Summer School – Advanced Concepts
for Databases and Logic Programming

Semantic Web Knowledge Bases

Monday, 18.09.2017

Prof. Dr. Dietmar Seipel
Julius–Maximilians University Würzburg

Contents

Overview

Front End Technology

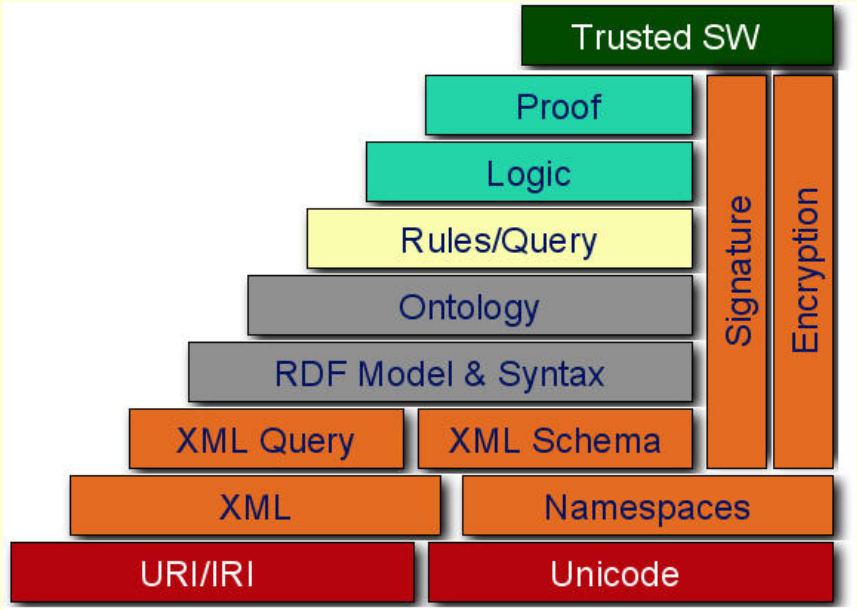
- In the semantic web, information can be represented in
  - the Web Ontology Language (OWL) or
  - the Resource Description Framework (RDF).
- We investigate query answering for RDF using the common query language SPARQL for RDF.
- OWL ontologies can be extended by rules, cf. the semantic web rule language SWRL.

Back End Technology

- For handling rules in a declarative way, deductive databases and logic programming use the query language DATALOG and the programming language PROLOG, respectively.
- For complex data structures including rules, advanced features are offered for
  - data modelling and
  - database programming.
- The deductive database system DDBASE uses an extension called DATALOG*.

1 Knowledge in the Semantic Web

The Semantic Web Stack of the W3C contains RDF and ontologies on the levels 4 and 5, respectively:



Linked Data

Linked Data is about using the Web to connect related data that wasn't previously linked, or using the Web to lower the barriers to linking data currently linked using other methods.

In Wikipedia, Linked Data is a term used to describe a recommended best practice for

- exposing,
- sharing, and
- connecting

pieces of data, information, and knowledge on the Semantic Web using URIs and RDF.

- The collection of interrelated datasets on the Web is referred to as Linked Data and it can be used by various *Semantic Web technologies* (RDF, OWL, SKOS, SPARQL, etc.) that provide an environment where applications can query that data, draw inferences using vocabularies, etc.
- All these technologies should provide data in a *common format* in order to facilitate on–the–fly data access or conversion.
- There have been many discussions among data generator, data publisher and data consumer communities about selecting a specific technology based on its validity, ease of use, learning curve, availability of tools in the commonly used programming environments etc.

The Resource Description Framework (RDF)

RDF is a language for representing information about resources in the World Wide Web. RDF is a *graph based data model* which provides a grammar for its syntax.

- RDF syntax can be written in various concrete *serialization formats*; the most used are Turtle, TSV, RDF/XML, TriG, N–Triples, N–Quads and JSON–LD.
- RDFa is also a concrete syntax for RDF, but it was not defined by the RDF Working Group.

Turtle (Terse RDF Triple Language, .ttl)

... is a format for expressing data in the RDF data model with a syntax similar to SPARQL. RDF, in turn, represents information using *triples*, each of which consists of a *subject*, a *predicate*, and an *object* – all expressed as a Web URI.

Turtle provides a way to group three URIs to make a triple, and provides ways to abbreviate such information, for example by factoring out common portions of URIs. Example:

```
@prefix ex: <http://example.org/>.
@prefix w3: <http://www.w3.org/2001/XMLSchema#/>
ex:Book ex:hasInformation _:id1.
_:id1
   ex:Information ex:Author;
   ex:LastName "Mueller" w3:string .
```

Tsv

A tab–separated values (Tsv) file (.tsv) is a simple text format for storing data in a tabular structure, e.g., database table or spreadsheet data, and a way of exchanging information between databases.

- Each record in the table is one line of the text file.
- Each field value of a record is separated from the next by a tab character.

The Tsv format is thus a type of the more general delimiter–separated values format.

RDF/XML

... is a syntax, defined by the W3C, to express (i.e. serialize) an RDF graph as an XML document.

- RDF/XML is sometimes misleadingly called simply RDF because it was introduced among the other W3C specifications defining RDF and it was historically the first W3C standard RDF serialization format.
- Although the RDF/XML format is still in use, other RDF serializations are now preferred by many RDF users, both because they are more human–friendly, and because some RDF graphs are not representable in RDF/XML due to restrictions on the syntax of XML QNames.

JSON–LD

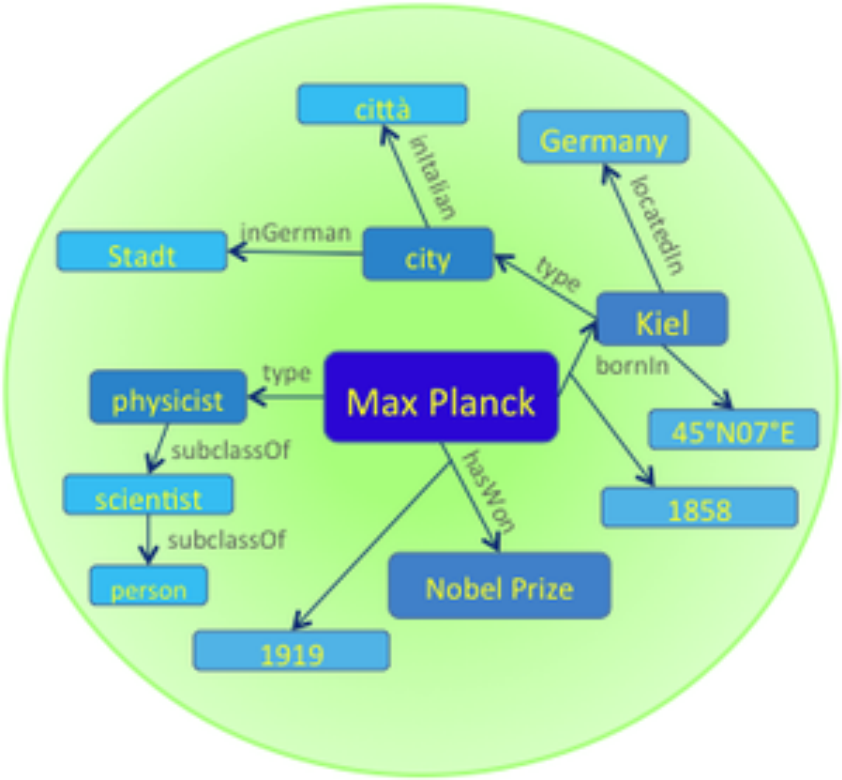
... is a concrete RDF syntax. Hence, a JSON–LD document is both an RDF document and a JSON document and correspondingly represents an instance of an RDF data model.

However, JSON–LD also extends the RDF data model to optionally allow JSON–LD to serialize Generalized RDF Datasets.

- In a generalized RDF triple, subject, predicate, and object can be an IRI, a blank node or a literal. A generalized RDF graph is a set of generalized RDF triples.
- A generalized RDF dataset comprises a distinguished generalized RDF graph, and zero or more pairs each associating an IRI, a blank node or a literal to a generalized RDF graph

Example: YAGO (Yet Another Great Ontology)

YAGO is a large general–purpose knowledge base derived from Wikipedia and other sources.



It contains

- more than 10 million entities (persons, organizations, cities, etc.) and
- more than 120 million facts about these entities.

It also contains time and space information, and unifies data from Wikipedias in 11 languages. YAGO is available for free at https://yago-knowledge.org

The information in YAGO is extracted from

- Wikipedia (e.g., categories, redirects, infoboxes),
- WordNet (e.g., synsets, hyponymy), and
- GeoNames.

- The accuracy of YAGO was manually evaluated to be above 95% on a sample of facts. To integrate it to the linked data cloud, YAGO has been linked to the DBpedia ontology and to the SUMO ontology.
- YAGO 3 is provided in Turtle and TSV formats. Dumps of the whole database are available, as well as thematic and specialized dumps.
  It can also be queried through various online browsers and through a SPARQL endpoint hosted by OpenLink Software.
- The source code of YAGO 3 is now available as well as open source under the GPL v3 license. E.g., https://github.com/yago-naga/yago3
- This allows everybody to use the extraction code, and also to contribute to YAGO.

Data and Knowledge Engineerung in the Semantic Web (Web 2.0)

We will consider knowledge modelled in the languages

- RDF/RDF(S) and
- OWL for ontologies

and have a closer look at

- *query answering* for RDF in SPARQL and
- *knowledge engineering* and *modelling* for OWL.

The rule–based extension SWRL of OWL can be handled using technologies from deductive databases with its declarative rule–language DATALOG, which we will investigate later.

1.1 RDF and SPARQL

Data Model:

The Resource Description Framework (RDF) is a data model for semantic web data, that is also used with OWL.

Query Language:

We will look at the query language SPARQL.

1.1.1 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a graph–based specification of a formal language, which is used to describe metadata. It is often mentioned in connection with the semantic web.

A statement is a triple s–p–o consisting of three components:

- the subject s is an IRI or a blank node,
- the predicate p is an IRI, and
- the object o is an IRI, a literal or a blank node.

The corresponding data type system, which describes the syntax, is called RDF Schema or RDFS. It is comparable to the DTD concept for XML.

Resources

For denoting resources, URLs and URIs are existing since a while. URLs were defined in 1994 by Tim Berners–Lee; in 1998, the URI syntax become a separate specification.

- Uniform Resource Locator (URL),
- Uniform Resource Identifier (URI), and
- Internationalized Resource Identifier (IRI).

After RDF was finished, IRIs actually became a reality in 2005, and in SPARQL and any following specs, the concept of URI references was replaced by IRIs. IRIs extend upon URIs by using the Universal Character Set, whereas URIs were limited to ASCII with far fewer characters.

Any IRI or literal denotes something in the world (the universe of discourse). These things are called resources.

Elements

In RDF, strings, numbers and dates are called *literals*. A literal consists of two or three elements:

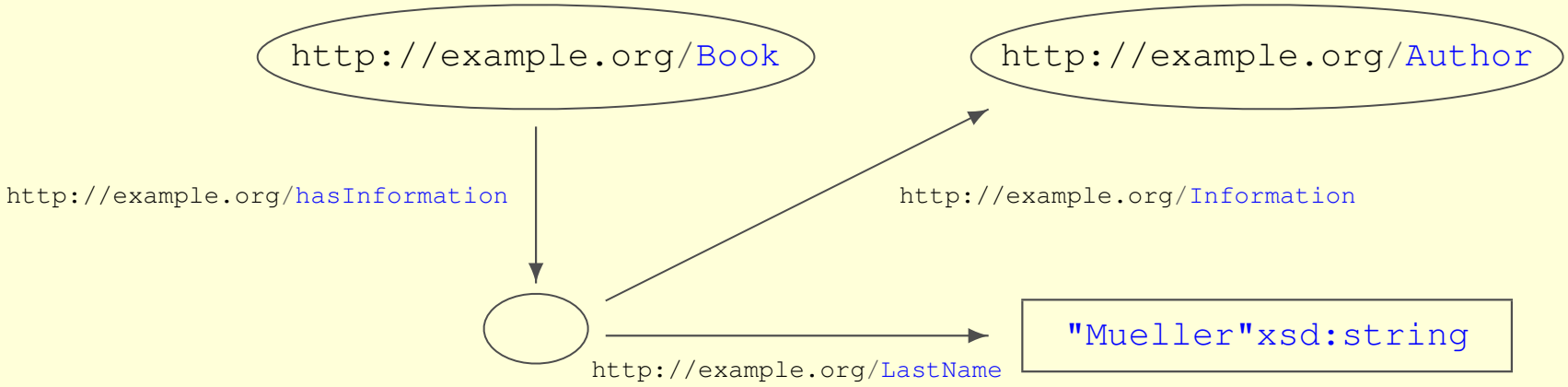
- a lexical form (represented as a unicode string, which should be in normal form C),
- a data type IRI (identifying a datatype that determines how the lexical form maps to a literal value), and
- a non–empty, well–formed language tag, if and only if the data type IRI is `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`.

The data type consists of a co–domain, a function, and a lexical form.
The RDF data types are similar to the data types in XML.

There are also blank nodes (BNodes) which represent subjects and objects.
Moreover it is possible to represent ambiguous relations with blank nodes.

Example: Knowledge Base about Books

RDF Graph



RDF/XML Syntax

```
<?xml version="1.0" encoding="utf?8"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:ex="http://example.org/">
   <rdf:Description rdf:about="http://example.org/Book">
      <ex:hasInformation rdf:nodeID="id1"/>
   </rdf:Description>
   <rdf:Description rdf:nodeID="id1">
      <ex:Information rdf:resource="http://example.org/Author"/>
      <ex:LastName
         rdf:Type="http://www.w3.org/2001/XMLSchema#string">
         Mueller
      </ex:LastName>
   </rdf:Description>
</rdf:RDF>
```

`rdf:nodeID="id1"` denotes a blank node.

Turtle Syntax

```
@prefix ex: <http://example.org/>.
@prefix w3: <http://www.w3.org/2001/XMLSchema#/>
ex:Book ex:hasInformation _:id1.
_:id1
   ex:Information ex:Author;
   ex:LastName "Mueller" w3:string .
```

The blank node `_:id1` is useful, since the RDF graph contains a node with two outgoing edges, which should be bundled.

1.1.2 Vocabulary

RDF Vocabulary

Besides `rdf:Description`, there is, e.g., the following vocabulary.

Classes

`rdf:XMLLiteral` – the class of XML literal values

`rdf:Property` – the class of properties

`rdf:Statement` – the class of RDF statements

`rdf:Alt, rdf:Bag, rdf:Seq` – containers of alternatives, unordered containers, and ordered containers
(`rdfs:Container` is a super–class of the three)

`rdf:List` – the class of RDF Lists

`rdf:nil` – an instance of `rdf:List` representing the empty list

Properties

`rdf:subject` – the subject of the subject RDF statement

`rdf:predicate` – the predicate of the subject RDF statement

`rdf:object` – the object of the subject RDF statement

`rdf:type` – an instance of `rdf:Property`
used to state that a resource is an instance of a class

`rdf:first` – the first item in the subject RDF list

`rdf:rest` – the rest of the subject RDF list after rdf:first

`rdf:value` – idiomatic property used for structured values

RDF(S) Vocabulary

Classes

- `rdfs:Resource` – the class resource, everything
- `rdfs:Literal` – the class of literal values, e.g. strings and integers
- `rdfs:Class` – the class of classes
- `rdfs:Datatype` – the class of RDF data types
- `rdfs:Container` – the class of RDF containers

Properties

`rdfs:subClassOf` – the subject is a subclass of a class

`rdfs:subPropertyOf` – the subject is a subproperty of a property

`rdfs:domain` – a domain of the subject property

`rdfs:range` – a range of the subject property

`rdfs:label` – a human–readable name for the subject

`rdfs:comment` – a description of the subject resource

`rdfs:member` – a member of the subject resource

`rdfs:seeAlso` – further information about the subject resource

`rdfs:isDefinedBy` – the definition of the subject resource

1.1.3 The Query Language SPARQL

SPARQL is a graph–based query language for RDF.

- The recursive acronym stands for SPARQL Protocol and RDF Query Language.
- There have been a W3C Working Draft in 2006 and a W3C Recommendation in 2008.

SPARQL queries are based on graph patterns, that search for parts of the RDF graph that are conform with them. The Turtle–like syntax can use

- query variables,
- modifiers,
- functions, filters, etc.

SPARQL variables can be in the position of subjects, predicates, and objects.

They have the form `?variable` or `$variable`; both are equivalent.

A prefix is given by

```
PREFIX prefixname: <URI>
```

Instead, a base name space can be defined by

```
BASE <URI>
```

In the following, we will use the RDF document below, which defines three RDF triples, for SPARQL queries:

```
@prefix ex: <http://example.org/>
ex:Semantic Web
   ex:PublishedBy
      <http://springer.com/Verlag> ;
   ex:title
      "Semantic Web - Grundlagen" ;
   ex:Author
      ex:Hitzler, ex:Kroetsch, ex:Rudolph, ex:Sure .
```

The following query determines all relationships between the two URIs `http://example.org/SemanticWeb` and `http://springer.com/Verlag`:

```
BASE <http://example.org/>
SELECT ?relationship
WHERE
   { <SemanticWeb>
     ?relationship
     <http://springer.com/Verlag> . }
```

The result is

relationship
http://example.org/PublishedBy

Blank nodes can be subjects or objects.

During the evaluation, variables and blank nodes can be replaced by arbitrary elements of the RDF graphs.

The replacements of blank nodes do not belong to the result of the query, since blank nodes just require the existence of a node.

If the same ID is used in the result, then the same node is referred to, but the IDs of the result do not necessarily coincide with the IDs of the graph.

Graph Patterns in SPARQL

- Simple graph patterns

  ```
  <Subject> <Predicate> <Object>
  ```

  can contain variables, blank nodes, and URIs.

- Complex graph patterns consist of several simple or empty graph patterns, that can be grouped by curly brakets.

Examples

The following query contains a complex graph pattern with three elements: a grouping graph pattern with two triples, an empty grouping graph pattern, and a simple graph pattern with one triple.

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
WHERE
   { { ?book ex:PublishedBy
          <http://springer.com/Verlag> .
       ?book ex:Title ?title . }
     { }
     ?book ex:Author ?author . }
```

Optional graph patterns have the operator `OPTIONAL`. Their variables do not have to be bound; if they are not bound, then they are returned as empty. The evaluation of optional graph patterns is done independently from each other.

```
{ ?book ex:PublishedBy <http://springer.com/Verlag> .
     OPTIONAL { ?book ex:Title ?title . }
     OPTIONAL { ?book ex:Author ?author . }
}
```

The following table could be a possible result, where an entry `_:a` denotes a blank node and an empty entry shows that a variable could not be bound:

book	title	author
http://example.org/book1	"Title1"	http://example.org/author1
http://example.org/book2	"Title2"	
http://example.org/book3	"Title3"	_:a
http://example.org/book4		_:a
http://example.org/book5		

Alternative Graph Patterns

The operator `UNION` basically represents the logical `OR` of its two arguments, which are graph patterns:

```
{ ?book ex:PublishedBy <http://springer.com/Verlag> .
   { ?book ex:Author ?author . } UNION
   { ?book ex:Verfasser ?author . } }
```

Data Values

SPARQL distinguishes between typed and untyped data, and between literals with and without language output.

Some data types, such as `xsd:Integer` and `decimal` are recognised automatically.

Filter

Filters refer to the complete grouping graph pattern in which they occur.

The following SPARQL query determines all books published by `Springer`, whose price is smaller than `35`:

```
@prefix ex: <http://example.org/>
SELECT ?book
WHERE
   { ?book ex:PublishedBy <http://spinger.com/Verlag> .
     ?book ex:Price ?price
     FILTER (?price < 35) }
```

Output Formats

SELECT

For the RDF document

```
@prefix ex: <http://example.org/> .
   ex:Gabi ex:Email "gabi@example.org" .
   ex:Gabi ex:Email "g_mueller@example.org" .
   ex:Gabi ex:Telephone "123456789" .
   ex:Gabi ex:Telephone "987654321" .
```

the following SPARQL query selects all Email addresses and telephone numbers of persons:

```
PREFIX ex: <http://example.org/>
SELECT *
WHERE { ?person ex:Email ?email .
        ?person ex:Telephone ?telephone . }
```

In general, the resulting table is potentially redundant, but it is easily readable by humans. Here we get:

Person	Email	Telephone
http://example.org/Gabi	"gabi@example.org"	"123456789"
http://example.org/Gabi	"gabi@example.org"	"987654321"
http://example.org/Gabi	"g_mueller@example.org"	"123456789"
http://example.org/Gabi	"g_mueller@example.org"	"987654321"

CONSTRUCT

With `CONSTRUCT`, the output can be guided by a pattern to avoid redundancy:

```
PREFIX ex: <http://example.org/>
CONSTRUCT {
   ?person ex:Mailbox ?email .
   ?person ex:Telephone ?telephone . }
WHERE {
   ?person ex:Email ?email .
   ?person ex:Telephone ?telephone . }
```

In the result, blank nodes from the pattern are replaced by a new ID for every variable assignment. With `CONSTRUCT`, also constant values are possible in the result.

ASK

`ASK` tests whether a query has a solution and correspondingly returns *true* or *false*.

```
PREFIX ex: <http://example.org/>
ASK { ?person ex:Email ?email .
      ?person ex:Telephone ?telephone . }
```

DESCRIBE

With `DESCRIBE`, it is possible to search for information with relevant properties.

```
PREFIX ex: <http://example.org/>
DESCRIBE <http://www.example.org/Gabi> ?person
WHERE { ?person ex:Email _a . }
```

Modifiers

Modifiers are used to control the exact form and size of the result list.

The following query returns all books together with their price, ordered by the price:

```
SELECT ?book, ?price
WHERE { ?book <http://example.org/price> ?price . }
ORDER BY ?price
```

Some modifiers are `ORDER BY`, `DISTINCT`, `REDUCED`, `OFFSET`, and `LIMIT`.

`DISTINCT` removes duplicates after the projection on the selected variables.

1.2 OWL Ontologies

In the following, we will explain OWL ontologies and show how to support knowledge engineering by detecting *anomalies* in the ontologies.

For OWL ontologies, the following reasoning tasks are important:

- *search:* query answering;
- *knowledge engineering / modelling:*
  analysis of the structure of the ontologies for anomalies.

Knowledge engineering and reasoning in the Semantic Web is based on ontology editors and specialized databases.

It can further be supported by deductive databases and logic programming techniques.

In the Semantic Web, *search* is more effective, since it is possible to reason about

- the ontology / taxonomy (i.e., the schema) and
- the instances.

This is called terminological or assertional (T–Box or A–Box) reasoning, respectively.

- In the following printer ontology, we could search for a printer from HP, and the result could be a laser–jet printer from HP, since the system knows that `hpLaserJetPrinter` is a sub–class of `hpPrinter`.
- It can also be derived, that all laser–jet printers from HP are no laser writers from Apple; in this case, this is very easy, since it is explicitely stored in the ontology.

1.2.1 The Web Ontology Language (OWL)

In OWL, we can mix concepts from

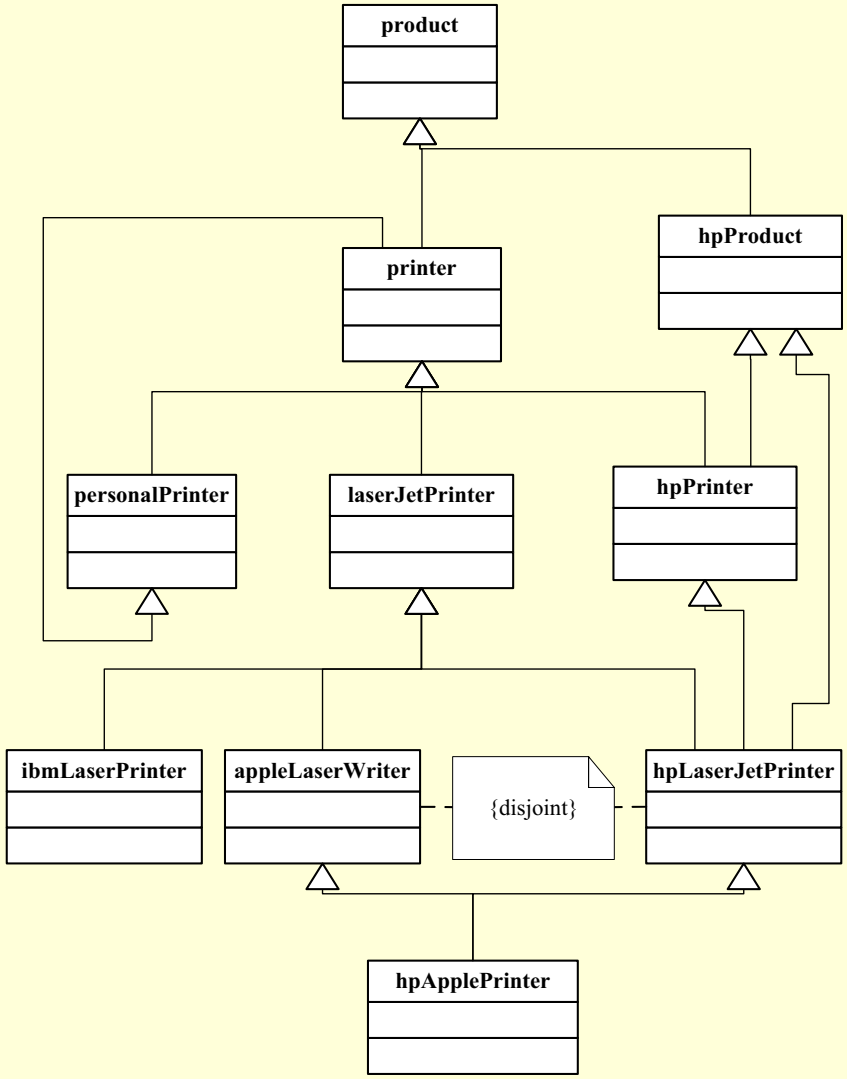
- `rdf` (Resource Description Framework) for defining instances and
- `rdfs` (`rdf` Schema) for defining the schema

of an application. Moreover, tags with the namespace `owl` are allowed.

There exist well–known, powerful tools for asking *queries* on and for *reasoning* with OWL ontologies.

The Semantic Web Rule Language (SWRL) incorporates *deduction rules* known from deductive databases and logic programming into OWL ontologies.

The Printer Ontology



The Printer Ontology in OWL

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:xsd="http://www.w3.org/2001/XLMSchema#"
   xmlns="file:/protege/Ontologies/p.owl#">

   <owl:Ontology rdf:about="Printer">
      <owl:VersionInfo> Printer Example, Version 1.3, 02.02.2013
      </owl:VersionInfo> </owl:Ontology>

   <owl:Class rdf:ID="printer"/>
   <owl:Class rdf:ID="laserJetPrinter">
      <rdfs:subClassOf rdf:resource="#printer"/> </owl:Class>
   ...
</rdf:RDF>
```

The following `owl:Class` element defines the class `appleLaserWriter`:

```
<owl:Class rdf:ID="appleLaserWriter">
   <rdfs:comment>
      Apple laser writers are laser jet printers
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="#laserJetPrinter"/>
   <owl:disjointWith rdf:resource="#hpLaserJetPrinter"/>
</owl:Class>
```

The `rdfs:subClassOf` sub–element states that `appleLaserWriter` is a sub–class of `laserJetPrinter`. The `owl:disjointWith` sub–element states that `appleLaserWriter` is disjoint from `hpLaserJetPrinter`.

Observe, that we refer using the attribute `rdf:resource` and a “`#`”, whereas the `owl:Class` element uses the attribute `rdf:ID` without a “`#`”.

The following `owl:Class` element defines a class of printers from a joint venture of HP and Apple:

```
<owl:Class rdf:ID="hpApplePrinter">
   <rdfs:comment>
      Printers from a joint venture of HP and Apple
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="#hpLaserJetPrinter"/>
   <rdfs:subClassOf rdf:resource="#appleLaserWriter"/>
</owl:Class>
```

The existence of such printers would contradict the `disjointWith` restriction between the classes `hpLaserJetPrinter` and `apperLaserWriter`.

The emptiness of the class `hpApplePrinter` can be detected by reasoners in the ontology editor Protégé.

Every `laserJetPrinter` is a `printer`, and every `hpPrinter` is an `hpProduct`:

```
<owl:Class rdf:ID="printer"/>

<owl:Class rdf:ID="laserJetPrinter">
   <rdfs:subClassOf rdf:resource="#printer"/>
</owl:Class>

<owl:Class rdf:ID="hpProduct"/>

<owl:Class rdf:ID="hpPrinter">
   <rdfs:subClassOf rdf:resource="#hpProduct"/>
</owl:Class>
```

Redundant subClassOf Relation

Since `hpLaserJetPrinter` is a sub–class of `hpPrinter` and `hpPrinter` is a sub–class of `hpProduct`, it is redundant to explicitely state that `hpLaserJetPrinter` is a sub–class of `hpProduct`.

```
<owl:Class rdf:ID="hpLaserJetPrinter">
   <rdfs:subClassOf rdf:resource="#laserJetPrinter"/>
   <rdfs:subClassOf rdf:resource="#hpPrinter"/>
   <rdfs:subClassOf rdf:resource="#hpProduct"/>
   <owl:disjointWith rdf:resource="#appleLaserWriter"/>
</owl:Class>
```

This redundancy is not an error. We could simply consider it as an anomaly, that should be reported to the knowledge engineer.

This anomaly is not reported by reasoners in the ontology editor Protégé.

Instances

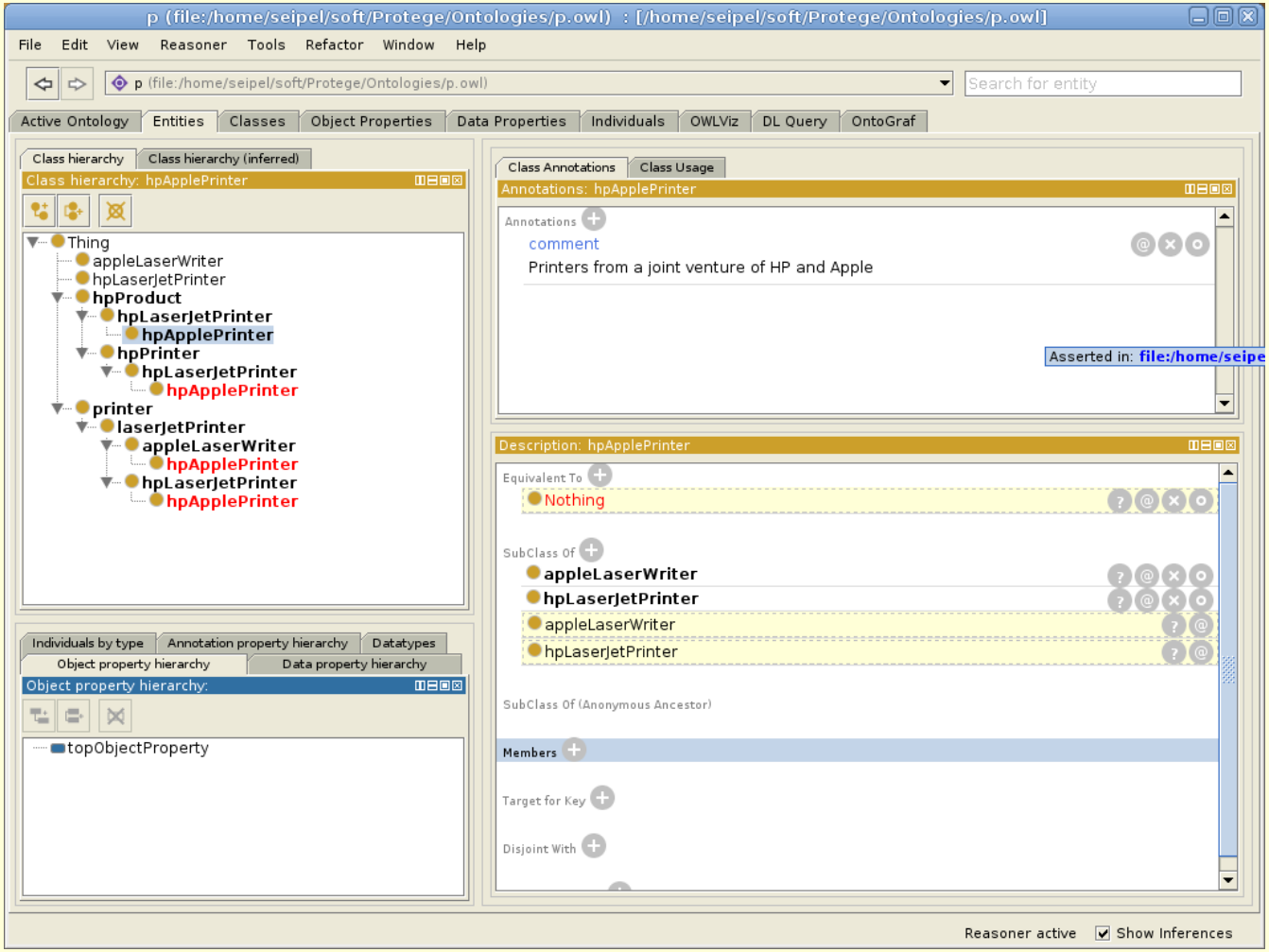
Finally, we have some instances of some of the defined classes:

```
<appleLaserWriter rdf:ID="1001"/>
<appleLaserWriter rdf:ID="1002"/>

<hpLaserJetPrinter rdf:ID="1003"/>
<hpLaserJetPrinter rdf:ID="1004"/>
```

As mentioned before, there cannot exist instances of the class `hpApplePrinter`.

1.2.2 The Ontology Editor Protégé



The ontology editor Protégé has some plugged in reasoners, such as

- FaCT++,
- HermiT, and
- Racer.

In the session shown in the screenshot above, the emptiness of the class `hpApplePrinter` was be detected by the ontology reasoner FaCT++.

It is inferred that the class `hpApplePrinter` is `EquivalentTo` the empty class `Nothing`. Clicking the question mark shows an explanation.

There are also databases for handling `rdf` data, so called triple stores, such as Sesame or Jena. They use extensions of SQL – most notably SPARQL – as a query language.

1.2.3 Design Analysis of Ontologies for Anomalies

Analysis and Verification of Knowledge

Cycle

```
?- isa(C1, C2), subClassOf(C2, C1).

C1 = personalPrinter,
C2 = printer
```

Partition Error

```
?- disjointWith(C1, C2),
   subClassOf(C, C1), subClassOf(C, C2).

C  = hpApplePrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter
```

The class `C` is a sub–class of two disjoint classes `C1` and `C2`.

Incompleteness

```
?- isa(C1, C), isa(C2, C), isa(C3, C),
   disjointWith(C1, C2), not(disjointWith(C2, C3)).

C  = laserJetPrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter,
C3 = ibmLaserPrinter
```

The class `C` has three sub–classes `C1`, `C2` and `C3`, from which only the two sub–classes `C1` and `C2` are declared as disjoint in the knowledge base.

The fact that `C2` and `C3` are disjoint and that `C1` and `C3` are disjoint as well, possibly was forgotten by the knowledge engineer during the creation of the knowledge base.

Redundant subClassOf/instanceOf Relations

```
% redundant_isa(?Chain) <-

redundant_isa(C1->C2->C3) :-
   isa(C1, C2), subClassOf(C2, C3),
   isa(C1, C3).

?- redundant_isa(Chain).

Chain = hpLaserJetPrinter -> hpPrinter -> hpProduct
```

The sub–class relation between `C1` and `C3` can be derived by transitivity over the class `C2`.

Here, `isa(C1, C2), subClassOf(C2, C3),` requires that this deduction is done over at least two levels.

Undefined Reference

During the development of an ontology in OWL, it is possible that we reference a class that we have not yet defined.

```
% undefined_reference(+Owl, ?Ref) <-

undefined_reference(Owl, Ref) :-
   rdf_reference(Owl, Ref),
   not(owl_class(Owl, Ref)).
```

If we load such an ontology into Protégé, then the ontology reasoners may produce wrong results, even for unrelated parts of the ontology.

We have worked on the analysis and evaluation of declarative rules in DATALOG* for decision support systems.

A design analysis of ontologies with rules (c.f. the Semantic Web Rule Language SWRL, which extends OWL by rules) is given in

J. Baumeister, D. Seipel:
Anomalies in Ontologies with Rules,
Journal of Web Semantics: Science, Services and Agents
on the World Wide Web 8 (2010), No. 1, pp. 55–68.

Research Issues for Semantic Web Knowledge Bases

- data and knowledge engineering
- knowledge representation with RDF and ontologies
- rules in the Semantic Web
  - extensions of OWL
  - SWRL
- analysis and verification of knowledge

for publications or theses (master, PhD)

2 Knowledge in Deductive Databases

Logic programming is distinguished from other programming paradigms by features such as the ease of handling the *data structure* of *terms* and the powerful built–in *control structure* of *backtracking*.

- The logic programming language PROLOG can represent and access classical relational databases and semantic web knowlege.
  It is very well–suited for embedded database programming.
- Relations and complex objects can be represented as term structures.
- Declarative rules can represent both inference rules for deriving conclusions from given information and integrity constraints.

In deductive databases – i.e., in the database context – frequently a restricted version called DATALOG is used.

A survey of deductive databases is given in

J. Minker, D. Seipel, C. Zaniolo:
Logic and Databases: History of Deductive Databases,
in Handbook of Computational Logic, North Holland, 2014.

The extension of deductive databases by disjunction (incomplete information) is described in

J. Minker, D. Seipel:
Disjunctive Logic Programming: A Survey and Assessment,
in A. Kakas, F. Sadri (Eds.), Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I,
Springer, LNAI 2407, 2002, pp. 472–511.

A current renaissance of DATALOG has been described by S. Abiteboul.

2.1 Data Model / Syntax of Logic Programming

Constant Symbol:	a, 10, 'Smith, John B.'
Variable Symbol:	X, Lname (starts with a capital letter)
Term:	$f(t_1, \ldots, t_n)$,
	with function symbol f and terms t_i
	a, X (constant and variable symbols are terms),
	f(g(a,b),X,10), a*(b+c) (complex terms),
	[LNAME, . . . , DNO] (this is a list)
Predicate Symbol:	employee, e_works_on_p, transitive_closure
Atom:	$p(t_1, \ldots, t_n)$,
	with predicate symbol p and terms t_i.

Domain–Specific Language (DSL)

DSLs with an extended syntax can be defined.

Statements using Infix–Notation, e.g.

```
'Jack' likes books.
```

Rules for Decision Support using Quasi–Quotations, e.g.

```
| rule ||
if Weather = rainy and Umbrella = no
or Weather = Thunderstorm
then Clothes = wet .  |
```

D. Seipel, F. Nogatz, S. Abreu:

Domain–Specific Languages in PROLOG *for Declarative Expert Knowledge in Rules and Ontologies*,
Journal of Computer Languages, Systems & Structures, 2017.

Extra–Logical Features

- `assert` and `retract` can update the internal PROLOG database
- meta–predicates like

  `maplist`, `findall`, and `ddbase_aggregate`

  form control structures
- the cut "`!`" freezes variable bindings and cuts off alternative computations paths
- ...

Terms in Infix / Prefix Form

It is possible to define binary, infix functors $\odot$ in PROLOG. Then, the binary term $\odot(t_1, t_2)$ can be represented in infix form as $t_1 \odot t_2$.

The infix term

```
Jack likes books
```

would be

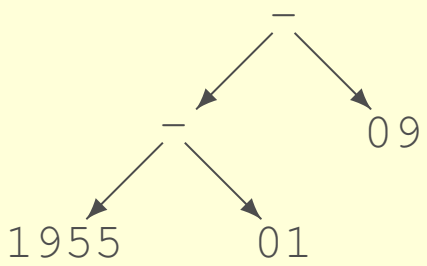
```
likes('Jack', books)
```

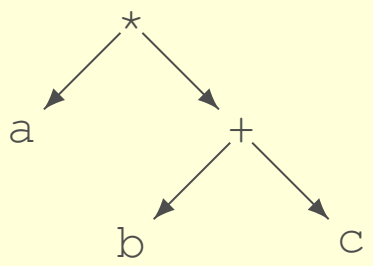
in standard prefix form.

Terms in Infix / Prefix Form

- The infix term `1955-01-09` representing a date has the prefix form `-(-(1955,01),09).`
- The infix term `a*(b+c)` representing an arithmetic expression has the prefix form `*(a,+(b,c)).`

The operator trees for the terms above are given in the following:

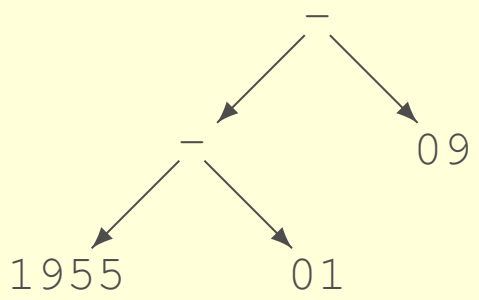


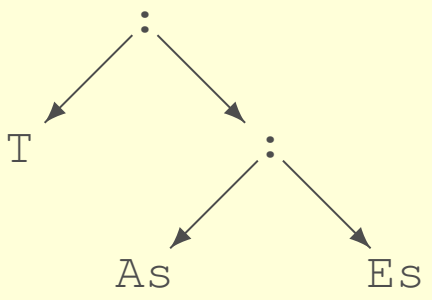


When an infix functor $\odot$ is used multiple times in a term $a \odot b \odot c$, then there are rules in PROLOG that determine whether a and b or b and c are joined first in the prefix form.

- The infix term `1955-01-09` representing a date has the prefix form `-(-(1955,01),09)`.
- The infix term `T:As:Es` representing an XML element has the prefix form `:(T,:(As,Es))`.

The operator trees for the terms above are given in the following:





Term Representation for Lists: Circumfix

A non–empty list is represented as a binary term structure `.(X, Xs)`, where

- the head `X` is the first element and
- the tail `Xs` represents the rest of the list.

The list functor "`.`" – which recently became "`[|]`" in SWI PROLOG – is binary, and the empty list is represented by "`[]`".

Additionally, PROLOG supports the compact list notation $[X_1, X_2, \ldots, X_n]$. It is equivalent to $[X_1 | [X_2, \ldots, X_n]]$.

```
[c] = .(c, [])
[a, b, c] = .(a, .(b, .(c, [])))
```

This *syntactic sugar* helps the user to better understand the list.

Term Representation for XML

An XML element

```
<table name="employee">
   <attribute name="FNAME"/>
</table>
```

can be represented by a complex term in field notation (FN):

```
table:[name:employee]:[
   attribute:[name:'FNAME']:[] ].
```

This infix form is using the binary functor ":".

Above, the list of sub–elements of the `attribute` element is empty.

Facts, Rules, and Goals

Literal: atom A oder negated atom `not`(A)

Fact: A

with atom A; e.g.,

```
employee('John', 'B', 'Smith', ...)
```

Rule: $\underbrace{A}_{\text{head}}$ `:-` $\underbrace{B_1, \ldots, B_m}_{\text{body}}$

with atom A and literals B_i, example later

Goal: `:-` $B_1, \ldots, B_m$

with literals B_i

A set of facts for the same predicate symbol corresponds to a relation in databases. Rules generalize views. Goals are used for expressing queries.

The Logic Programming Language PROLOG

Well–known algorithms for searching in graphs and binary search trees can be implementated in PROLOG.

The benefits of PROLOG are

- the elegant handling of data structures (lists, trees, XML),
- (implicit) backtracking, and
- the compact representation of case distinctions in different rules.

The algorithms are typically recursive. Recursion can be formulated nicely due to the compact list access.

Also meta–predicates support a compact and elegant encoding.

Important Concepts

- Terms (for Data / Control Structures) and Unification
- SLDNF–Resolution for Top–Down–Evaluation
- Backtracking

PROLOG allows for

- declarative programming,
- compact programs, and
- rapid prototyping, agile software development.

Data Structures, Operations, and Control Structures

- The restriction to a few basic data types and a single complex data type – namely the terms, which are generic and subsume all the other types – standardizes the data structures.
- There are no explicit type declarations.
- There exists a large collection of generic operations that are applicable to terms – and thus to all data types.
- Frequently, meta–predicates are used. In addition to standard control structures, such as branching (if–then–else), loops (for, while), and recursion, user–defined control structures can be built as meta–predicates.

The Meta–Predicate `findall/3`

Finding of all solutions for a goal:

```
findall( X,
   goal(X),
   Xs )
```

Declare allows for the following equivalent set notation (DSL):

```
Xs <= { X | goal(X) }
```

Further important meta–predicates are `checklist/2` and `maplist/3` for lists, as well as the predicates for loops (control structures) from the library `loops.pl` (e.g., `foreach-do`).

Software Engineering Aspects

PROLOG supports abstraction and compact code, and thus stimulates refactoring:

- The generic type of terms with generic operations supports abstraction and code reuse.
- User–defined control structures allow for further abstraction.
- Unification, implicit backtracking, and abstaining from explicit type declarations, result in very compact code and support rapid prototyping.
- Declarativity makes the code much more readable and thus extensible.

Switching from conventional programming languages to the logic programming paradigm is difficult and usually requires training and effort.

2.2 Data and Knowledge Engineering in DATALOG

Modelling Relational Databases in Logic Programming

- The syntax of logic programming can be used for representing *tables* from relational databases. The tuples of a table become PROLOG facts with the same predicate symbol – usually, the table name is used.
- The *data dictionary* of a relational database can also be represented using PROLOG facts. This can be done using PROLOG terms that correspond to an XML representation of the data dictionary.
- *Queries* and *integrity constraints* can be represented as PROLOG rules. Conjunctive queries are asked in the form of PROLOG goals, which are then evaluated using the PROLOG rules.

DATALOG and its Extensions

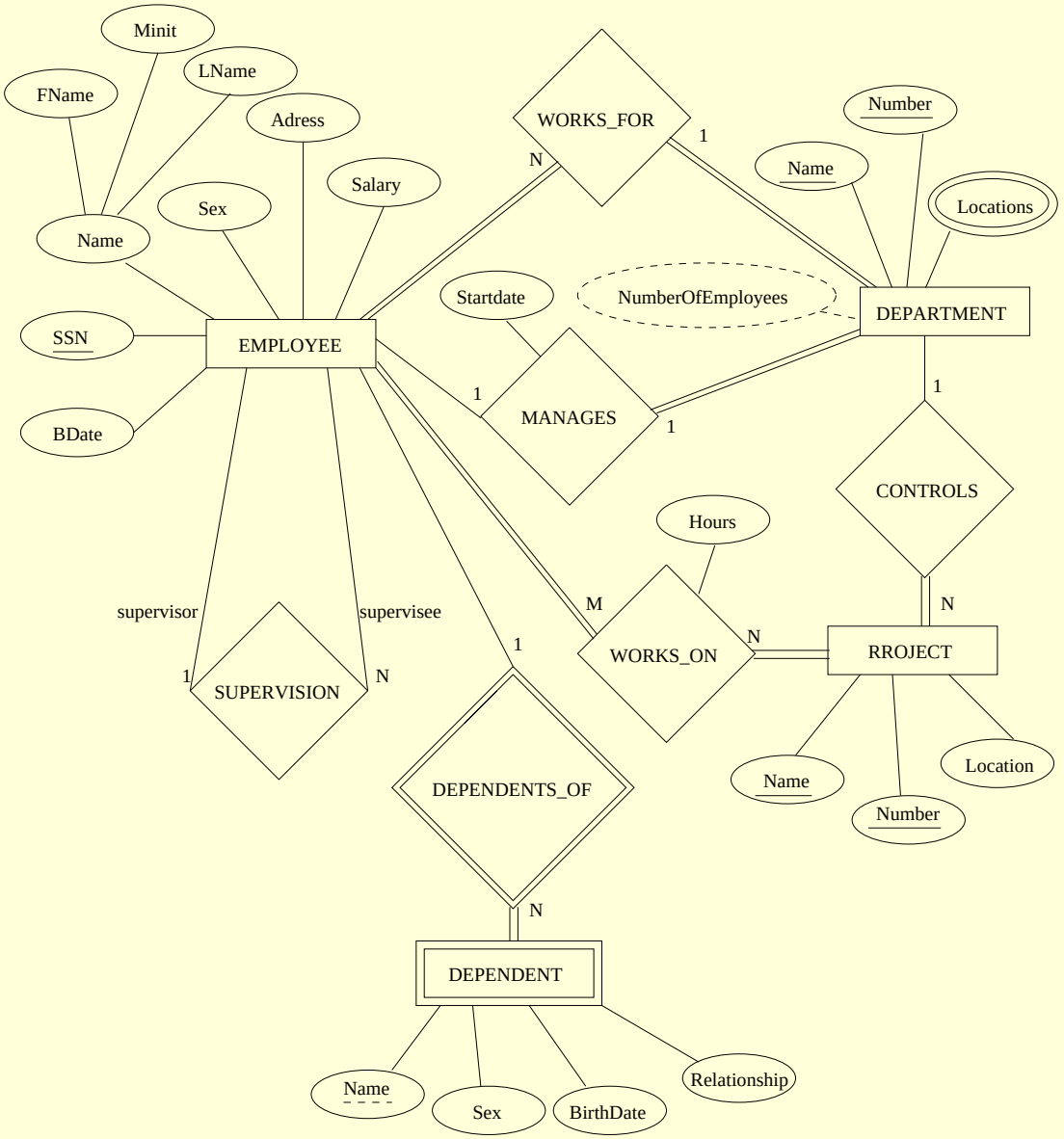
- DATALOG is a restricted version of PROLOG without function symbols and extra–logical features, that is very well–investigated in theory. It ensures termination and the efficient evaluation of recursive rules.
- In practical applications, several extension are used. The language DATALOG$^{\mathsf{fun}}$ adds function symbols, and DATALOG$^{\mathsf{not}}$ adds default negation; DATALOG$^{\mathsf{fun,not}}$ adds both.
- As part of Declare, we have developed a deductive database system DDBASE using the extended language DATALOG*, that integrates PROLOG and its extra–logical features into DATALOG$^{\mathsf{fun,not}}$.

2.2.1 Database Tables

The relational database `company` from the well–known book of Elmasri and Navathe contains – amoung others – the tables

- `employee`,
- `department`,
- `works_on`, and
- `project`.

The complete *ER–diagram* of the relational database `company` follows.


Minit
FName
LName
Adress
Salary
Sex
Name
SSN
BDate
EMPLOYEE
WORKS_FOR
N
1
Number
Name
Locations
DEPARTMENT
Startdate
NumberOfEmployees
1
MANAGES
1
1
CONTROLS
Hours
M
WORKS_ON
N
N
RROJECT
supervisor
supervisee
1
SUPERVISION
1
N
1
DEPENDENTS_OF
N
Name
Location
Number
DEPENDENT
Name
Sex
BirthDate
Relationship

The relevant part of the relational database `company` consists of the following 4 tables:

EMPLOYEE									
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	444444444	1955-01-09	731 Fondren, Houston, TX	M	30000	222222222	5
Franklin	T	Wong	222222222	1945-12-08	638 Voss, Houston, TX	M	40000	111111111	5
Alicia	J	Zelaya	777777777	1958-07-19	3321 Castle, Spring, TX	F	25000	333333333	4
Jennifer	S	Wallace	333333333	1931-06-20	291 Berry, Bellaire, TX	F	43000	111111111	4
Ramesh	K	Narayan	555555555	1952-09-15	975 Fire Oak, Humble, TX	M	38000	222222222	5
Joyce	A	English	666666666	1962-07-31	5631 Rice, Houston, TX	F	25000	222222222	5
Ahmad	V	Jabbar	888888888	1959-03-29	980 Dallas, Houston, TX	M	25000	333333333	4
James	E	Borg	111111111	1927-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT			
DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Headquarters	1	111111111	1971-06-19
Administration	4	333333333	1985-01-01
Research	5	222222222	1978-05-22

WORKS_ON		
ESSN	PNO	HOURS
111111111	20	NULL
222222222	2	10.0
222222222	3	10.0
333333333	20	15.0
333333333	30	20.0
444444444	1	32.5
444444444	2	7.5
555555555	3	40.0
666666666	1	20.0
666666666	2	20.0
777777777	10	10.0
777777777	30	30.0
888888888	10	35.5
888888888	30	5.0

PROJECT			
PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

A database table p can be represented by a set of DATALOG facts, namely one fact $p(t_1, \ldots, t_n)$ for each tuple $(t_1, \ldots, t_n)$ of the table. Conversely, the set of all given facts for a predicate p corresponds to a database table.

The database tables `employee, department, works_on`, and `project` are represented by DATALOG facts inluding the following:

```
employee('John', 'B', 'Smith', 444444444,
   '1955-01-09', '731 Fondren, Houston, TX',
   'M', 30000, 222222222, 5).
department('Research', 5, 222222222, '1978-05-22').
works_on(444444444, 1, 32.5).
project('ProductX', 1, 'Bellaire', 5).
...
```

Above, quoted values are strings. Non–quoted values are numbers.

A relational tuple can be represented by RDF triples. E.g., the tuple

```
employee('John', 'B', 'Smith', 444444444,
   '1955-01-09', '731 Fondren, Houston, TX',
   'M', 30000, 222222222, 5)
```

can be represented by 11 RDF triples *s–p–o*; we show 7 of them here:

```
EMPLOYEE/SSN=444444444 - EMPLOYEE#FNAME -   "John",
EMPLOYEE/SSN=444444444 - EMPLOYEE#MINIT -   "B",
EMPLOYEE/SSN=444444444 - EMPLOYEE#LNAME -   "Smith",
EMPLOYEE/SSN=444444444 - EMPLOYEE#SSN -     "444444444",
EMPLOYEE/SSN=444444444 - EMPLOYEE#BDATE -   "1955-01-09",
EMPLOYEE/SSN=444444444 - EMPLOYEE#ADDRESS - "731 Fondren, Houston, TX",
...
EMPLOYEE/SSN=444444444 - rdf:type -         EMPLOYEE.
```

The attribute/value pairs $A{:}v$ of a tuple of a table T become triples s–p–o:

- The subject s is obtained as $T/K = v$ by concatenating the table name with the primary key K and its value v.
- The predicate p is obtained as $T\#A$ by concatenating the table name with the attribute A;
- The object o is obtained from the value v.

This mapping corresponds to *reification*. The last triple s–`rdf:type`–T assigns the subject s to the relation T.

The relational representation with n–ary tuples (here $n = 10$) is much more compact and readable than the RDF representation with triples ($n = 3$).

For tables T with several attributes $A_1, \ldots, A_k$, in the primary key, such as WORKS_ON, the subject

$$T/A_1 = v_1; \ldots; A_k = v_k$$

of a tuple is composed by concatenating the strings obtained from the corresonding attribute/value pairs A_1:v_1, ... A_k:v_k.

E.g., from

```
works_on(444444444, 1, 32.5).
```

we get the triple *s*–*p*–*o*:

```
WORKS_ON/ESSN=444444444;PNO=1 -
   WORKS_ON#HOURS - "32.5"
```

2.2.2 Views and Queries vs. Rules and Goals

A rule corresponds to a `VIEW` statement defining a relation for the head predicate.

An SQL VIEW describing a projection

```
CREATE VIEW EMPLOYEE_DEPARTMENT AS
   SELECT SSN, DNO
   FROM   EMPLOYEE
```

corresponds to a DATALOG rule:

```
employee_department(SSN, DNO) :-
   employee(_, _, _, SSN, _,_,_,_,_, DNO).
```

Both describe a projection of `EMPLOYEE` on the two attributes `SSN` and `DNO`.

The anonymous variables "_" are wild cards for argument positions of `employee` which we are not interested in.

Applying the rule to the DATALOG facts of the database derives 8 DATALOG facts from to the original facts for `employee`:

```
employee_department('111111111', 1)
employee_department('222222222', 5)
employee_department('333333333', 4)
...
```

An SQL VIEW with a join

```
CREATE VIEW E_WORKS_ON_P AS
   SELECT FNAME, LNAME, PNAME, HOURS
   FROM   EMPLOYEE, WORKS_ON, PROJECT
   WHERE  EMPLOYEE.SSN = WORKS_ON.ESSN
   AND    PROJECT.PNUMBER = WORKS_ON.PNO
```

corresponds to a DATALOG rule:

```
e_works_on_p(FNAME, LNAME, PNAME, HOURS) :-
   works_on(SSN, PNO, HOURS),
   employee(FNAME, _, LNAME, SSN, _,_,_,_,_,_),
   project(PNAME, PNO, _,_).
```

We have 3 atoms in the body of the DATALOG rule, since we join 3 tables according to `FROM`. Every atom represents a tuple of the database.

The equalities from the `WHERE` part of the SQL statement are represented by the equal arguments `SSN` and `PNO` in the atoms of the rule body.

Applying the rule to the database tables derives DATALOG facts that – compared to the facts form `works_on` – contain the names of the employees and projects instead of the numbers:

```
e_works_on_p('Ahmad', 'Jabbar', 'Computerization', 35.5)
e_works_on_p('Ahmad', 'Jabbar', 'Newbenefits', 5.0)
e_works_on_p('Alicia', 'Zelaya', 'Computerization', 10.0)
e_works_on_p('Alicia', 'Zelaya', 'Newbenefits', 30.0)
e_works_on_p('Franklin', 'Wong', 'ProductY', 10.0)
e_works_on_p('Franklin', 'Wong', 'ProductZ', 10.0)
e_works_on_p('James', 'Borg', 'Reorganization', 'NULL')
...
```

- SQL VIEW:

```
CREATE VIEW DIRECT_SUPERVISOR AS
   SELECT SSN, SUPERSSN
   FROM   EMPLOYEE
```

- DATALOG rule:

```
direct_supervisor(SSN_1, SSN_2) :-
   employee(_,_,_, SSN_2, _,_,_,_, SSN_1, _),
   SSN_1 \= 'NULL'.
```

Both describe a projection. The built–in atom `SSN_1 \= 'NULL'` ensures that the supervisor is not null – e.g., the value of the attribute `SUPERSSN` of `'James E. Borg'` is `'NULL'`, since he is the boss of the company and has no supervisor.

- SQL SELECT:

```
SELECT *
FROM   E_WORKS_ON_P
```

  The SELECT statement calls the view.

- DATALOG goal:

```
?- e_works_on_p(FNAME, LNAME, PNAME, HOURS).
```

The query is submitted to the DATALOG interpreter as a goal.
The goal corresponds to the SELECT statement calling the view.

2.2.3 Integrity Constraints in DATALOG*

- Primary Key Constraint for Employee:

```
primary_key_violation(employee, X, Y) :-
    X = employee(_,_,_, SSN, _,_,_,_,_,_),
    Y = employee(_,_,_, SSN, _,_,_,_,_,_),
    call(X), call(Y), X \= Y.
```

- Foreign Key Constraint for Employee:

```
foreign_key_violation(employee('DNO'), X) :-
    X = employee(_,_,_,_,_,_,_,_,_, DNO),
    call(X),
    not(department(_, DNO, _,_)).
```

In DDBASE, the primary and foreign key contraints of a relational database are transformed to such rules, which are then tested on database updates.

Argument Positions vs. Field Notation (FN)

- Like in other programming languages, the arguments t_i of an atom $p(t_1, \ldots, t_n)$ are handed over by position in DATALOG. E.g., in

  ```
  works_on(S, P, H),
  ```

  the first position $t_1 =$ `S` is the SSN of an employee who has worked on the project with the number $t_2 =$ `P` for $t_3 =$ `H` hours.

- In the database context, we could use a meta–interpreter for accessing arguments in *field notation* – in a more abstract way – by their corresponding attribute name. Then, according to the database schema,

  ```
  works_on('PNO':P, 'ESSN':S)
  ```

  means that the employee `S` has worked on the project `P`. The order of the arguments does not matter, and it is not necessary to reference the hours.

Semantic Constraints in Field Notation (FN)

- No employee should earn more than his manager:

```
trigger(salary, X, Y) :-
   employee('SSN':X, 'SALARY':S1, 'SUPERSSN':Y),
   employee('SSN':Y, 'SALARY':S2),
   S1 > S2.
```

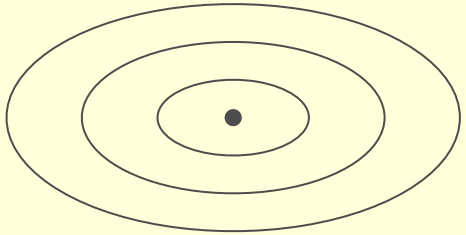
- Which employee works on a foreign project ?

```
trigger(employee_works_on_foreign_project, E, P) :-
   works_on('ESSN':E, 'PNO':P),
   employee('SSN':E, 'DNO':D1),
   project('PNUMBER':P, 'DNUM':D2),
   D1 \= D2.
```

FN abstracts from argument positions: `employee('SSN':E, 'DNO':D1)` corresponds to `employee(_,_,_, E, _,_,_,_,_, D1).`

2.2.4 Bottom–Up Evaluation of DATALOG

The bottom–up evaluation *iteratively* enlarges the relations for the predicates by repeatedly evaluating all rules until a fixpoint is reached.
Thus, e.g., all transitive supervisors can be derived – which is usually not possible using SQL systems.



Every rule instance A :– $B_1, \ldots, B_n$ derives facts A for its head predicate.
The facts for the body predicates B_i are derived using rules themselves.

Thus, it can happen that a rule transitively helps to derive facts for one of its body predicates (recursion).

Non–Recursive Example

A DATALOG program containing the two rules from above

```
employee_department(SSN, DNO) :-
   employee(_, _, _, SSN, _,_,_,_,_, DNO).
e_works_on_p(FNAME, LNAME, PNAME, HOURS) :-
   works_on(SSN, PNO, HOURS),
   employee(FNAME, _, LNAME, SSN, _,_,_,_,_,_),
   project(PNAME, PNO, _,_).
```

and the DATALOG facts for the database tables can be evaluated bottom–up by applying both rules – which we have already shown.

A DATALOG program containing the following 2 rules and 4 facts

```
a(X, Y) :- b(Y, X).
b(X, Y) :- c(X, Y), d(X).
a(5, 6).
c(1, 2).
c(3, 4).
d(3).
```

can be evaluated bottom–up by interatively applying the rules and facts, respectively, in parallel as follows:

1. The first iteration derives the 4 facts.
2. The second iteration derives the 4 facts again plus the new fact

   ```
   b(3, 4).
   ```

   The new fact is derived using an instance of the second rule, whose body facts had been derived in iteration 1:

   ```
   b(3, 4) :- c(3, 4), d(3).
   ```

The first rule `a(X, Y) :- b(Y, X)` could not be used here, since we had no facts for `b` after iteration 1.

3. The third iteration derives the 5 previously derived facts again plus the new fact

   ```
   a(3, 4).
   ```

   The new fact is derived using an instance of the first rule, whose body fact had been derived in iteration 2:

   ```
   a(4, 3) :- b(3, 4).
   ```

Finally, now further new facts can be derived, and we have obtained the following 6 facts:

```
a(5, 6).  a(4, 3).
b(3, 4).
c(1, 2).  c(3, 4).
d(3).
```

Example with Default Negation

Given a DATALOG$^{\text{not}}$ program containing the facts for the database tables, one of the rules from above, and a further rule describing an employee `SSN` who works on a project `PNO` that belongs to a department `DNO` where the employee does not work:

```
employee_department(SSN, DNO) :-
   employee(_, _, _, SSN, _,_,_,_,_, DNO).
employee_works_on_foreign_project(SSN, PNO, DNO) :-
   works_on(SSN, PNO, _),
   project(_, PNO, _, DNO),
   not(employee_department(SSN, DNO)).
```

The default negation is represented as "`not`" in DATALOG*.
It should be *stratified*, which means here that `employee_department` can be evaluated before `employee_works_on_foreign_project`, since it does not transitively depend on it.

We use two iteration processes.

1. The first iteration process uses the database facts and the first rule and consists of three iteration steps. The first iteration step derives the database facts for `employee`, `works_on`, and `project`. The second iteration step derives the 8 facts for `employee_department`. The third iteration step does not derive new facts.

2. Now, the second iteration process can start. It uses the second rule and consists of two iteration steps. The first iteration step uses the instance

```
employee_works_on_foreign_project('333333333', 20, 1) :-
   works_on('333333333', 20, '15.0'),
   project('Reorganization' 20, 'Houston' 1),
   not(employee_department('333333333', 1)).
```

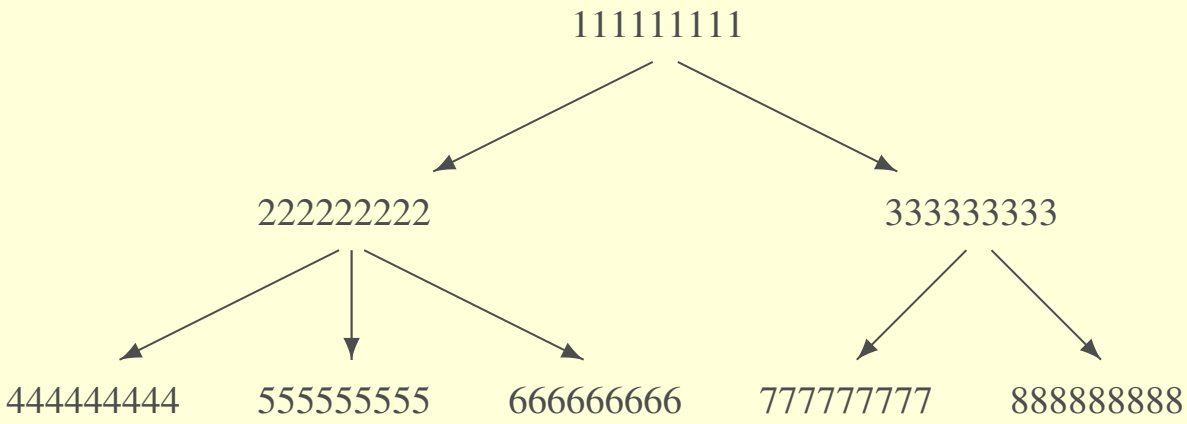
   for deriving the new head fact, since employee `'333333333'` does not work in department `1`. The second iteration step does not derive new facts.

If we would use the DATALOG^{not} program containing the facts for the database tables, together with the two mentioned rules in the same iteration process, then the second rule we would erroneously derive many facts including

```
employee_works_on_foreign_project('111111111', 20, 1).
employee_works_on_foreign_project('222222222', 2, 5).
employee_works_on_foreign_project('222222222', 3, 5).
employee_works_on_foreign_project('333333333', 20, 1).
employee_works_on_foreign_project('333333333', 30, 4).
...
```

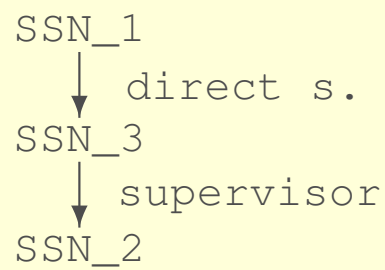
before the first rule can derive the correct facts for `employee_department`.

Example with Recursion: Transitive Closure



The following recursive DATALOG rule set derives the transitive supervisor relation on the social security numbers:

```
supervisor(SSN_1, SSN_2) :-
   direct_supervisor(SSN_1, SSN_2).
supervisor(SSN_1, SSN_2) :-
   direct_supervisor(SSN_1, SSN_3),
   supervisor(SSN_3, SSN_2).
```



Given a DATALOG program containing a rule describing how `direct_supervisor` can be derived from `employee`, the facts for the database table `employee`, and the two transitive closure rules from above:

```
direct_supervisor(SSN_1, SSN_2) :-
   employee(_,_,_, SSN_2, _,_,_,_, SSN_1, _),
   SSN_1 \= 'NULL'.
...
supervisor(SSN_1, SSN_2) :-
   direct_supervisor(SSN_1, SSN_2).
supervisor(SSN_1, SSN_2) :-
   direct_supervisor(SSN_1, SSN_3),
   supervisor(SSN_3, SSN_2).
```

1. The first iteration derives the facts for `employee`.
2. The second iteration derives the facts for `direct_supervisor` from the facts for `employee`:

```
direct_supervisor(111111111, 222222222).
direct_supervisor(111111111, 333333333).
direct_supervisor(222222222, 444444444).
direct_supervisor(222222222, 555555555).
direct_supervisor(222222222, 666666666).
direct_supervisor(333333333, 777777777).
direct_supervisor(333333333, 888888888).
```

3. The third iteration translates these facts to the corresponding 7 facts for `supervisor`.

```
supervisor(111111111, 222222222).
...
supervisor(333333333, 888888888).
```

4. The fourth iteration derives the 5 new facts that `111111111` is the transitive (indirect) supervisor of the employees `444444444` to `888888888`:

```
supervisor(111111111, 444444444).
supervisor(111111111, 555555555).
supervisor(111111111, 666666666).
supervisor(111111111, 777777777).
supervisor(111111111, 888888888).
```

Since the hierarchy is of limited depth 2 here, the relations corresponding to these facts could also be derived in SQL.

For arbitrary hierarchies of unlimited depth it was not possible to derive the transitive supervisors in standard SQL before SQL–99.

In principle, all rules can be used in all iterations. But, a rule can only fire and derive facts, as soon as facts for the body atoms have been derived in previous iterations. From then on, the rule can always be used to derive the same facts.

One of the purposes of *efficient* bottom–up evaluation is to avoid these redundant derivations – especially in the presence of recursion.

Finally, in iteration 4, the 5 facts for transitive supervisors are derived.
Iteration 5 does not derive any new facts.

Thus, a fixpoint is reached, and the iteration terminates.

Comparison with SQL

- Non–recursive DATALOG could be simulated in SQL by mapping the rules to `View` statements – or to `INSERT` statements whose result is computed using a `SELECT` statement.
- Recursion brings higher expressivity to DATALOG.
- There are DATALOG extensions which allow for default negation and aggregate operations as well.
- The rule–based approach of DATALOG supports modularization: instead of one single, complex `VIEW` statement in SQL, a set of simpler and more compact DATALOG rules can be used.

Transitive closure cannot be formulated in all relational database systems. Some systems, however, offer limited forms of recursion – cf. SQL:2003.

```
CREATE RECURSIVE VIEW supervisor(Emp, Sup) AS
   SELECT Emp, Sup
   FROM   direct_supervisor
   UNION
   SELECT D.Emp, S.Sup
   FROM   direct_supervisor D, supervisor S
   WHERE  D.Sup = S.Emp
```

This assumes a table `direct_supervisor` with the attributes `Emp` and `Sup`. Obviously, this SQL implementation is structurally equivalent to the following shorter rule implementation ("`;`" means "or").

```
supervisor(Emp, Sup) :-
   ( direct_supervisor(Emp, Sup)
   ; direct_supervisor(Emp, X), supervisor(X, Sup) ).
```

Course on Deductive Databases

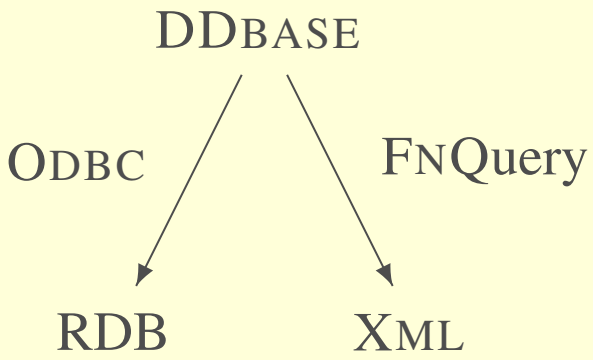
- foundations and applications of PROLOG and DATALOG, data modelling and programming;
- the deductive database system DDBASE;
- efficient evaluation of DATALOG programs;
- the extended language DATALOG*;
- further language constructs in the system Declare (formerly DDK, DISLOG Developers' Kit):
  - complex objects / data structures,
  - default negation and disjunction;
- applications on the basis of PROLOG and DDBASE.

2.3 The Deductive Database System DDBASE

The deductive database system DDBASE, which is part of Declare, combines PROLOG and DATALOG. It can process *hybrid knowledge bases* containing

- relational databases and
- XML documents

within the same query using ODBC and FNQuery, respectively:



This extends *database programming languages* (DBPL) by XML capabilities.

ODBC

The following PROLOG rule accesses a relational database – given by the connection handle `mysql` – using the ODBC library of SWI PROLOG.

```
generate_html_table(Salary, table:Rows) :-
   concat('SELECT fname, minit, lname, salary \
      FROM employee WHERE salary >= ', Salary, Query),
   Types = [types([atom,atom,atom,integer])],
   findall( Row,
      ( odbc_query(mysql, Query, row(F,M,L,S), Types),
        Row = tr:[td:[F], td:[M], td:[L], td:[S]] ),
      Rows ).
```

The query string `Query` is obtained by concatenating a partial select statement with the value for the salary. `Types` gives the types of the components of the result tuples.

The `findall` Statement

The call

```
findall(Template, Goal, Results)
```

to the meta–predicate `findall/3` computes

- the list `Results`
- of all terms `Template`
- that fullfil `Goal`.

- The call `odbc_query(mysql, Query, row(F,M,L,S), Types)` returns the values `F,M,L,S` for the attributes `fname, minit, lname, salary` of the table `employee`.
- By backtracking, the `findall` statement produces a list `Rows` of PROLOG terms `Row` of the form

  ```
  tr:[td:[F], td:[M], td:[L], td:[S]],
  ```

  which represent XML elements in FNQuery.
- For a given `Salary`, the call `generate_html_table(Salary, table:Rows)` produces a PROLOG term `table:Rows`, which represents the following HTML table in FNQuery.

The generated HTML table

```
<table>
   <tr><th>Fname</th><th>Minit</th><th>Lname</th><th>Salary</th></tr>
   <tr><td>John</td><td>B</td><td>Smith</td><td>30000</td></tr>
   <tr><td>Franklin</td><td>T</td><td>Wong</td><td>40000</td></tr>
   <tr><td>Jennifer</td><td>S</td><td>Wallace</td><td>43000</td></tr>
   <tr><td>Ramesh</td><td>K</td><td>Narayan</td><td>38000</td></tr>
   <tr><td>James</td><td>E</td><td>Borg</td><td>55000</td></tr>
</table>
```

can be rendered in a web browser:

Fname	Minit	Lname	Salary
John	B	Smith	30000
Franklin	T	Wong	40000
Jennifer	S	Wallace	43000
Ramesh	K	Narayan	38000
James	E	Borg	55000

By ODBC, we can make SQL tables available in DDBASE:

```
employee(A,B,C,D,E,F,G,H,I,J) :-
   ddbase_call(odbc(mysql),
      company:employee(A,B,C,D,E,F,G,H,I,J)).

works_on(A,B,C) :-
   ddbase_call(odbc(mysql), company:works_on(A,B,C)).
```

It is also possible to generate these rules in DDBASE, which avoids the error–prone, repeated use of many variable symbols.
The call `ddbase_connect(odbc(mysql), M, Database:Table)` asserts a corresponding rule in a PROLOG module `M`.

The following two aggregation statements refer to the PROLOG predicates `employee/10` and `works_on/3`; these could be stored in PROLOG or provided by ODBC.

In the second statement, the facts for `works_on/3` are derived using FNQuery from an XML document `works_on.xml`.

Aggregation Queries

The call

```
ddbase_aggregate(Template, Goal, Results)
```

to the meta–predicate `ddbase_aggregate/3` computes

- the list `Results`
- of all terms `Template`
- that fullfil `Goal`.

Terms with aggregation predicates, such as `sum, avg, min, max, list`, in `Template` are replaced by the corresponding aggregated values.

Binary aggregation predicates can be defined by the user.

The meta–predicate `ddbase_aggregate/3` in the following query groups over the employees:

- for every employee – given by `FNAME,MINIT,LNAME,SSN` –
- the corresponding list of all tuples `[PNO,HOURS]` is computed:

```
?- ddbase_aggregate( [F, M, L, SSN, list([PNO,HOURS])],
      ( works_on(SSN, PNO, HOURS),
        employee(F, M, L, SSN, _,_,_,_,_,_) ),
      Tuples ),
   Attributes =
      ['FNAME','MINIT','LNAME','SSN','[PNO,HOURS]'],
   xpce_display_table(Attributes, Tuples).
```

The result is a list of nested tuples; it can be displayed as a table in the XPCE extension of SWI PROLOG.

```
Tuples = [
  ['Ahmad', 'V', 'Jabbar', '888888888', [[10, 35.5], [30, 5.0]]],
  ['Alicia', 'J', 'Zelaya', '777777777', [[10, 10.0], [30, 30.0]]],
  ... ]
```

FNAME	MINIT	LNAME	SSN	[PNO,HOURS]
Ahmad	V	Jabbar	888888888	[[10, 35.5], [30, 5.0]]
Alicia	J	Zelaya	777777777	[[10, 10.0], [30, 30.0]]
Franklin	T	Wong	222222222	[[2, 10.0], [3, 10.0]]
James	E	Borg	111111111	[[20, 0.0]]
Jennifer	S	Wallace	333333333	[[20, 15.0], [30, 20.0]]
John	B	Smith	444444444	[[1, 32.5], [2, 7.5]]
Joyce	A	English	666666666	[[1, 20.0], [2, 20.0]]
Ramesh	K	Narayan	555555555	[[3, 40.0]]

Thus, DDBASE can produce NF^2 (NFNF: Non–First Normal Form) tables, which is not possible in SQL.

In DDBASE, we can define arbitary binary aggregation predicates.

`ddbase_aggregate/3` groups over all variable symbols that occur standalone in the result template, which here is `[F, M, L, SSN, list([PNO, HOURS])]`; these variable symbols are `F, M, L, SSN`.

- For every `F, M, L, SSN`, i.e., for every employee, the above call to `ddbase_aggregate/3` computes the list `Xs` of all corresponding pairs `[PNO, HOURS]`.
- Then, the call `list(Xs, Pairs)` to the user–defined aggregation predicate `list/2` simply passes `Xs` to `Pairs`.
- Thus, `ddbase_aggregate/3` produces a nested tuple `[F, M, L, SSN, Pairs]` for every employee. `Pairs` is a list of lists; it represents a relation.

The resulting list `Tuples` is the output.

Aggregation on RDB and XML

The following slightly modified statement aggregates the working hours of the employees of the departments:

```
?- ddbase_aggregate( [DNO, sum(HOURS)],
       ( employee(_,_,_, SSN, _,_,_,_,_, DNO),
         Row := doc('works_on.xml')/row::[@'ESSN'=SSN],
         H := Row@'HOURS', atom_number(H, HOURS) ),
       Tuples ).
Tuples = [[1, 0.0], [4, 115.5], [5, 140.0]]
```

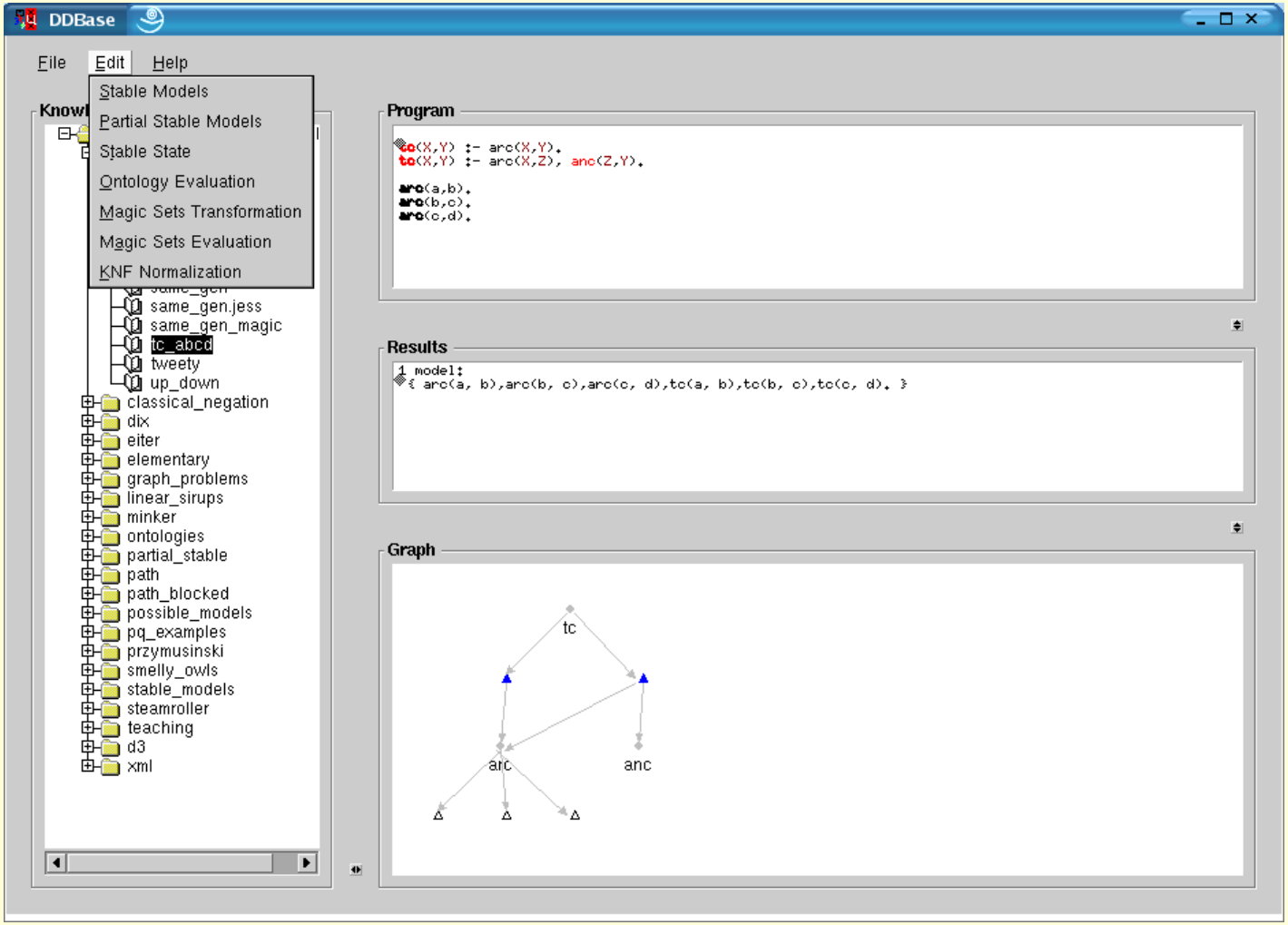
The attribute value `H` of the attribute `'HOURS'` of `Row` is an atom that has to be converted to a number `HOURS`.

The template `[DNO, sum(HOURS)]` leads to a grouping on the department numbers. For every `DNO`, first the list `Xs` of all corresponding `HOURS` is computed, and then the sum is computed by the call `sum(Xs, Sum)`; thus, we obtain a standard result tuple `[DNO, Sum]`.

A query optimizer of DDBASE could rearrange the Goal in `ddbase_aggregate/3`; this could be done by changing the order of the calls to the predicate `employee/10` provided by ODBC and the XML document `works_on.xml`.

- It might be the best to first completely load the table EMPLOYEE from the relational database to PROLOG using ODBC and to index it on the third argument position, which holds the `SSN`.
- Then, in a singe pass through the XML document, the working hours can be obtained using a path expression – a concept known from XPATH and XQUERY– in FNQuery from the XML rows, and the corresponding department numbers of the employees can be obtained using the index.

DDBASE offers a graphical IDE for DATALOG^{not} :



2.4 The Semantic Web Library ClioPatria

We can connect the logic programming language Prolog and Rdf using the tool ClioPatria of Swi–Prolog.

It provides some useful features, such as a preconfigured Http SparQL server.

Instructions can be downloaded at

```
http://cliopatria.swi-prolog.org/
    swish/pldoc/doc/home/swipl/src/ClioPatria/
    ClioPatria/web/help/Download.txt
```

The main predicate of CLIOPATRIA is `rdf/3`.

We determine the birth dates of people somebody knows by

```
known_with_birthdate(Me, She, Date) :-
    rdf(Me, foaf:knows, She),
    rdf(She, schema:birthDate, Date).
```

which is equivalent to the SPARQL query:

```
SELECT ?me ?she ?date
WHERE {
    ?me foaf:knows ?she .
    ?she schema:birthDate ?date
}
```

The result may be as follows:

```
?- known_with_birthdate(Me, She, Date).
Me = 'http://example.org/bob#me'
She = 'http://example.org/alice#me'
Date = literal(type(xsd:date, 1990-07-04') ;
...
```

Research Issues in Deductive Databases

- data and knowledge engineering
  - integrated development environments (IDE)
  - suitable extensions of DATALOG
- query optimization for DATALOG and its extensions
- integration with other technologies
  - multi–paradigm programming: declarative + imperative
  - integration with semantic web technologies
  - non–monotonic reasoning in artificial intelligence
- Big Data Analysis

3 Literature

Semantic Web

G. Antoniou, P. Groth, F. van van Harmelen, R. Hoekstra:
A Semantic Web Primer, (Third Edition), MIT Press, 2012.

J. Baumeister, D. Seipel:
Anomalies in Ontologies with Rules, Journal of Web Semantics: Science, Services and Agents on the World Wide Web 8 (2010), No. 1, pp. 55–68.

P. Hitzler, M. Kötzsch, S. Rudoph, Y. Sure:
Semantic Web, Springer, 2008.

S. Abiteboul, P. Buneman, D. Suciu:
Data on the Web: From Relations to Semistructured Data and XML,
Morgan Kaufmann, 2000.

Deductive Databases

J. Minker, D. Seipel, C. Zaniolo:
Logic and Databases: History of Deductive Databases,
in Handbook of Computational Logic, North Holland, 2014.

S.K. Das: *Deductive Databases and Logic Programming*,
Addison–Wesley, 1992.

S. Ceri, G. Gottlob, L. Tanca: *Logic Programming and Databases*,
Springer, 1990.

J.D. Ullman:
Principles of Database and Knowledge–Base Systems, Volume I/II,
Computer Science Press, 1988/89.

PROLOG

I. Bratko: PROLOG – *Programming for Artificial Intelligence,*
Addison–Wesley, 4th Edition, 2011.

W.F. Clocksin, C.S. Mellish: *Programming in* PROLOG*,*
Springer, 5th Edition, 2003.

J.W. Lloyd: *Foundations of Logic Programming,*
Springer, 2nd Edition, 1987.

Relational Databases

A. Kemper, A. Eickler: *Datenbanksysteme – Eine Einführung*.
6. Auflage, Oldenbourg, 2006.

R. Ramakrishnan, J. Gehrke: *Database Management Systems*.
3rd Edition, McGraw–Hill, 2004.

R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*.
3rd Edition, Benjamin Cummings, 2000.

C.J. Date, H. Darwen: *A Guide to the* SQL *Standard*.
4th Edition, Addison–Wesley, 1997.