assert/4` using *SourceRef* `user`. *Subject* and *Predicate* are resources. *Object* is either a resource or a term `literal`(*Value*). See `rdf/3` for an explanation of *Value* for typed and language qualified literals. All arguments are subject to name-space expansion (see section 3.5).

rdf_assert(+*Subject, +Predicate, +Object, +SourceRef***)**

As `rdf_assert/3`, adding *SourceRef* to specify the orgin of the triple. *SourceRef* is either an atom or a term of the format *Atom*:*Int* where *Atom* normally refers to a filename and *Int* to the line-number where the description starts.

rdf_retractall(?*Subject, ?Predicate, ?Object***)**

Removes all matching triples from the database. Previous Prolog implementations also provided a backtracking `rdf_retract/3`, but this proved to be rarely used and could always be replaced with `rdf_retractall/3`. As `rdf_retractall/4` using an unbound *SourceRef*.

rdf_retractall(?*Subject, ?Predicate, ?Object, ?SourceRef***)**

As `rdf_retractall/4`, also matching on the *SourceRef*. This is particulary useful to update all triples coming from a loaded file.

rdf_update(+*Subject, +Predicate, +Object, +Action***)**

Replaces one of the three fields on the matching triples depending on *Action*:

subject(*Resource***)**
Changes the first field of the triple.

predicate(*Resource***)**
Changes the second field of the triple.

object(*Object***)**
Changes the last field of the triple to the given resource or `literal`(*Value*).

graph(*Graph***)**
Moves the triple from its current named graph to *Graph*. Note that updating the source has no consequences for the semantics and therefore the *generation* (see `rdf_generation/1`) is *not* updated.

rdf_update(+*Subject, +Predicate, +Object, +Graph, +Action***)**

As `rdf_update/4` but allows for specifying the graph.

3.3.2 Transactions

The predicates from section 3.3.1 perform immediate and atomic modifications to the database. There are two cases where this is not desirable:

1. If the database is modified using information based on reading the same database. A typical case is a forward reasoner examining the database and asserting new triples that can be deduced from the already existing ones. For example, *if* $length(X) > 2$ *then size(X) is large*:

```
        (   rdf(X, length, literal(L)),
            atom_number(L, IL),
            IL > 2,
            rdf_assert(X, size, large),
            fail
        ;   true
        ).
```

   Running this code without precautions causes an error because `rdf_assert/3` tries to get a write lock on the database which has an a read operation (rdf/3 has choicepoints) in progress.

2. Multi-threaded access making multiple changes to the database that must be handled as a unit.

Where the second case is probably obvious, the first case is less so. The storage layer may require reindexing after adding or deleting triples. Such reindexing operatations however are not possible while there are active read operations in other threads or from choicepoints that can be in the same thread. For this reason we added `rdf_transaction/2`. Note that, like the predicates from section 3.3.1, `rdf_transaction/2` raises a permission error exception if the calling thread has active choicepoints on the database. The problem is illustrated below. The `rdf/3` call leaves a choicepoint and as the read lock originates from the calling thread itself the system will deadlock if it would not generate an exception.

```
1 ?- rdf_assert(a,b,c).

Yes
2 ?- rdf_assert(a,b,d).

Yes
3 ?- rdf(a,b,X), rdf_transaction(rdf_assert(a,b,e)).
ERROR: No permission to write rdf_db `default' (Operation would deadlock)
^  Exception: (8) rdf_db:rdf_transaction(rdf_assert(a, b, e)) ? no debug
4 ?-
```

rdf_transaction(*:Goal*)
Same as `rdf_transaction`(*Goal*, `user`).

rdf_transaction(*:Goal, +Id*)
After starting a transaction, all predicates from section 3.3.1 append their operation to the *transaction* instead of modifying the database. If *Goal* succeeds rdf_transaction cuts all choicepoints in *Goal* and executes all recorded operations. If *Goal* fails or throws an exception, all recorded operations are discarded and `rdf_transaction/1` fails or re-throws the exception.

On entry, `rdf_transaction/1` gains exclusive access to the database, but does allow readers to come in from all threads. After the successful completion of *Goal* `rdf_transaction/1` gains completely exclusive access while performing the database updates.

Transactions may be nested. Committing a nested transactions merges its change records into the outer transaction, while discarding a nested transaction simply destroys the change records belonging to the nested transaction.

The *Id* argument may be used to identify the transaction. It is passed to the begin/end events posted to hooks registered with `rdf_monitor/2`. The *Id* `log`(*Term*) can be used to enrich the journal files with additional history context. See section 4.5.1.

rdf_active_transaction(*?Id*)
: True if *Id* is the identifier of a currently active transaction (i.e. `rdf_active_transaction/1` is called from `rdf_transaction/2` with matching *Id*). Note that transaction identifier is not copied and therefore need not be ground and can be further instantiated during the transaction. *Id* is first unified with the innermost transaction and backtracking with the identifier of other active transaction. Fails if there is no matching transaction active, which includes the case where there is no transaction in progress.

3.4 Loading and saving to file

The `rdf_db` module can read and write RDF-XML for import and export as well as a binary format built for quick load and save described in section 3.4.3. Here are the predicates for portable RDF load and save.

rdf_load(*+InOrList*)
: Load triples from *In*, which is either a stream opened for reading, an atom specifying a filename, a URL or a list of valid inputs. This predicate calls `process_rdf/3` to read the source one description at a time, avoiding limits to the size of the input. By default, this predicate provides for caching the results for quick-load using `rdf_load_db/1` described below. Caching strategy and options are description in section 3.4.1.

rdf_load(*+FileOrList, +Options*)
: As `rdf_load/1`, providing additional options. The options are handed to the RDF parser and implemented by `process_rdf/3`. In addition, the following options are provided:

cache(*+Bool*)
: If `true` (default), try to use cached data or create a cache file. Otherwise load the source.

db(*+Graph*)
: Deprecated. New code should use the `graph`(*+Graph*) option.

format(*+Format*)
: Specify the source format explicitly. Normally this is deduced from the filename extension or the mime-type. The core library understands the formats `xml` (RDF/XML) and `triples` (internal quick load and cache format).

graph(*+Graph*)
: Load the data in the given named graph. The default is the URL of the source.

if(+*Condition*)
Condition under which to load the source. *Condition* is the same as for the Prolog `load_files/2` predicate: `changed` (default) load the source if it was not loaded before or has changed; `true` (re-)loads the source unconditionally and `not_loaded` loads the source if it was not loaded, but does not check for modifications.

silent(+*Bool*)
If *Bool* is `true`, the message reporting completion is printed using level `silent`. Otherwise the level is `informational`. See also `print_message/2`.

register_namespaces(+*Bool*)
If `true` (default `false`), register `xmlns:ns=url` namespace declarations as rdf_db:ns(ns,url) namespaces if there is no conflict.

rdf_unload(+*Spec*)
Remove all triples loaded from *Spec*. *Spec* is either a graph name or a source specificatipn. If *Spec* does not refer to a loaded database the predicate succeeds silently.

rdf_save(+*File*)
Save all known triples to the given *File*. Same as `rdf_save`(*File, []*).

rdf_save(+*File*, +*Options*)
Save with options. Provided options are:

graph(+*URI*)
Save all triples that belong to the named-graph *URI*. Saving arbitrary selections is possible using predicates from section 3.4.2.

db(+*FileRef*)
Deprecated synonym for `graph`(*URI*).

anon(+*Bool*)
if `anon`(*false*) is provided anonymous resources are only saved if the resource appears in the object field of another triple that is saved.

base_uri(+*BaseURI*)
If provided, emit `xml:base`="*BaseURI*" in the header and emit all URIs that are relative to the base-uri. The `xml:base` declaration can be suppressed using the option `write_xml_base`(*false*)

write_xml_base(+*Bool*)
If `false` (default `true`), do *not* emit the `xml:base` declaration from the given `base_uri` option. The idea behind this option is to be able to create documents with URIs relative to the document itself:

```
        ...,
        rdf_save(File,
                 [ base_uri(BaseURI),
                   write_xml_base(false)
                 ]),
        ...
```

convert_typed_literal(*:Converter*)
: If present, raw literal values are first passed to *Converter* to apply the reverse of the `convert_typed_literal` option of the RDF parser. The *Converter* is called with the same arguments as in the RDF parser, but now with the last argument instantiated and the first two unbound. A proper convertor that can be used for both loading and saving must be a logical predicate.

encoding(*+Encoding*)
: Define the XML encoding used for the file. Defined values are `utf8` (default), `iso_latin_1` and `ascii`. Using `iso_latin_1` or `ascii`, characters not covered by the encoding are emitted as XML character entities (`&#...;`).

document_language(*+XMLLang*)
: The value *XMLLang* is used for the `xml:lang` attribute in the outermost `rdf:RDF` element. This language acts as a default, which implies that the `xml:lang` tag is only used for literals with a *different* language identifier. Please note that this option will cause all literals without language tag to be interpreted using *XMLLang*.

namespaces(*+List*)
: Explicitely specify saved namespace declarations. See `rdf_save_header/2` option namespaces for details.

rdf_graph(*?DB*)
: True if *DB* is the name of a graph with at least one triple.

rdf_graph_property(*T*)
: rue when Property is a property of Graph. Defined properties are:

hash(*Hash*)
: Hash is the (MD5-)hash for the content of Graph. See section 3.4.4 for details.

source(*URL*)
: The graph is loaded from the Source.

source_last_modified(*TimeStamp*)
: Time is the last-modified timestamp of Source at the moment that the graph was loaded from Source.

triples(*Count*)
: True when Count is the number of triples in Graph.

rdf_source(*?DB*)
: Deprecated. Use `rdf_graph/1` or `rdf_source/2` in new code.

rdf_source(*?DB, ?SourceURL*)
: True if the named graph *DB* was loaded from the source *SourceURL*. A named graph is associated with a *SourceURL* by `rdf_load/2`. The association is stored in the internal binary format, which ensures proper maintenance of the original source through caching and the persistency layer.

rdf_make
: Re-load all RDF sourcefiles (see `rdf_source/1`) that have changed since they were loaded the last time. This implies all triples that originate from the file are removed and the file is

re-loaded. If the file is cached a new cache-file is written. Please note that the new triples are added at the end of the database, possibly changing the order of (conflicting) triples.

3.4.1 library(semweb/rdf_cache): Caching triples

The library `semweb/rdf_cache` defines the caching strategy for triples sources. When using large RDF sources, caching triples greatly speedup loading RDF documents. The cache library implements two caching strategies that are controlled by `rdf_set_cache_options/1`.

Local caching This approach applies to files only. Triples are cached in a sub-directory of the directory holding the source. This directory is called `.cache` (`_cache` on Windows). If the cache option `create_local_directory` is `true`, a cache directory is created if posible.

Global caching This approach applies to all sources, except for unnamed streams. Triples are cached in directory defined by the cache option `global_directory`.

When loading an RDF file, the system scans the configured cache files unless `cache`(*false*) is specified as option to `rdf_load/2` or caching is disabled. If caching is enabled but no cache exists, the system will try to create a cache file. First it will try to do this locally. On failure it will try to configured global cache.

rdf_set_cache_options(+*Options***)**

Set cache options. Defined options are:

enabled(*Bool***)**
: If `true` (default), caching is enabled.

local_directory(*Atom***)**
: Local directory to use for caching. Default `.cache` (Windows: `_cache`).

create_local_directory(*Bool***)**
: If `true` (default `false`), create a local cache directory if none exists and the directory can be created.

global_directory(*Atom***)**
: Global directory to use for caching. The directory is created if the option `create_global_directory` is also given and set to `true`. Sub-directories are created to speedup indexing on filesystems that perform poorly on directories with large numbers of files. Initially not defined.

create_global_directory(*Bool***)**
: If `true` (default `false`), create a global cache directory if none exists.

3.4.2 Partial save

Sometimes it is necessary to make more arbitrary selections of material to be saved or exchange RDF descriptions over an open network link. The predicates in this section provide for this. Character encoding issues are derived from the encoding of the *Stream*, providing support for `utf8`, `iso_latin_1` and `ascii`.

rdf_save_header(+*Stream, +Options***)**
: Save an RDF header, with the XML header, `DOCTYPE`, `ENTITY` and opening the `rdf:RDF` element with appropriate namespace declarations. It uses the primitives from section 3.5 to generate the required namespaces and desired short-name. *Options* is one of:

  graph(+*URI***)**
  : Only search for namespaces used in triples that belong to the given named graph.

  db(+*FileRef***)**
  : Deprecated synonym for `graph`(*FileRef*).

  namespaces(+*List***)**
  : Where *List* is a list of namespace abbreviations (see section 3.5). With this option, the expensive search for all namespaces that may be used by your data is omitted. The namespaces `rdf` and `rdfs` are added to the provided *List*. If a namespace is not declared, the resource is emitted in non-abreviated form.

rdf_save_footer(+*Stream***)**
: Close the work opened with `rdf_save_header/2`.

rdf_save_subject(+*Stream, +Subject, +FileRef***)**
: Save everything known about *Subject* that matches *FileRef*. Using an variable for *FileRef* saves all triples with *Subject*.

rdf_quote_uri(+*URI, -Quoted***)**
: Quote a UNICODE *URI*. First the Unicode is represented as UTF-8 and then the unsafe characters are mapped to be represented as US-ASCII.

3.4.3 Fast loading and saving

Loading and saving RDF format is relatively slow. For this reason we designed a binary format that is more compact, avoids the complications of the RDF parser and avoids repetitive lookup of (URL) identifiers. Especially the speed improvement of about 25 times is worth-while when loading large databases. These predicates are used for caching by `rdf_load/[1,2]` under certain conditions.

rdf_save_db(+*File***)**
: Save all known triples into *File*. The saved version includes the *SourceRef* information.

rdf_save_db(+*File, +FileRef***)**
: Save all triples with *SourceRef FileRef*, regardless of the line-number. For example, using `user` all information added using `rdf_assert/3` is stored in the database.

rdf_load_db(+*File***)**
: Load triples from *File*.

3.4.4 MD5 digests

The `rdf_db` library provides for *MD5 digests*. An MD5 digest is a 128 bit long hash key computed from the triples based on the RFC-1321 standard. MD5 keys are computed for each individual triple and added together to compute the final key, resulting in a key that describes the triple-set but is independant from the order in which the triples appear. It is claimed that it is practically impossible for two different datasets to generate the same MD5 key. The Triple20 editor uses the MD5 key for detecting whether the triples associated to a file have changed as well as to maintain a directory with snapshots of versioned ontology files.

rdf_md5(+*Graph*, -*MD5*)
: Return the MD5 digest for all triples in the database associated to *Graph*. The *MD5* digest itself is represented as an atom holding a 32-character hexadecimal string. The library maintains the digest incrementally on `rdf_load/[1,2]`, `rdf_load_db/1`, `rdf_assert/[3,4]` and `rdf_retractall/[3,4]`. Checking whether the digest has changed since the last `rdf_load/[1,2]` call provides a practical means for checking whether the file needs to be saved.

  Deprecated. New code should use `rdf_graph_property`(*Graph, hash(Hash)*).

rdf_atom_md5(+*Text*, +*Times*, -*MD5*)
: Computes the MD5 hash from *Text*, which is an atom, string or list of character codes. *Times* is an integer ≥ 1. When > 0, the MD5 algorithm is repeated *Times* times on the generated hash. This can be used for password encryption algorithms to make generate-and-test loops slow.

  Deprecated. New code should use the `crypt` library provided by the `clib` package.

3.5 Namespace Handling

Prolog code often contains references to constant resources in a known XML namespace. For example, `http://www.w3.org/2000/01/rdf-schema#Class` refers to the most general notion of a class. Readability and maintability concerns require for abstraction here. The dynamic and multifile predicate rdf_db:ns/2 maintains a mapping between short meaningful names and namespace locations very much like the XML `xmlns` construct. The initial mapping contains the namespaces required for the semantic web languages themselves:

```
ns(rdf,     'http://www.w3.org/1999/02/22-rdf-syntax-ns#').
ns(rdfs,    'http://www.w3.org/2000/01/rdf-schema#').
ns(owl,     'http://www.w3.org/2002/7/owl#').
ns(xsd,     'http://www.w3.org/2000/10/XMLSchema#').
ns(dc,      'http://purl.org/dc/elements/1.1/').
ns(dcterms, 'http://purl.org/dc/terms/').
ns(skos,    'http://www.w3.org/2004/02/skos/core#').
ns(eor,     'http://dublincore.org/2000/03/13/eor#').
```

All predicates for the semweb libraries use `goal_expansion/2` rules to make the SWI-Prolog compiler rewrite terms of the form *Id* : *Local* into the fully qualified URL. In addition, the following predicates are supplied:

rdf_equal(*Resource1*, *Resource2*)
: Defined as *Resource1* = *Resource2*. As this predicate is subject to goal-expansion it can be used to obtain or test global URL values to readable values. The following goal unifies *X* with `http://www.w3.org/2000/01/rdf-schema#Class` without more runtime overhead than normal Prolog unification.

```
        rdf_equal(rdfs:'Class', X)
```

rdf_current_ns(*?Alias, ?URI*) *[nondet]*
: Query defined namespace aliases (prefixes).[8]

rdf_register_ns(*+Alias, +URL*)
: Same as `rdf_register_ns`(*Alias, URL, []*).

rdf_register_ns(*+Alias, +URL, +Options*)
: Register *Alias* as a shorthand for *URL*. Note that the registration must be done before loading any files using them as namespace aliases are handled at compiletime through `goal_expansion/2`. If *Alias* already exists the default is to raise a permission error. If the option `force`(*true*) is provided, the alias is silently modified. Rebinding an alias must be done *before* any code is compiled that relies on the alias. If the option `keep`(*true*) is provided the new registration is silently ignored.

rdf_global_id(*?Alias:Local, ?Global*)
: Runtime translation between *Alias* and *Local* and a *Global* URL. Expansion is normally done at compiletime. This predicate is often used to turn a global URL into a more readable term.

rdf_global_object(*?Object, ?NameExpandedObject*)
: As `rdf_global_id/2`, but also expands the type field if the object is of the form `literal(type`*(Type, Value)*). This predicate is used for goal expansion of the object fields in `rdf/3` and similar goals.

rdf_global_term(*+Term0, -Term*)
: Expands all *Alias*:*Local* in *Term0* and return the result in *Term*. Use infrequently for runtime expansion of namespace identifiers.

3.5.1 Namespace handling for custom predicates

If we implement a new predicate based on one of the predicates of the semweb libraries that expands namespaces, namespace expansion is not automatically available to it. Consider the following code computing the number of distinct objects for a certain property on a certain object.

```
cardinality(S, P, C) :-
        (   setof(O, rdf_has(S, P, O), Os)
        ->  length(Os, C)
        ;   C = 0
        ).
```

Now assume we want to write `labels/2` that returns the number of distict labels of a resource:

```
labels(S, C) :-
        cardinality(S, rdfs:label, C).
```

[8]Older versions of this library did not export the table rdf_db:ns/2. Please use this new public interface.

This code will *not work* as `rdfs:label` is not expanded at compile time. To make this work, we need to add an `rdf_meta/1` declaration.

```
:- rdf_meta
        cardinality(r,r,-).
```

rdf_meta(*:Heads*)

This predicate defines the argument types of the named predicates, which will force compile time namespace expansion for these predicates. *Heads* is a coma-separated list of callable terms. Defined argument properties are:

:
: Argument is a goal. The goal is processed using `expand_goal/2`, recursively applying goal transformation on the argument.

+
: The argument is instantiated at entry. Nothing is changed.

-
: The argument is not instantiated at entry. Nothing is changed.

?
: The argument is unbound or instantiated at entry. Nothing is changed.

@
: The argument is not changed.

r
: The argument must be a resource. If it is a term ⟨*namespace*⟩:⟨*local*⟩ it is translated.

o
: The argument is an object or resource.

t
: The argument is a term that must be translated. Expansion will translate all occurences of ⟨*namespace*⟩:⟨*local*⟩ appearing anywhere in the term.

As it is subject to `term_expansion/2`, the `rdf_meta/1` declaration can only be used as a *directive*. The directive must be processed before the definition of the predicates as well as before compiling code that uses the rdf meta-predicates. The atom `rdf_meta` is declared as an operator exported from library `rdf_db.pl`. Files using `rdf_meta/1` *must* explicitely load `rdf_db.pl`. The example below defines the rule `concept/1`.

```
:- use_module(library(semweb/rdf_db)).  % for rdf_meta
:- use_module(library(semweb/rdfs)).    % for rdfs_individual_of

:- rdf_meta
        concept(r).

%%      concept(?C) is nondet.
%
```

```
%       True if C is a concept.

concept(C) :-
        rdfs_individual_of(C, skos:'Concept').
```

In addition to expanding *calls*, `rdf_meta/1` also causes expansion of clause-heads for predicates that match a declaration. This is typically used write Prolog statements about resources. The following example produces three clauses with expanded (single-atom) arguments:

```
:- use_module(library(semweb/rdf_db)).

:- rdf_meta
        label_predicate(r).

label_predicate(rdfs:label).
label_predicate(skos:prefLabel).
label_predicate(skos:altLabel).
```

3.6 Monitoring the database

Considering performance and modularity, we are working on a replacement of the `rdf_edit` (see section 12) layered design to deal with updates, journalling, transactions, etc. Where the rdf_edit approach creates a single layer on top of rdf_db and code using the RDF database must select whether to use rdf_db.pl or rdf_edit.pl, the new approach allows to register *monitors*. This allows multiple modules to provide additional services, while these services will be used regardless of how the database is modified.

Monitors are used by the persistency library (section 4.5) and the literal indexing library (section 4.4).

rdf_monitor(*:Goal, +Mask*)
: *Goal* is called for modifications of the database. It is called with a single argument that describes the modification. Defined events are:

  assert(*+S, +P, +O, +DB*)
  : A triple has been asserted.

  retract(*+S, +P, +O, +DB*)
  : A triple has been deleted.

  update(*+S, +P, +O, +DB, +Action*)
  : A triple has been updated.

  new_literal(*+Literal*)
  : A new literal has been created. *Literal* is the argument of `literal`(*Arg*) of the triple's object. This event is introduced in version 2.5.0 of this library.

  old_literal(*+Literal*)
  : The literal *Literal* is no longer used by any triple.

transaction(*+BeginOrEnd, +Id*)

Mark begin or end of the *commit* of a transaction started by `rdf_transaction/2`. *BeginOrEnd* is `begin`(*Nesting*) or `end`(*Nesting*). *Nesting* expresses the nesting level of transactions, starting at '0' for a toplevel transaction. *Id* is the second argument of `rdf_transaction/2`. The following transaction Ids are pre-defined by the library:

parse(*Id*)

A file is loaded using `rdf_load/2`. *Id* is one of `file`(*Path*) or `stream`(*Stream*).

unload(*DB*)

All triples with source *DB* are being unloaded using `rdf_unload/1`.

reset

Issued by `rdf_reset_db/0`.

load(*+BeginOrEnd, +Spec*)

Mark begin or end of `rdf_load_db/1` or load through `rdf_load/2` from a cached file. *Spec* is currently defined as `file`(*Path*).

rehash(*+BeginOrEnd*)

Marks begin/end of a re-hash due to required re-indexing or garbage collection.

Mask is a list of events this monitor is interested in. Default (empty list) is to report all events. Otherwise each element is of the form +Event or -Event to include or exclude monitoring for certain events. The event-names are the functor names of the events described above. The special name `all` refers to all events and `assert`(*load*) to assert events originating from `rdf_load_db/1`. As loading triples using `rdf_load_db/1` is very fast, monitoring this at the triple level may seriously harm performance.

This predicate is intended to maintain derived data, such as a journal, information for *undo*, additional indexing in literals, etc. There is no way to remove registered monitors. If this is required one should register a monitor that maintains a dynamic list of subscribers like the XPCE broadcast library. A second subscription of the same hook predicate only re-assignes the mask.

The monitor hooks are called in the order of registration and in the same thread that issued the database manipulation. To process all changes in one thread they should be send to a thread message queue. For all updating events, the monitor is called while the calling thread has a write lock on the RDF store. This implies that these events are processed strickly synchronous, even if modifications originate from multiple threads. In particular, the `transaction` *begin*, ... *updates* ... *end* sequence is never interleaved with other events. Same for `load` and `parse`.

3.7 Miscellaneous predicates

This section describes the remaining predicates of the `rdf_db` module.

rdf_node(*-Id*)

Generate a unique reference. The returned atom is guaranteed not to occur in the current database in any field of any triple.

rdf_bnode(*-Id*)

Generate a unique blank node reference. The returned atom is guaranteed not to occur in the current database in any field of any triple and starts with '__bnode'.

rdf_is_bnode(+*Id*)
Succeeds if *Id* is a blank node identifier (also called *anonymous resource*). In the current implementation this implies it is an atom starting with a double underscore.

rdf_is_resource(+*Id*)
Succeeds if *Id* is a resource. Note that this resource need not to appear in any triple.

rdf_is_literal(+*Id*)
Succeeds if *Id* is an RDF literal term. Note that this literal need not to appear in any triple.

rdf_source_location(+*Subject, -SourceRef*)
Return the source-location as *File*:*Line* of the first triple that is about *Subject*.

rdf_generation(-*Generation*)
Returns the *Generation* of the database. Each modification to the database increments the generation. It can be used to check the validity of cached results deduced from the database. Modifications changing multiple triples increment *Generation* with the number of triples modified, providing a heuristic for 'how dirty' cached results may be.

rdf_estimate_complexity(*?Subject, ?Predicate, ?Object, -Complexity*)
Return the number of alternatives as indicated by the database internal hashed indexing. This is a rough measure for the number of alternatives we can expect for an `rdf_has/3` call using the given three arguments. When called with three variables, the total number of triples is returned. This estimate is used in query optimisation. See also `rdf_predicate_property/2` and `rdf_statistics/1` for additional information to help optimisers.

rdf_statistics(*?Statistics*)
Report statistics collected by the `rdf_db` module. Defined values for *Statistics* are:

lookup(*?Index, -Count*)
Number of lookups using a pattern of instantiated fields. *Index* is a term `rdf`(*S,P,O*), where *S*, *P* and *O* are either + or −. For example `rdf`(+,+,-) returns the lookups with subject and predicate specified and object unbound.

properties(-*Count*)
Number of unique values for the second field of the triple set.

sources(-*Count*)
Number of files loaded through `rdf_load/1`.

subjects(-*Count*)
Number of unique values for the first field of the triple set.

literals(-*Count*)
Total number of unique literal values in the database. See also section 3.1.1.

triples(-*Count*)
Total number of triples in the database.

triples_by_file(*?File, -Count*)
Enumerate the number of triples associated to each file.

searched_nodes(-*Count*)
Number of nodes explored in `rdf_reachable/3`.

gc(-*Count, -Time***)**
: Number of garbage collections and time spent in seconds represented as a float.

rehash(-*Count, -Time***)**
: Number of times the hash-tables were enlarged and time spent in seconds represented as a float.

core(-*Bytes***)**
: Core used by the triple store. This includes all memory allocated on behalf of the library, but *not* the memory allocated in Prolog atoms referenced (only) by the triple store.

rdf_match_label(+*Method,* +*Search,* +*Atom***)**
True if *Search* matches *Atom* as defined by *Method*. All matching is performed case-insensitive. Defines methods are:

exact
: Perform exact, but case-insensitive match.

substring
: *Search* is a sub-string of *Text*.

word
: *Search* appears as a whole-word in *Text*.

prefix
: *Text* start with *Search*.

like
: *Text* matches *Search*, case insensitively, where the '*' character in *Search* matches zero or more characters.

lang_matches(+*Lang,* +*Pattern***)**
True if *Lang* matches *Pattern*. This implements XML language matching conform RFC 4647. Both *Lang* and *Pattern* are dash-separated strings of identifiers or (for *Pattern*) the wildcart $\star$. Identifiers are matched case-insensitive and a $\star$ matches any number of identifiers. A short pattern is the same as $\star$.

lang_equal(+*Lang1,* +*Lang2***)**
True if *Lang1* and *Lang2* specify the same language, including regional and other modifiers. Language-specifiers are case-insensitive.

rdf_reset_db
Erase all triples from the database and reset all counts and statistics information.

rdf_version(-*Version***)**
Unify *Version* with the library version number. This number is, like to the SWI-Prolog version flag, defined as $10,000 \times Major + 100 \times Minor + Patch$.

3.8 Issues with rdf_db

This RDF low-level module has been created after two year experimenting with a plain Prolog based module and a brief evaluation of a second generation pure Prolog implementation. The aim was to be able to handle upto about 5 million triples on standard (notebook) hardware and deal efficiently with `subPropertyOf` which was identified as a crucial feature of RDFS to realise fusion of different data-sets.

The following issues are identified and not solved in suitable manner.

`subPropertyOf` of `subPropertyOf` is not supported.

Equivalence Similar to `subPropertyOf`, it is likely to be profitable to handle resource identity efficient. The current system has no support for it.

4 Plugin modules for rdf_db

The `rdf_db` module provides several hooks for extending its functionality. Database updates can be monitored and acted upon through the features described in section 3.6. The predicate `rdf_load/2` can be hooked to deal with different formats such as *rdfturtle*, different input sources (e.g. http) and different strategies for caching results.

4.1 Hooks into the RDF library

The hooks below are used to add new RDF file formats and sources from which to load data to the library. They are used by the modules described below and distributed with the package. Please examine the source-code if you want to add new formats or locations.

`rdf_turtle.pl` Load files in the Turtle format. See section 5.

`rdf_zlib_plugin.pl` Load `gzip` compressed files transparently. See section 4.2.

`rdf_http_plugin.pl` Load RDF documents from HTTP servers. See section 4.3.

rdf_db:rdf_open_hook(*+Input, -Stream, -Format*)
: Open an input. *Input* is one of `file`(*+Name*), `stream`(*+Stream*) or `url`(*Protocol, URL*). If this hook succeeds, the RDF will be read from Stream using `rdf_load_stream/3`. Otherwise the default open functionality for file and stream are used.

rdf_db:rdf_load_stream(*+Format, +Stream, +Options*)
: Actually load the RDF from *Stream* into the RDF database. *Format* describes the format and is produced either by `rdf_input_info/3` or `rdf_file_type/2`.

rdf_db:rdf_input_info(*+Input, -Modified, -Format*)
: Gather information on *Input*. *Modified* is the last modification time of the source as a POSIX time-stamp (see `time_file/2`). *Format* is the RDF format of the file. See `rdf_file_type/2` for details. It is allowed to leave the output variables unbound. Ultimately the default modified time is '0' and the format is assumed to be `xml`.

rdf_db:rdf_file_type(*?Extension, ?Format*)
: True if *Format* is the default RDF file format for files with the given extension. *Extension* is lowercase and without a '.'. E.g. `owl`. *Format* is either a built-in format (`xml` or `triples`) or a format understood by the `rdf_load_stream/3` hook.

rdf_db:url_protocol(*?Protocol*)
: True if *Protocol* is a URL protocol recognised by `rdf_load/2`.

4.2 library(semweb/rdf_zlib_plugin): Reading compressed RDF

This module uses the `zlib` library to load compressed files on the fly. The extension of the file must be `.gz`. The file format is deduced by the extension after stripping the `.gz` extension. E.g. `rdf_load('file.rdf.gz')`.

4.3 library(semweb/rdf_http_plugin): Reading RDF from a HTTP server

This module allows for `rdf_load('http://...')`. It exploits the library `http/http_open.pl`. The format of the URL is determined from the mime-type returned by the server if this is one of `text/rdf+xml`, `application/x-turtle` or `application/turtle`. As RDF mime-types are not yet widely supported, the plugin uses the extension of the URL if the claimed mime-type is not one of the above. In addition, it recognises `text/html` and `application/xhtml+xml`, scanning the XML content for embedded RDF.

4.4 library(semweb/rdf_litindex): Indexing words in literals

The library `semweb/rdf_litindex.pl` exploits the primitives of section 4.4.1 and the NLP package to provide indexing on words inside literal constants. It also allows for fuzzy matching using stemming and 'sounds-like' based on the *double metaphone* algorithm of the NLP package.

rdf_find_literals(*+Spec, -ListOfLiterals*)

Find literals (without type or language specification) that satisfy *Spec*. The required indices are created as needed and kept up-to-date using hooks registered with `rdf_monitor/2`. Numerical indexing is currently limited to integers in the range $\pm 2^{3}0$ ($\pm 2^{6}2 on64 - bitplatforms$). *Spec* is defined as:

and(*Spec1, Spec2*)

Intersection of both specifications.

or(*Spec1, Spec2*)

Union of both specifications.

not(*Spec*)

Negation of *Spec*. After translation of the full specification to *Disjunctive Normal Form* (DNF), negations are only allowed inside a conjunction with at least one positive literal.

case(*Word*)

Matches all literals containing the word *Word*, doing the match case insensitive and after removing diacritics.

stem(*Like*)

Matches all literals containing at least one word that has the same stem as *Like* using the Porter stem algorithm. See NLP package for details.

sounds(*Like*)

Matches all literals containing at least one word that 'sounds like' *Like* using the double metaphone algorithm. See NLP package for details.

prefix(*Prefix*)

Matches all literals containing at least one word that starts with Prefix, discarding diacritics and case.

between(*Low, High*)
: Matches all literals containing an integer token in the range *Low..High*, including the boundaries.

ge(*Low*)
: Matches all literals containing an integer token with value *Low* or higher.

le(*High*)
: Matches all literals containing an integer token with value *High* or lower.

Token
: Matches all literals containing the given token. See `tokenize_atom/2` of the NLP package for details.

rdf_token_expansions(*+Spec, -Expansions*)

Uses the same database as `rdf_find_literals/2` to find possible expansions of *Spec*, i.e. which words 'sound like', 'have prefix', etc. *Spec* is a compound expression as in `rdf_find_literals/2`. *Expansions* is unified to a list of terms `sounds`(*Like, Words*), `stem`(*Like, Words*) or `prefix`(*Prefix, Words*). On compound expressions, only combinations that provide literals are returned. Below is an example after loading the ULAN[9] database and showing all words that sounds like 'rembrandt' and appear together in a literal with the word 'Rijn'. Finding this result from the 228,710 literals contained in ULAN requires 0.54 milliseconds (AMD 1600+).

```
?- rdf_token_expansions(and('Rijn', sounds(rembrandt)), L).

L = [sounds(rembrandt, ['Rambrandt', 'Reimbrant', 'Rembradt',
                        'Rembrand', 'Rembrandt', 'Rembrandtsz',
                        'Rembrant', 'Rembrants', 'Rijmbrand'])]
```

Here is another example, illustrating handling of diacritics:

```
?- rdf_token_expansions(case(cafe), L).

L = [case(cafe, [cafe, café])]
```

rdf_tokenize_literal(*+Literal, -Tokens*)

Tokenize a literal, returning a list of atoms and integers in the range $-1073741824 \ldots 1073741823$. As tokenization is in general domain and task-dependent this predicate first calls the hook `rdf_litindex:tokenization`(*Literal, -Tokens*). On failure it calls `tokenize_atom/2` from the NLP package and deletes the following: atoms of length 1, floats, integers that are out of range and the english words `and`, `an`, `or`, `of`, `on`, `in`, `this` and `the`. Deletion first calls the hook `rdf_litindex:exclude_from_index`(*token, X*). This hook is called as follows:

```
no_index_token(X) :-
        exclude_from_index(token, X), !.
```

[9] Unified List of Artist Names from the Getty Foundation.

```
no_index_token(X) :-
        ...
```

4.4.1 Literal maps: Creating additional indices on literals

'Literal maps' provide a relation between literal values, intended to create additional indexes on literals. The current implementation can only deal with integers and atoms (string literals). A literal map maintains an ordered set of *keys*. The ordering uses the same rules as described in section 3.1.1. Each key is associated with an ordered set of *values*. Literal map objects can be shared between threads, using a locking strategy that allows for multiple concurrent readers.

Typically, this module is used together with `rdf_monitor/2` on the channals `new_literal` and `old_literal` to maintain an index of words that appear in a literal. Further abstraction using Porter stemming or Metaphone can be used to create additional search indices. These can map either directly to the literal values, or indirectly to the plain word-map. The SWI-Prolog NLP package provides complimentary building blocks, such as a tokenizer, Porter stem and Double Metaphone.

rdf_new_literal_map(*-Map*)
: Create a new literal map, returning an opaque handle.

rdf_destroy_literal_map(*+Map*)
: Destroy a literal map. After this call, further use of the *Map* handle is illegal. Additional synchronisation is needed if maps that are shared between threads are destroyed to guarantee the handle is no longer used. In some scenarios `rdf_reset_literal_map/1` provides a safe alternative.

rdf_reset_literal_map(*+Map*)
: Delete all content from the literal map.

rdf_insert_literal_map(*+Map, +Key, +Value*)
: Add a relation between *Key* and *Value* to the map. If this relation already exists no action is performed.

rdf_insert_literal_map(*+Map, +Key, +Value, -KeyCount*)
: As `rdf_insert_literal_map/3`. In addition, if *Key* is a new key in *Map*, unify *KeyCount* with the number of keys in *Map*. This serves two purposes. Derived maps, such as the stem and metaphone maps need to know about new keys and it avoids additional foreign calls for doing the progress in `rdf_litindex.pl`.

rdf_delete_literal_map(*+Map, +Key*)
: Delete *Key* and all associated values from the map. Succeeds always.

rdf_delete_literal_map(*+Map, +Key, +Value*)
: Delete the association between *Key* and *Value* from the map. Succeeds always.

rdf_find_literal_map(*+Map, +KeyList, -ValueList*) *[det]*
: Unify *ValueList* with an ordered set of values associated to all keys from *KeyList*. Each key in *KeyList* is either an atom, an integer or a term `not`(*Key*). If not-terms are provided, there must be at least one positive keywords. The negations are tested after establishing the positive matches.

rdf_keys_in_literal_map(*+Map, +Spec, -Answer*)
Realises various queries on the key-set:

all
Unify *Answer* with an ordered list of all keys.

key(*+Key*)
Succeeds if *Key* is a key in the map and unify *Answer* with the number of values associated with the key. This provides a fast test of existence without fetching the possibly large associated value set as with `rdf_find_literal_map/3`.

prefix(*+Prefix*)
Unify *Answer* with an ordered set of all keys that have the given prefix. See section 3.1 for details on prefix matching. *Prefix* must be an atom. This call is intended for auto-completion in user interfaces.

ge(*+Min*)
Unify *Answer* with all keys that are larger or equal to the integer *Min*.

le(*+Max*)
Unify *Answer* with all keys that are smaller or equal to the integer *Max*.

between(*+Min, +Max*)
Unify *Answer* with all keys between *Min* and *Max* (including).

rdf_statistics_literal_map(*+Map, +Key(-Arg...)*)
Query some statistics of the map. Provides keys are:

size(*-Keys, -Relations*)
Unify *Keys* with the total key-count of the index and *Relation* with the total *Key-Value* count.

4.5 library(semweb/rdf_persistency): Providing persistent storage

The `semweb/rdf_persistency` provides reliable persistent storage for the RDF data. The store uses a directory with files for each source (see `rdf_source/1`) present in the database. Each source is represented by two files, one in binary format (see `rdf_save_db/2`) representing the base state and one represented as Prolog terms representing the changes made since the base state. The latter is called the *journal*.

rdf_attach_db(*+Directory, +Options*)
Attach *Directory* as the persistent database. If *Directory* does not exist it is created. Otherwise all sources defined in the directory are loaded into the RDF database. Loading a source means loading the base state (if any) and replaying the journal (if any). The current implementation does not synchronise triples that are in the store before attaching a database. They are not removed from the database, nor added to the presistent store. Different merging options may be supported through the *Options* argument later. Currently defined options are:

concurrency(*+PosInt*)
Number of threads used to reload databased and journals from the files in *Directory*. Default is the number of physical CPUs determined by the Prolog flag `cpu_count` or 1 (one) on systems where this number is unknown. See also `concurrent/3`.

max_open_journals(+*PosInt*)
The library maintains a pool of open journal files. This option specifies the size of this pool. The default is 10. Raising the option can make sense if many writes occur on many different named graphs. The value can be lowered for scenarios where write operations are very infrequent.

silent(*Boolean*)
If `true`, supress loading messages from `rdf_attach_db/2`.

log_nested_transactions(*Boolean*)
If `true`, nested *log* transactions are added to the journal information. By default (`false`), no log-term is added for nested transactions.

The database is locked against concurrent access using a file `lock` in *Directory*. An attempt to attach to a locked database raises a `permission_error` exception. The error context contains a term `rdf_locked`(*Args*), where args is a list containing `time`(*Stamp*) and `pid`(*PID*). The error can be caught by the application. Otherwise it prints:

```
ERROR: No permission to lock rdf_db '/home/jan/src/pl/packages/semweb/DB'
ERROR: locked at Wed Jun 27 15:37:35 2007 by process id 1748
```

rdf_detach_db
Detaches the persistent store. No triples are removed from the RDF triple store.

rdf_current_db(-*Directory*)
Unify *Directory* with the current database directory. Fails if no persistent database is attached.

rdf_persistency(+*DB, +Bool*)
Change presistency of named database (4th argument of `rdf/4`). By default all databases are presistent. Using `false`, the journal and snapshot for the database are deleted and further changes to triples associated with *DB* are not recorded. If *Bool* is `true` a snapshot is created for the current state and further modifications are monitored. Switching persistency does not affect the triples in the in-memory RDF database.

rdf_flush_journals(+*Options*)
Flush dirty journals. With the option `min_size`(*KB*) only journals larger than *KB* Kbytes are merged with the base state. Flushing a journal takes the following steps, ensuring a stable state can be recovered at any moment.

1. Save the current database in a new file using the extension `.new`.
2. On success, delete the journal
3. On success, atomically move the `.new` file over the base state.

Note that journals are *not* merged automatically for two reasons. First of all, some applications may decide never to merge as the journal contains a complete *changelog* of the database. Second, merging large databases can be slow and the application may wish to schedule such actions at quiet times or scheduled maintenance periods.

4.5.1 Enriching the journals

The above predicates suffice for most applications. The predicates in this section provide access to the journal files and the base state files and are intented to provide additional services, such as reasoning about the journals, loaded files, etc.[10]

Using `rdf_transaction`(*Goal, log(Message)*), we can add additional records to enrich the journal of affected databases with *Term* and some additional bookkeeping information. Such a transaction adds a term `begin`(*Id, Nest, Time, Message*) before the change operations on each affected database and `end`(*Id, Nest, Affected*) after the change operations. Here is an example call and content of the journal file `mydb.jrn`. A full explanation of the terms that appear in the journal is in the description of `rdf_journal_file/2`.

```
?- rdf_transaction(rdf_assert(s,p,o,mydb), log(by(jan))).
```

```
start([time(1183540570)]).
begin(1, 0, 1183540570.36, by(jan)).
assert(s, p, o).
end(1, 0, []).
end([time(1183540578)]).
```

Using `rdf_transaction`(*Goal, log(Message, DB)*), where *DB* is an atom denoting a (possibly empty) named graph, the system guarantees that a non-empty transaction will leave a possibly empty transaction record in DB. This feature assumes named graphs are named after the user making the changes. If a user action does not affect the user's graph, such as deleting a triple from another graph, we still find record of all actions performed by some user in the journal of that user.

rdf_journal_file(*?DB, ?JournalFile*)
: True if *File* is the absolute file name of an existing named graph *DB*. A journal file contains a sequence of Prolog terms of the following format.[11]

  start(*Attributes*)
  : Journal has been opened. Currently *Attributes* contains a term `time`(*Stamp*).

  end(*Attributes*)
  : Journal was closed. Currently *Attributes* contains a term `time`(*Stamp*).

  assert(*Subject, Predicate, Object*)
  : A triple {Subject, Predicate, Object} was added to the database.

  assert(*Subject, Predicate, Object, Line*)
  : A triple {Subject, Predicate, Object} was added to the database with given *Line* context.

  retract(*Subject, Predicate, Object*)
  : A triple {Subject, Predicate, Object} was deleted from the database. Note that an `rdf_retractall/3` call can retract multiple triples. Each of them have a record in the journal. This allows for 'undo'.

[10] A library `rdf_history` is under development exploiting these features supporting wiki style editing of RDF.

[11] Future versions of this library may use an XML based language neutral format.

retract(*Subject, Predicate, Object, Line*)
Same as above, for a triple with associated line info.

update(*Subject, Predicate, Object, Action*)
See `rdf_update/4`.

begin(*Id, Nest, Time, Message*)
Added before the changes in each database affected by a transaction with transaction identifier `log`(*Message*). *Id* is an integer counting the logged transactions to this database. Numbers are increasing and designed for binary search within the journal file. *Nest* is the nesting level, where '0' is a toplevel transaction. *Time* is a time-stamp, currently using float notation with two fractional digits. *Message* is the term provided by the user as argument of the `log`(*Message*) transaction.

end(*Id, Nest, Others*)
Added after the changes in each database affected by a transaction with transaction identifier `log`(*Message*). *Id* and *Nest* match the begin-term. *Others* gives a list of other databases affected by this transaction and the *Id* of these records. The terms in this list have the format *DB*:*Id*.

rdf_db_to_file(*?DB, ?FileBase*)
Convert between *DB* (see `rdf_source/1`) and file base-file used for storing information on this database. The full file is located in the directory described by `rdf_current_db/1` and has the extension `.trp` for the base state and `.jrn` for the journal.

5 library(semweb/rdf_turtle): Turtle: Terse RDF Triple Language

To be done Better error handling

This module implements the Turtle language for representing the RDF triple model as defined by Dave Beckett from the Institute for Learning and Research Technology University of Bristol in the document:

- `http://www.w3.org/TeamSubmission/turtle/`
- `http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/#sec-conformance`

This parser passes all tests, except for test-28.ttl (decial number serialization) and test-29.ttl (uri containing ...%&...). It is unclear to me whether these tests are correct. Notably, it is unclear whether we must do %-decoding. Certainly, this is expected by various real-life datasets that we came accross with.

This module acts as a plugin to `rdf_load/2`, for processing files with one of the extensions `.ttl`, `.n3` or `.nt`.

rdf_read_turtle(*+Input, -Triples, +Options*)
Read a stream or file into a set of triples of the format

rdf(Subject, Predicate, Object)

The representation is consistent with the SWI-Prolog RDF/XML and ntriples parsers. Provided options are:

base_uri(+*BaseURI*)
Initial base URI. Defaults to file://<file> for loading files.

anon_prefix(+*Prefix*)
Blank nodes are generated as <Prefix>1, <Prefix>2, etc. If Prefix is not an atom blank nodes are generated as node(1), node(2), ...

resources(*URIorIRI*)
Officially, Turtle resources are IRIs. Quite a few applications however send URIs. By default we do URI->IRI mapping because this rarely causes errors. To force strictly conforming mode, pass `iri`.

prefixes(-*Pairs*)
Return encountered prefix declarations as a list of Alias-URI

namespaces(-*Pairs*)
Same as prefixes(Pairs). Compatibility to `rdf_load/2`.

base_used(-*Base*)
Base URI used for processing the data. Unified to [] if there is no base-uri.

on_error(+*ErrorMode*)
In `warning` (default), print the error and continue parsing the remainder of the file. If `error`, abort with an exception on the first error encountered.

error_count(-*Count*)
If on_error(warning) is active, this option cane be used to retrieve the number of generated errors.

rdf_load_turtle(+*Input, -Triples, +Options*) **deprecated**
Use `rdf_read_turtle/3`

rdf_process_turtle(+*Input, :OnObject, +Options*) *[det]*
Process Turtle input from *Input*, calling *OnObject* with a list of triples. *Options* is the same as for `rdf_load_turtle/3`.

Errors encountered are sent to `print_message/2`, after which the parser tries to recover and parse the remainder of the data.

6 library(semweb/rdf_turtle_write): Turtle - Terse RDF Triple Language writer

To be done Low-level string output takes 28% of the time. Move to C?

This module implements the Turtle language for representing the RDF triple model as defined by Dave Beckett from the Institute for Learning and Research Technology University of Bristol in the document:

- `http://www.w3.org/TeamSubmission/turtle/`
- `http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/#sec-conformance`

The Turtle format is designed as an RDF serialization that is easy to read and write by both machines and humans. Due to the latter property, this library goes a long way in trying to produce human-readable output.

In addition to the human-readable format, this library can write a *canonical* representation of RDF graphs. The canonical representation has the following properties:

- Equivalent graphs result in the same document. Graphs are considered equivalent iff they contain the same *set* of triples, regardless of the labeling of blank nodes in the graph.
- Changes to the graph are diff-friendly. This means
  - Prefixes are combined in the header and thus changes to the namespaces only result in changes in the header.
  - Blank nodes that are used only once (including collections) are written in-line with the object they belong to.
  - For other blank nodes we to realise stable labeling that is based on property-values.

rdf_save_turtle(+*Out, :Options*) *[det]*

Save an RDF graph as Turtle. *Options* processed are:

align_prefixes(+*Boolean*)
: Nicely align the @prefix declarations

base(+*Base*)
: Save relative to the given Base

canonize_numbers(+*Boolean*)
: If `true` (default `false`), emit numeric datatypes using Prolog's write to achieve canonical output.

comment(+*Boolean*)
: It `true` (default), write some informative comments between the output segments

encoding(+*Encoding*)
: Encoding used for the output stream. Default is UTF-8.

expand(*:Goal*)
: Query an alternative graph-representation. See below.

indent(+*Column*)
: Indentation for ; -lists. '0' does not indent, but writes on the same line. Default is 8.

graph(+*Graph*)
: Save only the named graph

group(+*Boolean*)
: If `true` (default), using P-O and O-grouping.

only_known_prefixes(+*Boolean*)
: Only use prefix notation for known prefixes. Without, some documents produce *huge* amounts of prefixes.

silent(+*Boolean*)
: If `true` (default `false`), do not print the final informational message.

single_line_bnodes(+*Bool*)
: If `true` (default `false`), write [...] and (...) on a single line.

subject_white_lines(+*Count*)
: Extra white lines to insert between statements about a different subject. Default is 1.

tab_distance(+*Tab*)
: Distance between tab-stops. '0' forces the library to use only spaces for layout. Default is 8.

user_prefixes(+*Boolean*)
: If `true` (default), use prefixes from `rdf_current_ns/2`.

The option `expand` allows for serializing alternative graph representations. It is called through `call/5`, where the first argument is the expand-option, followed by S,P,O,G. G is the graph-option (which is by default a variable). This notably allows for writing RDF graphs represented as rdf(S,P,O) using the following code fragment:

```
triple_in(RDF, S,P,O,_G) :-
    member(rdf(S,P,O), RDF).

    ...,
    rdf_save_turtle(Out, [ expand(triple_in(RDF)) ]),
```

Parameters

Out	is one of stream(Stream), a stream handle, a file-URL or an atom that denotes a filename.

rdf_save_canonical_turtle(+*Spec, +Options*) *[det]*

Save triples in a canonical format. This is the same as `rdf_save_turtle/3`, but using different defaults. In particular:

- encoding(utf8),
- indent(0),
- tab_distance(0),
- subject_white_lines(1),
- align_prefixes(false),
- user_prefixes(false)
- comment(false),
- group(false),
- single_line_bnodes(true)

To be done Work in progress. Notably blank-node handling is incomplete.

7 library(semweb/rdfs): RDFS related queries

The `semweb/rdfs` library adds interpretation of the triple store in terms of concepts from RDF-Schema (RDFS). There are two ways to provide support for more high level languages in RDF. One is to view such languages as a set of *entailment rules*. In this model the rdfs library would provide a predicate `rdfs/3` providing the same functionality as `rdf/3` on union of the raw graph and triples that can be derived by applying the RDFS entailment rules.

Alternatively, RDFS provides a view on the RDF store in terms of individuals, classes, properties, etc., and we can provide predicates that query the database with this view in mind. This is the approach taken in the `semweb/rdfs.pl` library, providing calls like `rdfs_individual_of`(*?Resource, ?Class*).[12]

7.1 Hierarchy and class-individual relations

The predicates in this section explore the `rdfs:subPropertyOf`, `rdfs:subClassOf` and `rdf:type` relations. Note that the most fundamental of these, `rdfs:subPropertyOf`, is also used by `rdf_has/[3,4]`.

rdfs_subproperty_of(*?SubProperty, ?Property*)
: True if *SubProperty* is equal to *Property* or *Property* can be reached from *SubProperty* following the `rdfs:subPropertyOf` relation. It can be used to test as well as generate sub-properties or super-properties. Note that the commonly used semantics of this predicate is wired into `rdf_has/[3,4]`.[13].[14]

rdfs_subclass_of(*?SubClass, ?Class*)
: True if *SubClass* is equal to *Class* or *Class* can be reached from *SubClass* following the `rdfs:subClassOf` relation. It can be used to test as well as generate sub-classes or super-classes.[15].

rdfs_class_property(*+Class, ?Property*)
: True if the domain of *Property* includes *Class*. Used to generate all properties that apply to a class.

rdfs_individual_of(*?Resource, ?Class*)
: True if *Resource* is an indivisual of *Class*. This implies *Resource* has an `rdf:type` property that refers to *Class* or a sub-class thereof. Can be used to test, generate classes *Resource* belongs to or generate individuals described by *Class*.

7.2 Collections and Containers

The RDF construct `rdf:parseType=Collection` constructs a list using the `rdf:first` and `rdf:next` relations.

[12]The SeRQL language is based on querying the deductive closure of the triple set. The SWI-Prolog SeRQL library provides *entailment modules* that take the approach outlined above.

[13]BUG: The current implementation cannot deal with cycles

[14]BUG: The current implementation cannot deal with predicates that are an `rdfs:subPropertyOf` of `rdfs:subPropertyOf`, such as `owl:samePropertyAs`.

[15]BUG: The current implementation cannot deal with cycles

rdfs_member(*?Resource, +Set*)
Test or generate the members of *Set*. *Set* is either an individual of `rdf:List` or `rdf:Container`.

rdfs_list_to_prolog_list(*+Set, -List*)
Convert *Set*, which must be an individual of `rdf:List` into a Prolog list of objects.

rdfs_assert_list(*+List, -Resource*)
Equivalent to `rdfs_assert_list/3` using *DB* = `user`.

rdfs_assert_list(*+List, -Resource, +DB*)
If *List* is a list of resources, create an RDF list *Resource* that reflects these resources. *Resource* and the sublist resources are generated with `rdf_bnode/1`. The new triples are associated with the database *DB*.

7.3 Labels and textual search

Textual search is partly handled by the predicates from the `rdf_db` module and its underlying C-library. For example, literal objects are hashed case-insensitive to speed up the commonly used case-insensitive search.

rdfs_label(*?Resource, ?Language, ?Label*) *[multi]*
Extract the label from *Resource* or generate all resources with the given *Label*. The label is either associated using a sub-property of `rdfs:label` or it is extracted from *Resource* by taking the part after the last `#` or `/`. If this too fails, *Label* is unified with *Resource*. *Language* is unified to the value of the `xml:lang` attribute of the label or a variable if the label has no language specified.

rdfs_label(*?Resource, ?Label*)
Defined as `rdfs_label`(*Resource, _, Label*).

rdfs_ns_label(*?Resource, ?Language, ?Label*)
Similar to `rdfs_label/2`, but prefixes the result using the declared namespace alias (see section 3.5) to facilitate user-friendly labels in applications using multiple namespaces that may lead to confusion.

rdfs_ns_label(*?Resource, ?Label*)
Defined as `rdfs_ns_label`(*Resource, _, Label*).

rdfs_find(*+String, +Description, ?Properties, +Method, -Subject*)
Find (on backtracking) *Subject*s that satisfy a search specification for textual attributes. *String* is the string searched for. *Description* is an OWL description (see section 14) specifying candidate resources. *Properties* is a list of properties to search for literal objects, *Method* defines the textual matching algorithm. All textual mapping is performed case-insensitive. The matching-methods are described with `rdf_match_label/3`. If *Properties* is unbound, the search is performed in any property and *Properties* is unified with a list holding the property on which the match was found.

8 Managing RDF input files

Complex projects require RDF resources from many locations and typically wish to load these in different combinations. For example loading a small subset of the data for debugging purposes or load a different set of files for experimentation. The library `semweb/rdf_library.pl` manages sets of RDF files spread over different locations, including file and network locations. RDF files are annotated using a *Manifest* file in RDF format.

Currently (September 2007), the E-culture server loads more than 120 RDF files, containing many different schemas, instance repositories and ontology mappings. Some resources, such as the W3C version of Wordnet come in many files. The server is initialised by loading (a subset of) these files. The subset is defined by predicates called `load_medium/0`, `load_tgn/1`, etc. This has become unmanageable. There is no way to find out exactly what will be loaded or whether all RDF files are in place except for actually executing the load. There is also no easy way to exploit concurrency to speedup the process.

For this reason we introduce RDF *Manifest* files that describe one or more RDF resources and their dependencies. The manifest file can be distributed along with a set of RDF files, providing a machine readable portable and declarative description of how the RDF files are intended to be combined. Software allows for listing the content of the library or loading an entry with all dependencies.

8.1 The Manifest file

A manifest file is an RDF file, often in Turtle [**?**] format, that provides meta-data about RDF resources. Often a manifest will describe RDF files in the current directory, but it can also describe RDF resources at arbitrary URL locations. The RDF schema for RDF library meta-data can be found in `rdf_library.ttl`. The namespace for the RDF library format is defined as `http://www.swi-prolog.org/rdf/library/` and abbreviated as `lib`.

The schema defines three root classes: lib:Namespace, lib:Ontology and lib:Virtual, which we describe below.

lib:Ontology

This is a subclass of owl:Ontology. It has two subclasses, lib:Schema and lib:Instances. These three classes are currently processed equally. The following properties are recognised on lib:Ontology:

dc:title

Title of the ontology. Displayed by `rdf_list_library/0`.

owl:versionInfo

Version of the ontology. Displayed by `rdf_list_library/0`.

owl:imports

Ontologies imported. If `rdf_load_library/2` is used to load this ontology, the ontologies referenced here are loaded as well. There are two subProperties: lib:schema and lib:instances with the obvious meaning.

owl:providesNamespace

Informally, providing a namespace is defined as providing subjects that resides in the namespace.

owl:usesNamespace

Informally, using a namespace is defined as providing objects that reside in the namespace.

lib:source
Defines the named graph into which the resource is loaded. If this ends in a `/`, the basename of each loaded file is appended to the given source. Defaults to the URL the RDF is loaded from.

owl:baseURI
Defines the base for processing the RDF data. If not provided this defaults to the named graph, which in turn defaults to the URL the RDF is loaded from.

owl:blankNodes
One of `share` or `noshare`. A SWI-Prolog RDF library extension that allows for sharing equivalent blank nodes. Sharing is the default.

lib:Virtual
Virtual ontologies do not refer to an RDF resource themselves. They only import other resources. For example the W3C WordNet manifest defines `wn-basic` and `wn-full` as virtual resources. The lib:Virtual resource is used as a second rdf:type:

```
<wn-basic>
        a lib:Ontology ;
        a lib:Virtual ;
        ...
```

lib:Namespace
Defines a URL to be a namespace. The definition provides the preferred mnemonic and can be referenced in the lib:providesNamespace and lib:usesNamespace properties. The `rdf_load_library/2` predicates registers encountered namespace mnemonics with rdf-db using `rdf_register_ns/2`. Typically namespace declarations use @prefix declarations. E.g.

```
@prefix     lib: <http://www.swi-prolog.org/rdf/library/> .
@prefix    rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:rdfs
        a lib:Namespace ;
        lib:mnemonic "rdfs" ;
        lib:namespace rdfs: .
```

8.1.1 Finding manifest files

The initial manifest file(s) are loaded into the system using `rdf_attach_library/1`.

rdf_attach_library(+*FileOrDirectory*)
Load meta-data on RDF repositories from *FileOrDirectory*. If the argument is a directory, this directory is processed recursively and each file named `Manifest.ttl` or `Manifest.rdf` is loaded.

Declared namespaces are added to the rdf-db namespace list. Encountered ontologies are added to a private database of `rdf_list_library.pl`.[16] Each ontology is given an *identifier*, derived from the basename of the URL without the extension. This, using the declaration below, the identifier of the declared ontology is `wn-basic`.

```
<wn-basic>
        a lib:Ontology ;
        a lib:Virtual ;
        dc:title "Basic WordNet" ;
        ...
```

rdf_list_library
List the available resources in the library. Currently only lists resources that have a dc:title property. See section 8.2 for an example.

It is possible for the initial set of manifests to refer to RDF files that are not covered by a manifest. If such a reference is encountered while loading or listing a library, the library manager will look for a manifest file in the directory holding the referenced RDF file and load this manifest. If a manifest is found that covers the referenced file, the directives found in the manifest will be followed. Otherwise the RDF resource is simply loaded using the current defaults.

Further exploration of the library is achieved using `rdf_list_library/1` or `rdf_list_library/2`:

rdf_list_library(+*Id*)
Same as `rdf_list_library`(*Id, []*).

rdf_list_library(+*Id*, +*Options*)
Lists the resources that will be loaded if *Id* is handed to `rdf_load_library/2`. See `rdf_attach_library/2` for how ontology identifiers are generated. In addition it checks the existence of each resource to help debugging library dependencies. Before doing its work, `rdf_list_library/2` reloads manifests that have changed since they were loaded the last time. For HTTP resources it uses the HEAD method to verify existence and last modification time of resources.

rdf_load_library(+*Id*, +*Options*)
Load the given library. First `rdf_load_library/2` will establish what resources need to be loaded and whether all resources exist. Than it will load the resources.

8.2 Usage scenarios

Typically, a project will use a single file using the same format as a manifest file that defines alternative configurations that can be loaded. This file is loaded at program startup using `rdf_attach_library/1`. Users can now list the available libraries using `rdf_list_libraries/0` and `rdf_list_libraries/1`:

[16] We could have used the global RDF store, but decided against that to avoid poluting the triple space.

```
1 ?- rdf_list_library.
ec-core-vocabularies E-Culture core vocabularies
ec-all-vocabularies All E-Culture vocabularies
ec-hacks            Specific hacks
ec-mappings         E-Culture ontology mappings
ec-core-collections E-Culture core collections
ec-all-collections  E-Culture all collections
ec-medium           E-Culture medium sized data (artchive+aria)
ec-all              E-Culture all data
```

Now we can list a specific category using `rdf_list_library/1`. Note this loads two additional manifests referenced by resources encountered in `ec-mappings`. If a resource does not exist is is flagged using `[NOT FOUND]`.

```
2 ?- rdf_list_library('ec-mappings').
% Loaded RDF manifest /home/jan/src/eculture/vocabularies/mappings/Manifest.ttl
% Loaded RDF manifest /home/jan/src/eculture/collections/aul/Manifest.ttl
<file:///home/jan/src/eculture/src/server/ec-mappings>
. <file:///home/jan/src/eculture/vocabularies/mappings/mappings>
. . <file:///home/jan/src/eculture/vocabularies/mappings/interface>
. . . file:///home/jan/src/eculture/vocabularies/mappings/interface_class_mapping
. . . file:///home/jan/src/eculture/vocabularies/mappings/interface_property_mapp
. . <file:///home/jan/src/eculture/vocabularies/mappings/properties>
. . . file:///home/jan/src/eculture/vocabularies/mappings/ethnographic_property_m
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_properties.ttl
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_property_seman
. . <file:///home/jan/src/eculture/vocabularies/mappings/situations>
. . . file:///home/jan/src/eculture/vocabularies/mappings/eculture_situations.ttl
. <file:///home/jan/src/eculture/collections/aul/aul>
. . file:///home/jan/src/eculture/collections/aul/aul.rdfs
. . file:///home/jan/src/eculture/collections/aul/aul.rdf
. . file:///home/jan/src/eculture/collections/aul/aul9styles.rdf
. . file:///home/jan/src/eculture/collections/aul/extractedperiods.rdf
. . file:///home/jan/src/eculture/collections/aul/manual-periods.rdf
```

8.2.1 Referencing resources

Resources and manifests are located either on the local filesystem or on a network resource. The initial manifest can also be loaded from a file or a URL. This defines the initial *base URL* of the document. The base URL can be overruled using the Turtle @base directive. Other documents can be referenced relative to this base URL by exploiting Turtle's URI expansion rules. Turtle resources can be specified in three ways, as absolute URLs (e.g. `<http://www.example.com/rdf/ontology.rdf¿`), as relative URL to the base (e.g. `<../rdf/ontology.rdf¿`) or following a *prefix* (e.g. prefix:ontology).

The prefix notation is powerful as we can define multiple of them and define resources relative to them. Unfortunately, prefixes can only be defined as absolute URLs or URLs relative to the base URL. Notably, they cannot be defined relative to other prefixes. In addition, a prefix can only be followed by a Qname, which excludes `.` and `/`.

Easily relocatable manifests must define all resources relative to the base URL. Relocation is automatical if the manifest remains in the same hierarchy as the resources it references. If the manifest is copied elsewhere (i.e. for creating a local version) it can use @base to refer to the resource hierarchy. We can point to directories holding manifest files using @prefix declarations. There, we can reference *Virtual* resources using prefix:name. Here is an example, were we first give some line from the initial manifest followed by the definition of the virtual RDFS resource.

```
@base <http://gollem.science.uva.nl/e-culture/rdf/> .

@prefix base:           <base_ontologies/> .

<ec-core-vocabularies>
        a lib:Ontology ;
        a lib:Virtual ;
        dc:title "E-Culture core vocabularies" ;
        owl:imports
                base:rdfs ,
                base:owl ,
                base:dc ,
                base:vra ,
                ...
```

```
<rdfs>
        a lib:Schema ;
        a lib:Virtual ;
        rdfs:comment "RDF Schema" ;
        lib:source rdfs: ;
        lib:providesNamespace :rdfs ;
        lib:schema <rdfs.rdfs> .
```

8.3 Putting it all together

In this section we provide skeleton code for filling the RDF database from a password protected HTTP repository. The first line loads the application. Next we include modules that enable us to manage the RDF library, RDF database caching and HTTP connections. Then we setup the HTTP authetication, enable caching of processed RDF files and load the initial manifest. Finally `load_data/0` loads all our RDF data.

```
:- use_module(server).

:- use_module(library(http/http_open)).
```

```
:- use_module(library(semweb/rdf_library)).
:- use_module(library(semweb/rdf_cache)).

:- http_set_authorization('http://www.example.org/rdf',
                          basic(john, secret)).

:- rdf_set_cache_options([ global_directory('RDF-Cache'),
                           create_global_directory(true)
                         ]).


:- rdf_attach_library('http://www.example.org/rdf/Manifest.ttl').

%%      load_data
%
%       Load our RDF data

load_data :-
        rdf_load_library('all').
```

8.4 Example: A Manifest for W3C WordNet

The manifest below allows for loading WordNet in the two predefined versions using one of

```
?- rdf_load_library('wn-basic', []).
?- rdf_load_library('wn-full', []).
```

```
@prefix     lib: <http://www.swi-prolog.org/rdf/library/> .
@prefix     owl: <http://www.w3.org/2002/07/owl#> .
@prefix     rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix    rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix     xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix      dc: <http://purl.org/dc/elements/1.1/> .
@prefix wn20schema: <http://www.w3.org/2006/03/wn/wn20/schema/> .
@prefix wn20instances: <http://www.w3.org/2006/03/wn/wn20/instances/> .

# Source from http://www.cs.vu.nl/~mark/pub/wntestrdf.zip

:wn20instances
        a lib:Namespace ;
        lib:mnemonic "wn20instances" ;
        lib:namespace wn20instances: .

:wn20schema
        a lib:Namespace ;
```

```
        lib:mnemonic "wn20schema" ;
        lib:namespace wn20schema: .

:dc
        a lib:Namespace ;
        lib:mnemonic "dc" ;
        lib:namespace dc: .

:owl
        a lib:Namespace ;
        lib:mnemonic "owl" ;
        lib:namespace owl: .

#       WordNet

<wn-common>
        a lib:Instances ;
        a lib:Virtual ;
        rdfs:comment "Common files between full and basic version of WordNet" ;
        lib:source wn20instances: ;
        lib:instances <wordnet-attribute.rdf> ;
        lib:instances <wordnet-causes.rdf> ;
        lib:instances <wordnet-classifiedby.rdf> ;
        lib:instances <wordnet-entailment.rdf> ;
        lib:instances <wordnet-frame.rdf> ;
        lib:instances <wordnet-glossary.rdf> ;
        lib:instances <wordnet-hyponym.rdf> ;
        lib:instances <wordnet-membermeronym.rdf> ;
        lib:instances <wordnet-partmeronym.rdf> ;
        lib:instances <wordnet-sameverbgroupas.rdf> ;
        lib:instances <wordnet-similarity.rdf> ;
        lib:instances <wordnet-synset.rdf> ;
        lib:instances <wordnet-substancemeronym.rdf> .

<wnbasic.rdfs>
        a lib:Schema ;
        lib:source wn20schema: ;
        lib:usesNamespace :owl .

<wn-basic>
        a lib:Ontology ;
        a lib:Virtual ;
        dc:title "Basic WordNet" ;
        owl:versionInfo "2.0" ;
        rdfs:comment "Light version of W3C WordNet" ;
        lib:schema <wnbasic.rdfs> ;
        lib:source wn20instances: ;
```

```
        lib:instances <wn-common> ;
        lib:instances <wordnet-senselabels.rdf> ;
        lib:providesNamespace :wn20schema ;
        lib:providesNamespace :wn20instances .

<wnfull.rdfs>
        a lib:Schema ;
        lib:source wn20schema: ;
        lib:usesNamespace :owl .

<wn-full>
        a lib:Ontology ;
        a lib:Virtual ;
        dc:title "Full WordNet" ;
        owl:versionInfo "2.0" ;
        rdfs:comment "Full version of W3C WordNet" ;
        lib:schema <full/wnfull.rdfs> ;
        lib:source wn20instances: ;
        lib:instances <wn-common> ;
        lib:instances <wordnet-antonym.rdf> ;
        lib:instances <wordnet-derivationallyrelated.rdf> ;
        lib:instances <wordnet-participleof.rdf> ;
        lib:instances <wordnet-pertainsto.rdf> ;
        lib:instances <wordnet-seealso.rdf> ;
        lib:instances <wordnet-wordsensesandwords.rdf> ;
        lib:providesNamespace :wn20schema ;
        lib:providesNamespace :wn20instances .
```

9 library(semweb/sparql_client): SPARQL client library

This module provides a SPARQL client. For example:

```
?- sparql_query('select * where { ?x rdfs:label "Amsterdam" }', Row,
                [ host('dbpedia.org'), path('/sparql')]).

Row = row('http://www.ontologyportal.org/WordNet#WN30-108949737') ;
false.
```

Or, querying a local server using an `ASK` query:

```
?- sparql_query('ask { owl:Class rdfs:label "Class" }', Row,
                [ host('localhost'), port(3020), path('/sparql/')]).
Row = true.
```

sparql_query(+*Query, -Result, +Options*) *[nondet]*

Execute a SPARQL query on an HTTP SPARQL endpoint. *Query* is an atom that denotes the query. *Result* is unified to a term rdf(S,P,O) for `CONSTRUCT` and `DESCRIBE` queries, row(...) for `SELECT` queries and `true` or `false` for `ASK` queries. *Options* are

host(+*Host*)

port(+*Port*)

path(+*Path*)
The above three options set the location of the server.

search(+*ListOfParams*)
Provide additional query parameters, such as the graph.

variable_names(-*ListOfNames*)
Unifies ListOfNames with a list of atoms that describe the names of the variables in a `SELECT` query.

Remaining options are passed to `http_open/3`. The defaults for Host, Port and Path can be set using `sparql_set_server/1`. The initial default for port is 80 and path is `/sparql/`.

sparql_set_server(+*OptionOrList*)

Set sparql server default options. Provided defaults are: host, port and repository. For example:

```
    set_sparql_default([ host(localhost),
                         port(8080)
                         repository(world)
                       ])
```

sparql_read_xml_result(+*Input, -Result*)

Specs from `http://www.w3.org/TR/rdf-sparql-XMLres/`. The returned *Result* term is of the format:

select(*VarNames, Rows*)
Where VarNames is a term v(Name, ...) and Rows is a list of row(....) containing the column values in the same order as the variable names.

ask(*Bool*)
Where Bool is either `true` or `false`

10 library(semweb/rdf_compare): Compare RDF graphs

This library provides predicates that compare RDF graphs. The current version only provides one predicate: `rdf_equal_graphs/3` verifies that two graphs are identical after proper labeling of the blank nodes.

Future versions of this library may contain more advanced operations, such as diffing two graphs.

rdf_equal_graphs(+*GraphA, +GraphB, -Substition*) *[semidet]*

True if *GraphA* and *GraphB* are the same under *Substition*. *Substition* is a list of BNodeA = BNodeB, where BNodeA is a blank node that appears in *GraphA* and BNodeB is a blank node that appears in *GraphB*.

	Parameters
GraphA	is a list of rdf(S,P,O) terms
GraphB	is a list of rdf(S,P,O) terms
Substition	is a list if NodeA = NodeB terms.

To be done The current implementation is rather naive. After dealing with the subgraphs that contain no bnodes, it performs a fully non-deterministic substitution.

11 library(semweb/rdf_portray): Portray RDF resources

To be done

- Define alternate predicate to use for providing a comment
- Use rdf:type if there is no meaningful label?
- Smarter guess whether or not the local identifier might be meaningful to the user without a comment. I.e. does it look 'word-like'?

This module defines rules for `user:portray/1` to help tracing and debugging RDF resources by printing them in a more concise representation and optionally adding comment from the label field to help the user interpreting the URL. The main predicates are:

- `rdf_portray_as/1` defines the overall style
- `rdf_portray_lang/1` selects languages for extracting label comments

rdf_portray_as(+*Style*) *[det]*

Set the style used to portray resources. *Style* is one of:

`ns : id`
: Write as NS:ID, compatible with what can be handed to the rdf predicates. This is the default.

writeq
: Use quoted write of the full resource.

`ns : label`
: Write namespace followed by the label. This format cannot be handed to `rdf/3` and friends, but can be useful if resource-names are meaningless identifiers.

`ns` *:* `id` **=** `label`
: This combines ns:id with ns:label, providing both human readable output and output that can be pasted into the commandline.

rdf_portray_lang(+*Lang*) *[det]*

If *Lang* is a list, set the list or preferred languages. If it is a single atom, push this language as the most preferred language.

12 library(semweb/rdf_edit): Keep track of edits (deprecated)

It is anticipated that this library will eventually be superseeded by facilities running on top of the native `rdf_transaction/2` *and* `rdf_monitor/2` *facilities. See section 3.6.*

The module `rdf_edit.pl` is a layer than encasulates the modification predicates from section 3.3 for use from a (graphical) editor of the triple store. It adds the following features:

- *Transaction management*
  Modifications are grouped into *transactions* to safeguard the system from failing operations as well as provide meaningfull chunks for undo and journalling.
- *Undo*
  Undo and redo-transactions using a single mechanism to support user-friendly editing.
- *Journalling*
  Record all actions to support analysis, versioning, crash-recovery and an alternative to saving.

12.1 Transaction management

Transactions group low-level modification actions together.

rdfe_transaction(*:Goal***)**
Run *Goal*, recording all modifications to the triple store made through section 12.3. Execution is performed as in `once/1`. If *Goal* succeeds the changes are committed. If *Goal* fails or throws an exception the changes are reverted.

Transactions may be nested. A failing nested transaction only reverts the actions performed inside the nested transaction. If the outer transaction succeeds it is committed normally. Contrary, if the outer transaction fails, comitted nested transactions are reverted as well. If any of the modifications inside the transaction modifies a protected file (see `rdfe_set_file_property/2`) the transaction is reverted and `rdfe_transaction/1` throws a permission error.

A successful outer transaction ('level-0') may be undone using `rdfe_undo/0`.

rdfe_transaction(*:Goal, +Name***)**
As `rdfe_transaction/1`, naming the transaction *Name*. Transaction naming is intended for the GUI to give the user an idea of the next undo action. See also `rdfe_set_transaction_name/1` and `rdfe_transaction_name/2`.

rdfe_set_transaction_name(*+Name***)**
Set the 'name' of the current transaction to *Name*.

rdfe_transaction_name(*?TID, ?Name***)**
Query assigned transaction names.

rdfe_transaction_member(*+TID, -Action***)**
Enumerate the actions that took place inside a transaction. This can be used by a GUI to optimise the MVC (Model-View-Controller) feedback loop. *Action* is one of:

assert(*Subject, Predicate, Object*)

retract(*Subject, Predicate, Object*)

update(*Subject, Predicate, Object, Action*)

file(*load(Path)*)

file(*unload(Path)*)

12.2 File management

rdfe_is_modified(*?File*)
: Enumerate/test whether *File* is modified sinds it was loaded or sinds the last call to `rdfe_clear_modified/1`. Whether or not a file is modified is determined by the MD5 checksum of all triples belonging to the file.

rdfe_clear_modified(*+File*)
: Set the *unmodified-MD5* to the current MD5 checksum. See also `rdfe_is_modified/1`.

rdfe_set_file_property(*+File, +Property*)
: Control access right and default destination of new triples. *Property* is one of

  access(*+Access*)
  : Where access is one of `ro` or `rw`. Access `ro` is default when a file is loaded for which the user has no write access. If a transaction (see `rdfe_transaction/1`) modifies a file with access `ro` the transaction is reversed.

  default(*+Default*)
  : Set this file to be the default destination of triples. If *Default* is `fallback` it is only the default for triples that have no clear default destination. If it is `all` all new triples are added to this file.

rdfe_get_file_property(*?File, ?Property*)
: Query properties set with `rdfe_set_file_property/2`.

12.3 Encapsulated predicates

The following predicates encapsulate predicates from the `rdf_db` module that modify the triple store. These predicates can only be called when inside a *transaction*. See `rdfe_transaction/1`.

rdfe_assert(*+Subject, +Predicate, +Object*)
: Encapsulates `rdf_assert/3`.

rdfe_retractall(*?Subject, ?Predicate, ?Object*)
: Encapsulates `rdf_retractall/3`.

rdfe_update(+*Subject,* +*Predicate,* +*Object,* +*Action***)**
Encapsulates `rdf_update/4`.

rdfe_load(+*In***)**
Encapsulates `rdf_load/1`.

rdfe_unload(+*In***)**
Encapsulates `rdf_unload/1`.

12.4 High-level modification predicates

This section describes a (yet very incomplete) set of more high-level operations one would like to be able to perform. Eventually this set may include operations based on RDFS and OWL.

rdfe_delete(+*Resource***)**
Delete all traces of *resource*. This implies all triples where *Resource* appears as *subject*, *predicate* or *object*. This predicate starts a transation.

12.5 Undo

Undo aims at user-level undo operations from a (graphical) editor.

rdfe_undo
Revert the last outermost ('level 0') transaction (see `rdfe_transaction/1`). Successive calls go further back in history. Fails if there is no more undo information.

rdfe_redo
Revert the last `rdfe_undo/0`. Successive calls revert more `rdfe_undo/0` operations. Fails if there is no more redo information.

rdfe_can_undo(-*TID***)**
Test if there is another transaction that can be reverted. Used for activating menus in a graphical environment. *TID* is unified to the transaction id of the action that will be reverted.

rdfe_can_redo(-*TID***)**
Test if there is another undo that can be reverted. Used for activating menus in a graphical environment. *TID* is unified to the transaction id of the action that will be reverted.

12.6 Journalling

Optionally, every action through this module is immediately send to a *journal-file*. The journal provides a full log of all actions with a time-stamp that may be used for inspection of behaviour, version management, crash-recovery or an alternative to regular save operations.

rdfe_open_journal(+*File,* +*Mode***)**
Open a existing or new journal. If *Mode* equala `append` and *File* exists, the journal is first replayed. See `rdfe_replay_journal/1`. If *Mode* is `write` the journal is truncated if it exists.

rdfe_close_journal
Close the currently open journal.

rdfe_current_journal(*-Path*)

Test whether there is a journal and to which file the actions are journalled.

rdfe_replay_journal(*+File*)

Read a journal, replaying all actions in it. To do so, the system reads the journal a transaction at a time. If the transaction is closed with a *commit* it executes the actions inside the journal. If it is closed with a *rollback* or not closed at all due to a crash the actions inside the journal are discarded. Using this predicate only makes sense to inspect the state at the end of a journal without modifying the journal. Normally a journal is replayed using the `append` mode of `rdfe_open_journal/2`.

12.7 Broadcasting change events

To realise a modular graphical interface for editing the triple store, the system must use some sort of *event* mechanism. This is implemented by the XPCE library `broadcast` which is described in the XPCE User Guide. In this section we describe the terms brodcasted by the library.

rdf_transaction(*+Id*)

A 'level-0' transaction has been committed. The system passes the identifier of the transaction in *Id*. In the current implementation there is no way to find out what happened inside the transaction. This is likely to change in time.

If a transaction is reverted due to failure or exception *no* event is broadcasted. The initiating GUI element is supposed to handle this possibility itself and other components are not affected as the triple store is not changed.

rdf_undo(*+Type, +Id*)

This event is broadcasted after an `rdfe_undo/0` or `rdfe_redo/0`. *Type* is one of `undo` or `redo` and *Id* identifies the transaction as above.

13 Related packages and issues

The SWI-Prolog SemWeb package is designed to provide access to the Semantic Web languages from Prolog. It consists of the low level `rdf_db.pl` store with layers such as `semweb/rdfs.pl` to provide more high level querying of a triple set with relations such as `rdfs_individual_of/2`, `rdfs_subclass_of/2`, etc. SeRQL is a semantic web query language taking another route. Instead of providing alternative relations SeRQL defines a graph query on de *deductive closure* of the triple set. For example, under assumption of RDFS entailment rules this makes the query `rdf`(*S, rdf:type, Class*) equivalent to `rdfs_individual_of`(*S, Class*).

We developed a parser for SeRQL which compiles SeRQL path expressions into Prolog conjunctions of `rdf`(*Subject, Predicate, Object*) calls. *Entailment modules* realise a fully logical implementation of `rdf/3` including the entailment reasoning required to deal with a Semantic Web language or application specific reasoning. The infra structure is completed with a query optimiser and an HTTP server compliant to the Sesame implementation of the SeRQL language. The Sesame Java client can be used to access Prolog servers from Java, while the Prolog client can be used to access the Sesame SeRQL server. For further details, see the project home.

14 OWL

The SWI-Prolog Semantic Web library provides no direct support for OWL. OWL(-2) support is provided through Thea, an OWL library for SWI-Prolog See `http://www.semanticweb.gr/TheaOWLLib/`.

Acknowledgements

This research was supported by the following projects: MIA and MultimediaN project (www.multimedian.nl) funded through the BSIK programme of the Dutch Government, the FP-6 project HOPS of the European Commision.

The implementation of AVL trees is based on libavl by Brad Appleton. See the source file `avl.c` for details.

Index