The SILK Language

December 22, 2009

This version:
`$Id: SILK.tex 1241 2009-12-22 23:28:58Z mdean $`

Editor:
Mike Dean (BBN Technologies)

Authors:
Mike Dean (BBN Technologies)
Benjamin Grosof (Vulcan Inc.)
Michael Kifer (State University of New York at Stony Brook)

Issue 0.1: Copyright needs to be resolved, as long as significant portions are excerpted from SWSL. We may consider publishing this document under an open source license. □

Abstract

SILK (Semantic Inferencing for Large Knowledge) is an advanced knowledge representation language combining the latest work in theory, non-monotonic reasoning, business rules, and the Semantic Web. It is designed to be sufficiently expressive and scalable to support large challenges including Project Halo. This evolving document is the authoritative reference on the features, syntax, and semantics of SILK.

Status of this document

This is an Editor's Draft of a document being prepared for Vulcan Inc. It represents work in progress and is still rather raw and incomplete. A significant update is expected in Fall 2009. Please do not redistribute without permission of Benjamin Grosof at Vulcan.

Feedback on this document may be sent to silk-comments@semwebcentral.org.

Contents

Editor's Note 0.1: We should probably include a basic glossary of terms such as atom, axiom, body, constant, fact, formula, ground, head, literal, method, molecule, path expression, predicate, query, rule, rulebase, statement, term, variable, etc. The SWSL Glossary is not useful in this regard. We may also be able to consolidate some terms. □

Editor's Note 0.2: We can also create an index in LaTeX. □

1 Introduction

Semantic Inferencing with Large Knowledge (SILK) is an advanced knowledge representation language being developed as part of the Halo Advanced Research (HalAR) component of Vulcan Inc.'s Project Halo [Hal].

SILK includes a novel combination of features that hitherto have not been present in a single system. However, taken separately, almost all of the features of SILK have been implemented in either FLORA-2 [YKWZ08], SweetRules [GDG$^+$], or the commercial Ontobroker [Ont] system. Extensive feedback collected from the users of these systems has been incorporated in the design of SILK.

SILK is a rule-based language with non-monotonic semantics. Such languages are better suited for tasks that have programming flavor and that naturally rely on default information and inheritance. These tasks include service discovery, contracting, policy specification, and others. In addition, rule-based languages are quite common both in industry and research, and many people are more comfortable using them even for tasks that may not require defaults, such as service profile specification.

1.0.0.1 ***The layered structure of SILK.*** The features of SILK can be organized into several different *layers*. Unlike OWL, the layers are not based on the expressive power and computational complexity. Instead, each layer includes a number of new concepts that enhance the *modeling power* of the language. This is done in order to make it easier to learn the language and to help understand the relationship between the different features. Furthermore, most layers that extend the core of SILK are *independent* from each other — they can be implemented all at once or in any partial combination.

1.0.0.2 ***Complexity.*** Except for the equality layer, which boosts the complexity, all layers have the same complexity and decidability properties. For SILK, the most important reasoning task is *query answering*. The general problem of query answering is known to be only semi-decidable. However, there are large classes of problems that are decidable in polynomial time. The best-known, and perhaps the most useful, subclass consists of rules that do not use function symbols. However, many decidable classes of rules *with* function symbols are also known [NS97].

1.1 Acknowledgments

SILK was derived from the Semantic Web Services Language [BBB$^+$05]. The contributions of the Semantic Web Services Initiative Language Committee are gratefully acknowledged.

We received very helpful comments and corrections from other members of the HalAR team, including Paul Haley and Ken Murray.

1.2 Typography

This document distinguishes several types of sections that may be of varying interest to different audiences:

- **Example:** Contains a SILK language fragment.
- **Editor's Note:** Identifies an outstanding editorial issue.
- **Issue:** Identifies an outstanding design issue.
- **Rationale:** Discusses why certain design decisions were made.
- **Semantics:** Provides a formal description of a SILK language feature.

Editor's Note 1.1: MK: We probably don't need a special environment for the semantics. At least, not for the cases when the whole section is dedicated to the semantics. □

Editor's Note 1.2: The current macros renumber these items when new items are inserted. This is likely to be a problem at least for issue tracking. □

Editor's Note 1.3: We want to distinguish firm vs. "squishy" (underspecified or at risk) parts of the specification, and be able to easily generate the firm subset. This could involve reorganizing the document as well as additional markup. □

2 The SILK Language

This section describes the SILK language, including its features, syntax, and semantics.

Editor's Note 2.1: Consider dividing this section into multiple sections. □

Editor's Note 2.2: Adapt and incorporate examples from sources such as the FLORA-2 tutorial and possibly Ontoprise's F-Logic Tutorial. □

2.1 Overview of SILK

The SILK language is designed to provide support for a variety of tasks that range from service profile specification to service discovery, contracting, policy specification, and so on. The language is layered to make it easier to learn and to simplify the use of its various parts for specialized tasks that do not require the full expressive power of SILK. The layers of SILK are shown in Figure 1.

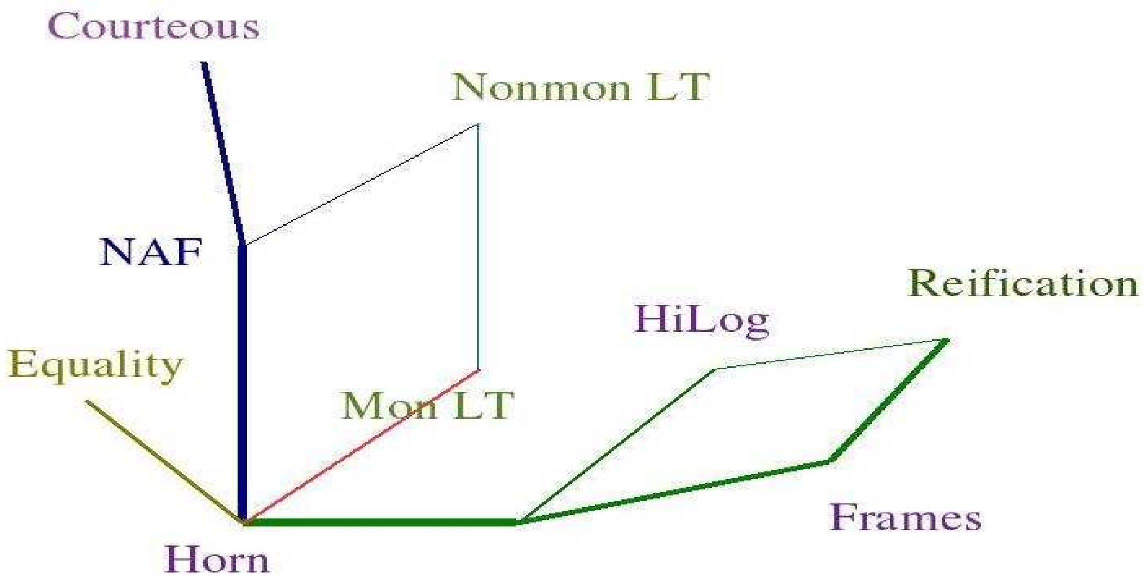

Figure 1: The Layered Structure of SILK

The core of the language consists of the pure *Horn* subset of SILK. The *monotonic Lloyd-Topor* (Mon LT) extension [Llo87] of the core permits disjunctions in the rule body and conjunction and implication in the rule head. *NAF* is an extension that allows negation in the rule body, which is interpreted as negation-as-failure. More specifically, negation is interpreted using the so called *well-founded semantics* [VRS91]. The *nonmonotonic Lloyd-Topor* extension (Nonmon LT) further permits quantifiers and implication in the rule body. The *Courteous rules* [Gro99, WGK$^+$09] extension introduces two new features: restricted classical negation and prioritized rules. *HiLog* and *Frames* extend the language with a different kind of ideas. HiLog [CKW93] enables a high degree of meta-programming by allowing variables to range over predicate symbols, function symbols, and even formulas. Despite these second-order features, the semantics of HiLog remains first-order and tractable. It has been argued [CKW93] that this semantics is more appropriate for many common tasks in knowledge representation than the classical second-order semantics. The *Frames* layer of SILK introduces the most common object-oriented features, such as the frame syntax, types, and inheritance. The syntax and semantics of this extension is inspired by F-logic [KLW95], and subsequent works [FLU94, YK03a, YK03b]. Finally, the *Reification* layer provides a mechanism for making objects out of a large class of SILK formulas, which puts such formulas into the domain of discourse and allows reasoning about them.

All of the above layers have been implemented in one system or another and have been found highly valuable in knowledge representation. For instance, FLORA-2 [YKWZ08] includes all layers except Courteous rules and Nonmonotonic Lloyd-Topor. SweetRules [GDG+] supports Courteous extensions, and Ontobroker [Ont] supports Nonmonotonic Lloyd-Topor and frames.

Four points should be noted about the layering structure of SILK.

1. The lines in Figure 1 represent inclusion dependencies among layers. For instance, the Nonmonotonic LT layer includes both NAF and Monotonic LT. Reification includes HiLog and Frames, Courteous includes NAF, etc.

2. The different branches of Figure 1 are orthogonal and they *all* can be combined. For instance, the Frames and HiLog layers can be combined with the Courteous and Nonmon LT layers. Likewise, the equality layer can be combined with any other layer. Thus, SILK is a unified language that combines all the layers into a coherent and powerful knowledge representation language. Its semantics will be described in a separate document. The semantics of the individual layers has been described in other publications and an overview of the overall semantics is provided in Section 2.17.

3. The Lloyd-Topor extensions and the Courteous rules extensions endow SILK with all the normal first-order connectives. Therefore, *syntactically* SILK contains all the connectives of full first-order logic. However, *semantically* SILK is incompatible with first-order logic. Their semantics agree only on a relatively small, but useful, subset of Horn rules.

4. Because of its non-monotonic flavor, SILK distinguishes between connectives with the classical first-order semantics and connectives that have non-monotonic semantics. For instance, it uses two different forms of negation —`naf`, for negation-as-failure, and `neg`, for classical negation. Likewise, it distinguishes between the classical implication, `<==` and `==>`, and the if-then connective `:-`used for rules.

2.2 Basic Definitions

In this section we define the basic syntactic components that are common to all layers of SILK. Additional syntax will be added as more layers are introduced.

SILK employs basic syntactic constructs from the W3C Rule Interchange Format (RIF) syntactic framework [BK08]. RIF supports XML data types, IRIs, etc., in a general way.

Issue 2.1: Compare SILK constants to those in the RIF Last Call Working Drafts. □

A *constant* is either a *numeric value*, a *symbol*, a *string*, or a *URI*.

- A ***numeric value*** is either an integer, a decimal, or a floating point number. For instance, `123`, `34.9`, `45e-11`. See Section 2.16 for more details on the relationship between SILK data types and the primitive data types in XML Schema [BM04].

- A ***symbol*** is a string of characters enclosed between a pair of single quotes. For instance, `'abc#$%'`. Single quotes that are part of a symbol are escaped with the backslash. For instance, the symbol `a'bc''d` is represented as `'a''bc\'\'d'`. The backslash is escaped with another backslash. Symbols that consist exclusively of alphanumeric characters and the underscore (_) and begin with a letter or an underscore do not need to be quoted.

- ***Strings*** are sequences of characters that are enclosed between a pair of double quote symbols, e.g., `"ab'%#cd"`. A double quote symbol that occurs in a string must be escaped with the backslash. For instance, the string `ab"cd"""gf` is represented as `"ab\"cd\"\"\"gf"`.

- A Uniform Resource Identifier (*URI*) can come either in the form of a *full URI* or in the abbreviated form of a *compact URI (CURIE)*.

  A ***full URI*** is a sequence of characters that has the form of a URI, as specified by IETF [BLFM05], and is enclosed between angle brackets. For instance, `<http://w3.org/>`.

  Issue 2.2: The use of the `<...>` notation is problematic, since we want to also use `<,>` for infix comparison operators. □

  A ***CURIE*** [BM08] has the form *prefix*:*local-name*. Here *prefix* is an alphanumeric symbol that is defined to be a shortcut for a URI as specified below; *local-name* is a string that must be acceptable as a path component in a URI. A CURIE is treated as a macro that expands into a full URI by concatenating the expansion of *prefix* (the URI represented by the prefix) with *local-name*.

Issue 2.3: CURIEs are actually based on IRIs [DG05] rather than URIs. We should consider using IRIs throughout SILK, as RIF does. □

A ***prefix declaration*** is a statement of the form

```
  :- prefix prefix-name = <URI> ;
```

The prefix can then be used instead of the URI in CURIEs. For instance, if we define

```
:- prefix w3 = <http://www.w3.org/TR/> ;
```

then the SILK URI `<http://www.w3.org/TR/xquery>` is considered to be the same as `w3:xquery`. Prefix declarations are treated as nothing more than macros and macro-expansion is expected to be done prior to any syntactic or semantic considerations (such as considering whether two SILK expressions are identical).

The following prefixes are pre-defined by SILK and do not need to be declared (but may be overriden by an explicit prefix declaration):

```
:- prefix silk = <http://vulcan.com/2008/silk#> ;
:- prefix silkb = <http://vulcan.com/2008/silk-builtins#> ;
:- prefix rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#> ;
:- prefix rdfs = <http://www.w3.org/2000/01/rdf-schema#> ;
:- prefix owl = <http://www.w3.org/2002/07/owl#> ;
:- prefix xsd = <http://www.w3.org/2001/XMLSchema#> ;
```

A ***variable*** is an alphanumeric symbol (plus the underscore), which is prefixed with the `?`-sign. Examples: `?_`, `?abc23`.

A ***first-order term*** is either a constant, a variable, or an expression of the form $\mathtt{t(t_1,\ldots,t_n)}$, where `t` is a constant, $\mathtt{t_1,\ldots,t_n}$ are first-order terms, and `n > 0`. Here the constant `t` is said to be used as a ***function symbol*** (or a ***functor***) and $\mathtt{t_1,\ldots,t_n}$ are used as **arguments**. Variable-free terms are also called ***ground***. The set of all ground terms is known as the ***Herbrand universe***.

Following Prolog, we also introduce special notation for lists: $\mathtt{[t_1,\ldots,t_n]}$ and $\mathtt{[t_1,\ldots,t_n \,|\, rest]}$, where $\mathtt{t_1,\ldots,t_n}$ and `rest` are first-order terms. The first form shows all the elements of the list explicitly and the latter shows explicitly only a prefix of the list and uses the first-order term `rest` to represent the tail. We should note that, like in Prolog, this is just a convenient shorthand notation. Lists are nothing but first-order terms that are representable with function symbols. For instance, if `cons` denotes a function symbol that prepends a term to the head of a list then `[a,b,c]` is represented as the first-order term `cons(a,cons(b,c))`.

A ***first-order atomic formula*** has the same form as a first-order term except that a variable cannot be a first-order atomic formula. We do not distinguish predicates as a separate class of constants, as this is usually not necessary, since first-order atomic formulas can be distinguished from first-order terms by the context in which they appear.

As many other rule-based languages, SILK has a special ***unification operator***, denoted `=`. The unification operator is always interpreted as an identity relation over the Herbrand universe. Therefore, a formula of the form

$$term_1 \texttt{ = } term_2$$

where both terms are ground, is true if and only if the two terms are identical. Since the semantics of the unification operator is fixed and is the same for all rulebases, it *cannot* appear in the head of a rule.

The `=` predicate is related to the equality predicate, `:=:`, which is introduced by the Equality Layer, Section 2.11.

To test that two terms cannot be identical (do not unify), SILK uses the ***disunification*** operator `!=`. It is interpreted as the negation of `=` so, for ground terms, $term_1$ `!=` $term_2$ iff the two terms are not identical.

Terms can also be connected by infix arithmetic (`+`, `-`, `*`, and `/`) and comparison (`<`, `<=`, `>=`, and `>`) operators.

A ***conjunctive formula*** is either an atomic formula or a formula of the form

atomic formula `and` *conjunctive formula* where `and` is a conjunctive connective. Here and henceforth in similar definitions, italicized words will be meta-symbols that denote classes of syntactic entities. For instance, *atomic formula* above means "any atomic formula." An ***and/or-formula*** is either a conjunctive formula or a formula of either of the forms

conjunctive-formula `or` *and/or-formula* *and/or-formula* `and` *and/or-formula*

In other words, an and/or formula is an arbitrary Boolean combination of atomic formulas using the connectives `and` and `or`.

To disambiguate the scope of connectives in and/or formulas, the user should use parentheses. When parentheses are not given, SILK assumes that `and` takes precedence over `or`.

Comments. SILK has two kinds of comments: single line comments and multiline comments. The syntax is the same as in Java. A ***single-line comment*** is any text that starts with a `//` and continues to the end of the current line. If `//` starts within a string (`"..."`) or a symbol (`'...'`) then these characters are considered to be part of the string or the symbol, and in this case they do not start a comment. A ***multiline comment*** begins with `/*` and ends with a matching `*/`. The combination `/*` does not start a comment if it appears inside a string or a symbol.

Issue 2.4: It is even more important to have structured comments that are captured in the rule representation and accessible to other tools. Consider use of a Javadoc-like syntax and/or RIF-like annotations.

MK: compare to RIF annotations. □

2.3 Horn Rules

A ***Horn rule*** has the form

```
  head :- body ;
```

where *head* is an atomic formula and *body* is a conjunctive formula.

Rules can be recursive, i.e., the predicate in the head of a rule can occur (with the same arity) in the body of the rule; or they can be mutually recursive, i.e., a head predicate can depend on itself through a sequence of rules.

All variables in a rule are considered *implicitly* quantified with `forall` outside of the rule, i.e., `forall ?X,?Y,...(head :- body)`. A variable that occurs in the body of a rule but not its head can be equivalently considered as being implicitly existentially quantified in the body. For instance,

```
forall ?X,?Y (p(?X) :- q(?X,?Y)) ;
```

is equivalent to

```
forall ?X (p(?X) :- exist ?Y (q(?X,?Y))) ;
```

Semantics: The semantics of a set of Horn rules can be characterized in several different ways: through the regular first-order entailment, as a minimal model (which in this case happens to be the intersection of all Herbrand models of the rule set) and as a least fixpoint of the immediate consequence operator corresponding to the rule set [Llo87]. □

2.4 Queries

A ***query*** has the form

```
?- query ;
```

where *query* is syntactically equivalent to a rule body. A query returns a result set, which is a set of binding lists. Each binding includes a variable and a typed value.

SILK also supports queries of external data sources (Section 2.18.1) and aggregation (Section 2.15).

2.4.1 Persistent Queries

SILK also supports ***persistent queries*** that incrementally return new results as they become available.

A persistent query is defined as an instance of `silk:PersistentQuery` with members `query` and optionally `action`, both of which are strings containing SILK statements. `action` is executed with variable bindings from `query`. If no `action` is specfied, `query` and its variable bindings are printed to stdout.

Example 2.1:

```
query1 # silk:PersistentQuery[query->"?- ?person # Person[name->?name] ;",
                                 action->"silkb:writeLn(?name)"] ;
```

□

2.5 The Monotonic Lloyd-Topor Layer

This layer extends the syntax of the Horn layer with three kinds of syntactic sugar:

1. Disjunction in the body of the rule
2. Conjunction in the head of the rule
3. Classical implication

A ***classical implication*** is a statement of either of the following forms:

$formula_1$ `==>` $formula_2$
$formula_1$ `<==` $formula_2$

The ***Lloyd-Topor implication*** (LT implication) is a special case of the classical implication where the formula in the head is a conjunction of atomic formulas and the formula in the body can contain both conjunctions and disjunctions of atomic formulas.

If classical implication holds in both directions between two formulas, their logical equivalence can be expressed in a single statement using ***classical bi-implication*** of the following form:

```
formula1 <==> formula2
```

The ***Lloyd-Topor bi-implication*** (LT bi-implication) is a special case of the classical bi-implication where both formulas are conjunctions of atomic formulas.

The monotonic LT layer extends Horn rules in the following way. A rule still has the form

```
head :- body ;
```

but *head* can now be a conjunction of atomic formulas and/or LT implications (including bi-implications) and *body* can consist of atomic formulas combined in arbitrary ways using the `and` and `or` connectives.

Semantics: The monotonic LT extensions are strictly syntactic: the semantics of rules using these extensions remains strictly classical first-order since they are transformed to rules that do not contain the monotonic LT extensions. Under the classical first-order semantics, these transformations are known to preserve equivalence, and this fact motivates their use in logic programming and SILK. □

The ***monotonic Lloyd-Topor transformations*** are listed below.

- `head :- body1 or body2` reduces to

  ```
  head :- body1 ;
  head :- body2 ;
  ```

- `head1 and head2 :- body` reduces to

  ```
  head1 :- body ;
  head2 :- body ;
  ```

- `(head1 <== head2) :- body` reduces to

  ```
  head1 :- head2 and body ;
  ```

- `(head1 ==> head2) :- body` reduces to

  ```
  head2 :- head1 and body ;
  ```

Complex formulas in the head are broken down using the last three reductions. Rule bodies that contain both disjunctions and conjunctions are first converted into disjunctive normal form and then are broken down using the first reduction rule.

Issue 2.5: Can SILK also allow if-and-only-if rules of the form `head <==> body` as syntactic sugar for the 2 corresponding rules? □

2.6 The NAF Layer

The NAF layer adds the negation-as-failure symbol, `naf`, in the rule body. For instance,

```
p(?X,?Y) :- q(?X,?Z) and naf r(?Z,?Y) ;
p(?X,?Y) :- q(?X,?Z) and naf (s(?Z,?Y) or q(?Y)) ;
```

More precisely, if φ is a subformula that is allowed to appear in the rule body, then `naf(`φ`)` is also an allowed subformula in the rule body. When φ is an atomic formula then no parentheses are required.

Semantics: In SILK we adopt the ***well-founded semantics*** [VRS91] as a way to interpret negation as failure. This semantics has good computational properties when no first-order terms of arity greater than 0 are involved, and the well-founded model is always defined and is unique. This model is three-valued, so some facts may have the "unknown" truth value. □

We should note one important convention regarding the treatment of variables that occur under the scope of `naf` and that do not occur anywhere outside of `naf` in the same rule. The well-founded semantics was defined only for ground atoms and the interpretation of unbound variables was left open. Therefore, if `?X` does not occur elsewhere in the rule then the meaning of

```
... :- ... and naf r(?X) and ...
```

can be informally defined as either

```
... :- ... and exist ?X (naf r(?X)) and ...
```

(i.e., `naf r(?X)` is true if `naf r(t)` is true for *some* ground term `t`) or as:

```
... :- ... and forall ?X (naf r(?X)) and ...
```

where `naf r(?X)` is true if `naf r(t)` is true for *all* ground terms `t`. In practice, the second interpretation is preferred, and this is also a convention used in SILK.

Issue 2.6: MK: this is a very error-prone convention in practice (turned out to be a poor decision in FLORA-2–also because it does not support explicit quantifiers). We should require explicit `naf exists` here and give an error of an unbound variable is encountered during evaluation. □

2.7 The Nonmonotonic Lloyd-Topor Layer

This layer introduces explicit bounded quantifiers (both `exist` and `forall`), classical implication symbols, `<==` and `==>`, and the bi-implication symbol `<==>` in the rule body. This essentially permits arbitrary first-order-looking formulas in the body of SILK. We say "first-order-looking" because it should be kept in mind that the semantics of SILK is *not* first-order and, for example, classical implication `A <== B` is interpreted in a non-classical way: as `(A or naf B)` rather than `(A or neg B)` (where `neg` denotes classical negation).

Recall that without explicit quantification, all variables in a rule are considered *implicitly* quantified with `forall` outside of the rule, i.e., `forall ?X,?Y,... (`***`head`***` :- `***`body`***`)`. A variable that occurs in the body of a rule but not its head can be equivalently considered as being implicitly existentially quantified in the body. For instance,

```
forall ?X,?Y (p(?X) :- q(?X,?Y)) ;
```

is equivalent to

```
forall ?X (p(?X) :- exist ?Y (q(?X,?Y))) ;
```

In the scope of the `naf` operator, unbound variables have a different interpretation under negation as failure. For instance, if `?X` is bound and `?Y` is unbound then

```
p(?X) :- naf q(?X,?Y) ;
```

means

```
forall ?X (p(?X) :- naf exist ?Y (q(?X,?Y))) ;
```

If we allow explicit universal quantification in the rule bodies then implicit existential quantification is not enough and an explicit existential quantifier is needed. This is because `forall` and `exist` do not commute and so, for example, `forall ?X (exist ?Y ...)` and `exist ?Y (forall ?X ...)` mean different things. If only implicit existential quantification were available, it would not be possible to differentiate between these two forms.

Formally, the Nonmonotonic Lloyd-Topor layer permits the following kinds of rules. The rule ***heads*** are the same as in the monotonic LT extension. The rule ***bodies*** are defined as follows.

- Any atomic formula is a legal rule body
- If f and g are legal rule bodies then so are
  - f `and` g
  - f `or` g
  - `naf` f
  - f `==>` g

  - f `<==` g
  - f `<==>` g

- If f is a legal rule body then so is

  - `exist ?X`$_1$`,...,?X`$_n$`(`f`)`

  where `?X`$_1$, ..., `?X`$_n$ are variables that occur *positively* (defined below) in f.

- If g_1, g_2 are legal rule bodies then

  - `forall ?X`$_1$`,...,?X`$_n$`(`g_1 `==>` g_2`)`
  - `forall ?X`$_1$`,...,?X`$_n$`(`g_2 `<==` g_1`)`

  are legal rule bodies provided that `?X`$_1$, ..., `?X`$_n$ occur *positively* in g_1

Positive occurrence of a variable in a formula is defined as follows:

- Any variable occurs positively in an atomic formula
- A variable occurs positively in f `and` g iff it occurs positively in either f or g.
- A variable occurs positively in f `or` g iff it occurs positively in both f and g.
- A variable occurs positively in f `==>` g iff it occurs positively in g.
- A variable occurs positively in f `<==` g iff it occurs positively in f.
- A variable occurs positively in f `<==>` g iff it occurs positively in both f and g.
- A variable occurs positively in `exist ?X`$_1$`,...,?X`$_n$`(`f`)` or `forall ?X`$_1$`,...,?X`$_n$`(`f`)` iff it occurs positively in f.

Editor's Note 2.3: Need rationale here. □

Issue 2.7: Explicit use of universal and, especially, mixed quantifiers is very error-prone. We might change this syntax to replace `forall` with the subset operator or something like that. □

Semantics: The semantics of Lloyd-Topor extensions is defined via a transformation into the NAF layer as shown below. As with monotonic Lloyd-Topor transformations, the nonmonotonic transformations are inspired by the fact that they are known to preserve equivalence under the classical first-order semantics (if `naf` is interpreted as classical negation). Further discussion of these transformations can be found in [Llo87]. □

2.7.0.1 ***Lloyd-Topor transformations.*** These transformations are designed to eliminate the extended forms that may occur in the bodies of the rules and reduce the rules to the NAF layer. These extended forms involve the various types of implication and the explicit quantifiers. Note that the rules, below, must be applied top-down, that is, to the conjuncts that appear directly in the rule body. For instance, if the rule body looks like

```
... :- ... and
       ((forall ?X (exist ?Y (foo(?Y,?Y) ==> bar(?X,?Z))))
        <== foobar(?Z))
       and ...
```

then one should first apply the rule for `<==`, then the rules for `forall` should be applied to the result, and finally the rules for `exist` and `==>`.

- Let the rule be of the form

  head `:-` $body_1$ `and (`f `==>` g`) and` $body_2$ `;`

  Then the LT transformation replaces it with the following pair of rules:

  head `:-` $body_1$ `and naf` f `and` $body_2$ `;`
  head `:-` $body_1$ `and` g `and` $body_2$ `;`

  The transformations for `<==` and `<==>` are similar.

- Let the rule be

  head `:-` $body_1$ `and forall` ?X_1,...,?X_n(g_1 `==>` g_2) `and` $body_2$
  `;`

  where ?X_1,...,?X_n are free variables that occur positively in g_1.

  The LT transformation replaces this rule with the following pair of rules, where q(?X'$_1$,...,?X'$_n$) is a new predicate of arity n and ?X'$_1$,...,?X'$_n$ are new variables:

  head `:-` $body_1$ `and naf` q(?X'$_1$,...,?X'$_n$) `and` $body_2$ `;`
  q(?X_1,...,?X_n) `:-` g_1 `and naf` g_2 `;`

  The transformation for `<==` is similar.

- Let the rule be

  head `:-` $body_1$ `and exist` ?X_1,...,?X_n(f) `and` $body_2$ `;`

  where ?X_1,...,?X_n are free variables that occur positively in f.

  The LT transformation replaces this rule with the following:

  head `:-` $body_1$ `and` f' `and` $body_2$ `;`

where `f'` is `f` with the variables `?X`$_1$`,...,?X`$_n$ consistently renamed into new variables. That is, explicit existential quantification can be replaced in this case with implicit quantification.

The above transformations reflect the difference between `naf` and `neg` in the classical tautologies `(f ==> g) <==> (neg f or g)` and `forall ?X (f) <==> neg exist neg ?X (f)` including that `naf p(?X)` is interpreted as `forall ?X (naf p(?X))` when `?X` does not occur elsewhere in the formula, as discussed in Section 2.6.

Editor's Note 2.4: The `neg exist neg` example above does not seem to be syntactically valid. □

2.8 The Courteous Layer

The courteous layer introduces *prioritized conflict handling*. Four new features are introduced into the syntax:

- ***classical negation*** of atomic formulas;
- a ***prioritization predicate***, which may be used to specify that some rules take precedence over other rules in the event of conflict;;
- ***rule labels***, which declare terms used to reference rules within the prioritization predicate;
- exclusions, which specify the scope of what constitutes conflict.

The theory behind the courteous logic programs is described in [Gro99, WGK$^+$09]. The courteous layer builds upon the NAF layer of SILK, described in Section 2.6.

2.8.1 Rule Labels

Each rule has an optional label, which is used for specifying prioritization in conjunction with the prioritization predicate (below). The syntactic form of a rule label is a term enclosed by a pair of braces: {...}. Thus, a ***labeled rule*** has the following form:

```
{label} head :- body ;
```

A *label* is a term, which may have variables. If so, these variables are interpreted as having the same scope as the implicitly quantified variables appearing in the rule expression. E.g., in the rule

```
{specialoffer(?X)} pricediscount(?X,10) :- loyalcustomer(?X) ;
```

the label `specialoffer(?X)` names the instance of the rule corresponding to the instance `?X`. However, the label term may not itself be a variable, so the following is illegal syntax:

```
{?X} pricediscount(?X,10) :- loyalcustomer(?X) ;
```

In general, labels are not unique; two or more rules (or instances of rules) may have the same label. However, sometimes it may be necessary to have unique rule labels.

Issue 2.8: MK: {label} may be hard to distinguish and parse, especially if we extend the syntax to constraints, aggregates, etc., which also use braces. I propose @{label}. Moreover, we should expand this so that other annotations (meta-info) could be specified inside @{...} after the label. □

2.8.2 Classical Negation

The classical negation connective, `neg`, may appear within the head and/or the body of a rule. Its scope is restricted to be an atomic formula, however. For example:

```
neg boy(?X) :- humanchild(?X) and neg male(?X) ;
{t14(?X,?Y)} p(?X,?Y) :- q(?X,?Y) and naf neg r(?X,?Y) ;
```

However, the following example is illegal syntax because `neg` negates a non-atomic formula.

```
u(?X) :- t(?X) and neg naf s(?X) ;
```

2.8.3 Prioritization Predicate

The prioritization predicate `silk:overrides` specifies precedence between rule labels, and thus between the rules labeled by those rule labels. The name of the prioritization predicate is syntactically reserved such that its name cannot be used for any other predicate.

A statement `silk:overrides(label1,label2)` indicates that the first argument, `label1`, has higher priority than the second argument, `label2`. For example, consider the following:

```
{rep} neg pacifist(?X) :- republican(?X) ;
{qua} pacifist(?X) :- quaker(?X) ;
{pri1} silk:overrides(rep,qua) ;
```

Here, the prioritization atom `silk:overrides(rep,qua)` specifies that `rep` has higher priority than `qua`. Continuing this example, suppose the following facts are also given:

```
{fac1} republican(nixon) ;
{fac2} quaker(nixon) ;
```

Then, under the courteous semantics, the literal `neg pacifist(nixon)` is entailed as a conclusion, and the literal `pacifist(nixon)` is *not* entailed as a conclusion, because the rule labeled `rep` takes precedence over the rule labeled `qua`.

Editor's Note 2.5: This is the only section that discusses literals. Can we substitute term? □

The prioritization predicate `silk:overrides`, while its name is syntactically reserved, is otherwise an ordinary predicate — it can appear freely in rules in the head and/or body. This may be useful for reasoning about the prioritization ordering.

Editor's Note 2.6: Cornsilk uses 2 user-level prioritization predicates, _overrides/2 and _overrides/4, which map to an underlying predicate. □

2.8.4 Exclusion

Rationale: This section formerly referred to mutexes for mutual exclusion, but has subseqently been generalized to work with k-ary exclusions. □

Editor's Note 2.7: Incorporate or replace with recent work on k-ary mutexes / exclusions. Presumably this includes Cornsilk's _opposes. □

The scope of what constitutes conflict is specified by mutual exclusion (mutex) statements, which are part of the rulebase and can be viewed as a kind of integrity constraint. Each such statement says that it is contradictory for a particular pair of literals (known as the "opposers") to be inferred, if an optional condition (known as the "given") holds true. The courteous LP semantics enforce that the set of sanctioned conclusions respects (i.e., is consistent with) all the mutexes within the given rulebase. Common uses for mutexes include specifying that two unary predicates are disjoint, or that a relation is functional; examples of these uses are given below.

An *unconditional* mutex has the following syntactic form:

```
 !- lit1 and lit2 ;
```

where `lit1` and `lit2` are classical literals. Intuitively, this statement means that it is a contradiction to derive both `lit1` and `lit2`. For example:

```
!- pricediscount(?CUST,5) and pricediscount(?CUST,10) ;
```

says that it is a contradiction to conclude that the discount offered to the same customer `?CUST` is both `5` and `10`. As another example,

```
!- lion(?X) and elephant(?X) ;
```

specifies that it is a contradiction to conclude that the same individual is both a lion and an elephant.

A *conditional* mutex has the following syntactic form:

```
 !- lit1 and lit2 | condition ;
```

Here *`condition`* is syntactically similar to a rule body, and `lit1` and `lit2` are classical literals. The symbol "|" separates the opposing literals from the condition. For example:

```
!- pricediscount(?CUST,?Y) and pricediscount(?CUST,?Z) | ?Y!=?Z ;
```

says that it is a contradiction to conclude that the discount offered to the same customer, `?CUST`, is both `?Y` and `?Z` *if* `?Y` and `?Z` are distinct values. This means that the relation `pricediscount` is functional.

Courteous LP also assumes that there is an implicit mutex between each atom *`A`* and its classical negation `neg` *`A`*. This implicit mutex is also known as a "classical" mutex.

Issue 2.9: Mutexes should be replaced with the more flexible `silk:opposes` predicate as is currently in the FLORA-2 prototype. □

2.8.5 Cancellation

Editor's Note 2.8: Describe the `silk:cancel` predicate and semantics. Discuss positive exceptions here or in a separate section. Legislative or regulatory examples may be appropriate. □

2.8.6 Omni-directional rule sets

See 3.4.

2.9 The Production Layer

Some SILK rules will need to interact with other computing environments to obtain data, perform calculations, or effect actions. This section describes such extra-logical features that are commonly found in production rule systems.

2.9.1 Procedural Attachments

External procedures can be invoked as SILK predicates or functions. They are distinguished from other predicates and functions by the presence of a binding signature. The binding signature indicates the name, type, and binding requirement of each argument.

Binding signatures are represented in SILK. An external predicate is declared to be of type `silk:ExternalPredicate`, which has frame syntax members `silk:arg`, `silk:binding`, and optionally `silk:vararg`.

An external function is declared to be of type `silk:ExternalFunction` and has the same members as `silk:ExternalProcedure` plus the additional member `silk:returnValue`.

`silk:arg` takes a `silk:Argument`, which has the member `silk:type`. The name of the instance is the name of the argument.

`silk:binding` takes a (typically skolem) `silk:Binding`, which has members `silk:in` and `silk:out`, which refer to argument names, and `silk:javaClass` (other such implementation members may be added later), which indicates the Java class that implements the external procedure predicat eor funtion.

The following example shows an external predicate with multiple binding patterns:

Example 2.2:

```
swrlb:anyURI # silk:ExternalPredicate[
silk:arg->{
uri[silk:type->xsd:string],
protocol[silk:type->xsd:string],
domain[silk:type->xsd:string],
port[silk:type->xsd:string],
part1[silk:type->xsd:string],
part2[silk:type->xsd:string],
fragment[silk:type->xsd:string]
},
silk:binding -> {anyURIBp1[
silk:in->{protocol,domain,port,part1,part2,fragment},
silk:out->uri,
silk:javaClass->"org.daml.swrl.jena.builtins.uri.AnyURI"
],
anyURIBp2[
silk:in-> uri,
silk:out->{protocol,domain,port,part1,part2,fragment},
silk:javaClass->"org.daml.swrl.jena.builtins.uri.AnyURI"
]}
] ;
```

□

These classes and predicates are defined in SILK as:

```
<>[owl:versionInfo->"$Id: bindingpatterns.silk 1117 2009-11-17 19:37:31Z mdean $",
   rdfs:comment->"SILK external procedure binding patterns"] ;

silk:AttachedProcedure[silk:arg => silk:Argument,
       silk:vararg {0:1} => silk:Argument, // variable argument, at end
                        silk:binding {1:*} => silk:BindingPattern] ;

  silk:ExternalPredicate ## silk:AttachedProcedure ;

  silk:ExternalFunction ## silk:AttachedProcedure[silk:returnValue => silk:Argument] ;

silk:Argument[silk:type {1:1} => silk:URI] ; // may be subsumed by named arguments
```

```
silk:BindingPattern[silk:in => silk:Argument,
                    silk:out => silk:Argument,
    silk:javaClass {0:1} => xsd:string] ; // other implementation members may be added
```

Issue 2.10: There may be a need for weak/root types used to handle underspecified objects and/or values. □

External procedures defined by SILK are discussed in section 2.18.

2.10 The HiLog Layer

HiLog [CKW93] extends the first-order syntax with higher-order features. In particular, it allows variables to range over function symbols, predicate symbols, and even atomic formulas. These features are useful for supporting reification and in reasoning about rather than strictly using formulas. HiLog further supports parameterized predicates, which are useful for generic definitions (illustrated below).

- ***HiLog term*** (H-term): A HiLog term is either a first-order term or an expression of the following form: $\mathtt{t(t_1,\ldots,t_n)}$, where `t`, $\mathtt{t_1}$, ..., $\mathtt{t_n}$ are HiLog terms.

This definition may seem quite similar to the definition of complex first-order terms, but, in fact, it defines a vastly larger set of expressions. In first-order terms, `t` must be a constant, while in HiLog it can be any HiLog term. In particular, it can be a variable or even another first-order term. For instance, the following are legal HiLog terms:

- *Regular first-order terms*: `c, f(a,?X), ?X`
- *Variables over function symbols*: `?X(a,?Y), ?X(a,?Y(?X))`
- *Parameterized function symbols*:
  `f(?X,a)(b,?X(c)), ?Z(?X,a)(b,?X(?Y)(d)), ?Z(f)(g,a)(p,?X)`

Such terms can be useful in knowledge representation and reflexive reasoning.

- ***HiLog atomic formula***: Any HiLog term is also a HiLog atomic formula.

Thus, expressions like `?X(a,?Y(?X))` are atomic formulas and thus can have truth values (when the variables are instantiated or quantified). Moreover, `?X` is also an atomic formula. Consequently, atomic formulas may be bound to variables and variables may be evaluated as formulas. For instance, the following HiLog query

```
?- q(?X) and ?X ;
p(a) ;
q(p(a)) ;
```

succeeds with the above database such that `?X` equals `p(a)`. Another interesting example of a HiLog rule is

```
call(?X) :- ?X ;
```

This can be viewed as a logical definition of the meta-predicate `call/1` in Prolog. Such a definition does not make sense in first-order logic (and is, in fact, illegal), but it is legal in HiLog and provides the expected semantics for `call/1`.

We will now illustrate one use of the *parameterized* predicates of the form `p(...)(...)`. The example shows a pair of rules that defines the transitive closure of a binary predicate. Depending on the actual predicate passed in as a parameter, we can get different transitive closures.

```
closure(?P)(?X,?Y) :- ?P(?X,?Y) ;
closure(?P)(?X,?Y) :- ?P(?X,?Z) and closure(?P)(?Z,?Y) ;
```

For instance, for the `parent` predicate, `closure(parent)` is defined by the above rules to be the ancestor relation. As another example, for an `edge` relation representing edges in a graph, `closure(edge)` represents the transitive closure of the graph.

2.11 The Equality Layer

This layer introduces the full ***equality predicate***, `:=:`. The equality predicate obeys the usual congruence axioms for equality. In particular, it is transitive, symmetric, reflexive, and logical entailment is invariant with respect to the substitution of equals by equals. For instance, if we are told that `bob:=:father(tom)` (`bob` is the same individual as the one denoted by the term `father(tom)`) then if `p(bob)` is known to be true then we should be able to derive `p(father(tom))`. If we are also told that `bob:=:uncle(mary)` is true then we can derive `father(tom):=:uncle(mary)`.

Equality in a Semantic Web language is important to be able to state that two different identifiers represent the same resource. For that reason, equality is part of OWL [DS04]. Although equality drastically increases the computational complexity, some forms of equality, such as ground equality, can be handled efficiently in a rule-based language.

The equality predicate `:=:` is different from the unification operator `=` in several respects. First, for variable free terms, $term_1$ `=` $term_2$ if and only if the two terms are identical. In contrast, two distinct terms can be equal with respect to `:=:`. Since `:=:` is reflexive, it follows that the relation that is used as an interpretation of `:=:` always contains the interpretation of `=`. Second, the unification operator `=` cannot appear in a rule head, while the equality predicate `:=:` can. When `:=:` occurs in the rule head (or as a fact), it is an assertion (conditioned on the truth value of the rule body) that two terms are equal. For instance, given the above definitions,

```
    p(1,2) ;
    p(2,3) ;
    f(a,?X):=:g(?Y,b) :- p(?X,?Y) ;
```

entails the following equalities between distinct terms: `f(a,1):=:g(2,b)` and `f(a,2):=:g(3,b)`.

Semantics: Informally, when $term_1$`:=:`$term_2$ occurs in the body of a rule and $term_1$, $term_2$ have variables, this predicate is interpreted as a test that variables can be consistently replaced with ground terms so that $term_1$ and $term_2$ will become equal with respect to `:=:` (note: equal, not identical!). For instance, in the query

```
  q(1) ;
  q(2) ;
  q(3) ;
  ?- f(a,?X):=:g(?Y,b) and q(?Y) ;
```

one answer substitution is `?X/1,?Y/2` and the other is `?X/2,?Y/3`. The formal definition of equality follows the standard outline of [Llo87] and will be given in a separate document. Section 2.17 provides an overview of the semantics. □

Editor's Note 2.9: This doen't reflect the current approach to derived and derived default equality in the LPDA working draft (the separate document?). Michael Kifer should update the semantics here and in 2.17. □

2.12 The Frames Layer

The Frames layer introduces object-oriented syntax modeled after F-logic [KLW95] and its subsequent enhancements [YK03b, YK03a]. The main syntactic additions of this layer include

- Frame syntax. Frames are called *molecules* here (following the F-logic terminology).
- Path expressions.
- Notation for class membership and subclasses.
- Notation for type specification, which is given by *signature molecules*.

The object-oriented extensions introduced by the Frames layer are orthogonal to the other layers described so far and can be combined with them within the SILK language.

As in most object-oriented languages, the three main concepts in the Frames layer of SILK are *objects*, *classes*, and *methods*. (We are borrowing from the object-oriented terminology here rather than AI terminology, so we refer to *methods* rather than *slots*.) Any class is also an object, and the same expression can denote an object or a class represented by this object in different contexts.

A ***method*** is a function that takes arguments and executes in the context of a particular object. When invoked, a method returns a result and can possibly alter the state of the knowledge base. A method that does not take arguments and does not change the knowledge base is called an ***attribute***. An object is represented by its *object Id*, the values of its attributes, and by the definitions of its methods. Method and attribute names are represented as objects, so one can reason about them in the same language.

An ***object Id*** is syntactically represented by a ground term. Terms that have variables are viewed as templates for collections of object Ids — one Id per ground instantiation of all the variables in the term. By *term* we mean any expression that can bind a variable. What constitutes a legal term depends on the layer. In the basic case, by term we mean just a first-order term. If the Frames layer is combined with HiLog, then terms are meant to be HiLog terms. Later, when we introduce reification, *reification terms* will also be considered.

2.12.0.1 ***Molecules.*** Molecules play the role of atomic formulas. We first describe atomic molecules and then introduce complex molecules. Although both atomic and complex molecules play the role of atomic formulas, complex molecules are *not* indivisible. This is why they are called molecules and not atoms. Molecules come in several different forms:

- **Value molecule**. If `t`, `m`, `v` are terms then `t[m -> v]` is a value molecule.

  Here `t` denotes an object, `m` denotes a ***method invocation*** in the scope of the object `t`, and `v` denotes a value that belongs to a set returned by this invocation. We call `m` "a method invocation" because if `m = s(`t_1`,...,`t_n`)`, i.e., has arguments, then `t[s(`t_1`,...,`t_n`) -> v]` is interpreted as an invocation of method `s` on arguments t_1`,...,`t_n in the context of the object `t`, which returns a set of values that contains `v`.

  The syntax `t[m -> {`v_1`,...,`v_k`}]` is also supported; it means that if `m` is invoked in the context of the object `t` then it returns a set that contains v_1`,...,`v_k. Thus, semantically, such a term is equivalent to a conjunction of `t[m -> `v_1`], ..., t[m -> `v_k`]`, so the expression `t[m -> {`v_1`,...,`v_k`]` is just a syntactic sugar.

- **Boolean valued molecule**. These molecules have the form `t[m]` where `t` and `m` are terms.

  Boolean molecules are useful to specify things like `mary[female]`. The same could be alternatively written as `mary[female -> true]`, but this is less natural.

  Issue 2.11: Are true and false special symbols? □

  Issue 2.12: Should we also be able to say `bill[not female]` as a shorthand for `bill[female -> false]`? □

- **Class membership molecule**: If `t` and `s` are terms then `t#s` is a membership molecule.

  If `t` and `s` are variable free, then such a molecule states that the object `t` is a member of class `s`. If these terms contain variables, then such a molecule can be viewed as many class membership statements, one per ground instantiation of the variables.

- **Subclass molecule**: If `t` and `s` are terms then `t##s` is a subclass molecule.

  If `t` and `s` are variable free, then such a molecule states that the object `t` is a subclass of `s`. As in the case of class membership molecules, subclass molecules that have variables can be viewed as statements about many subclass relationships.

- **Signature molecule**: If `t`, `m`, `v` are terms then `t[m => v]` is a signature molecule.

  If `t`, `m`, and `v` are variable-free terms then the informal meaning of the above signature molecule is that `t` represents a class, which has a method invocation `m` which returns a set of objects of type `v` (i.e., each object in the set belongs to class `v`). If these terms are non-ground then the signature represents a collection of statements — one statement per ground instantiation of the terms.

  When `m` itself has arguments, for instance `m = s(`t_1`,...,`t_n`)`, then the arguments are interpreted as types. Thus, `t[s(`t_1`,...,`t_n`) => v]` states that when the `n`-ary method `s` is invoked on an object of class `t` with arguments that belong to classes t_1, ..., t_n, the method returns a set of objects of class `v`.

- **Boolean signature molecules**: A Boolean signature molecule has the form `t[m=>]`. Its purpose is to provide type information for Boolean valued molecules. Namely, if `m=s(`t_1`,...,`t_n`)`, then when the method `s` is invoked on an object of class `t`, the method arguments must belong to classes t_1, ..., t_n.

- **Cardinality constraints**: Signature molecules can have associated cardinality constraints. Such molecules have the form

  t[s(t_1,...,t_n) {*min*:*max*} => v]

  where *min* and *max* are non-negative integers such that *min* $\leq$ *max*. *Max* can also be `*`, which means positive infinity.

  Such a signature states not only that the invocation of the method `s` with arguments of type t_1,...,t_n on an object of class `t` returns objects of class `v`, but also that the number of such objects in the result is no less than *min* and no more than *max*.

Semantics: The semantics of constraints in SILK is similar to constraints in databases and is *unlike* the cardinality restrictions in OWL

[DS04]. For instance, if a cardinality constraint says that an attribute should have at least two values and the rulebase derives only one then the constraint is *violated*. In contrast, OWL would *infer* that there is another, yet unknown, value. Likewise, if a cardinality constraint says that the number of elements is at most three while the rulebase derives four unequal elements then the constraint is, again, violated. This should be compared to the OWL semantics, which will infer that *some* pair of derived values in fact consists of equal elements. □

Editor's Note 2.10: The above semantics are largely duplicative with the more elaborated semantics given below. □

Signatures and type checking: Signatures are assertions about the expected types of the method arguments and method results. They typically do not have a direct effect on the inference (unless signatures appear in rule bodies). The signature information is optional.

Semantics: The semantics of signatures is defined as follows. First, the intended model of the knowledge base is computed (which in SILK is taken to be the well-founded model). Then, if typing needs to be checked, we must verify that this intended model is well-typed. A ***well-typed*** model is one where the value molecules conform to their signatures. For the precise definition of well-typed models see [KLW95]. (There can be several different notions of well-typed models. For instance, one for semi-structured data and another for completely structured data.) □

A type-checker can be written in SILK using just a few rules. Such a type checker is a query, which returns "No" if the model is well-typed and a counterexample otherwise. In particular, type-checking has the same complexity as querying. An example of such a type checker can be found in the FLORA-2 manual [YKWZ08].

Semantics: It is important to be aware of the fact that the semantics of the cardinality constraints in signature molecules is inspired by database theory and practice and it is *different* from the semantics of such constraints in OWL [DS04]. In SILK, cardinality constraints are restrictions on the intended models of the knowledge base, but they are not part of the axioms of the knowledge base. Therefore, the intended models of the knowledge base are determined without taking the cardinality constraints into account. Intended models that do not satisfy these restrictions are discarded. In contrast, in OWL cardinality constraints are represented as logical statements in the knowledge base and all models are computed by taking the constraints into account. Therefore, in OWL it is not possible to talk about knowledge base updates that violate constraints. For instance, the following signature `married[spouse {1:1} => married]` states that every married person has exactly one spouse. If `john:married` is true but there is no information about John's spouse then OWL will assume that `john` has some unknown spouse, while SILK will reject the knowledge base as inconsistent. If, instead, we know that `john[spouse -> mary]` and `john[spouse -> sally]` then OWL will conclude that `mary` and `sally` are the same object, while SILK will again rule the knowledge base to be inconsistent (because, in the absence of information to the contrary — for example, if no `:=:`-statements have been given — `mary` and `sally` will be deemed to be distinct objects). □

2.12.0.2 ***Inheritance in SILK.*** Inheritance is an optional feature, which is expressed by means of the syntactic features described below. In SILK, methods and attributes can be inheritable and non-inheritable. Non-inheritable methods/attributes correspond to class methods in Java, while inheritable methods and attributes correspond to instance methods.

The value- and signature-molecules considered so far involve *non-inheritable* attributes and methods. Inheritable methods are defined using the `*->` and `*=>` arrow types, i.e., `t[m *-> v]` and `t[m *=> v]`. For Boolean methods we use `t[*m]` and `t[m *=>]`.

Signatures obey the following laws of ***monotonic inheritance***:

- `t#s and s[m *=> v]` entail `t[m => v]`.
- `t##s and s[m *=> v]` entails `t[m *=> v]`.

These laws state that type declarations for inheritable methods are inherited to subclasses in an inheritable form, i.e., they can be further inherited. However, to the *members* of a class such declarations are inherited in a non-inheritable form. Thus, inheritance of signatures is propagated through subclasses, but stops once it hits class members.

Inheritance of value molecules is more involved. This type of inheritance is ***nonmonotonic*** and it can be overridden if the same method or attribute is defined for a more specific class. More precisely,

- `t#s and s[m *-> v]` entail `t[m -> v]` unless overridden or in conflict.
- `t##s and s[m *-> v]` entail `t[m *-> v]` unless overridden or in conflict.

Similarly to signatures, value molecules are inherited to subclasses in the inheritable form and to members of the classes in the non-inheritable form. However, the key difference is the phrase “unless overridden or in conflict.” Intuitively, this means that if, for example, there is a class `w` in-between `t` and `s` such that the inheritable method `m` is defined there then the inheritance from `s` is blocked and `m` should be inherited from `w` instead. Another situation when inheritance might be blocked arises due to multiple inheritance conflicts. For instance, if `t` is a subclass of both `s` and `u`, and if both `s` and `u` define the method `m`, then inheritance of `m` does not take place at all (either from `s` or from `u`; this policy can be modified by specifying appropriate rules, however). The precise model-theoretic semantics of inheritance with overriding is based on an extended form of the Well-Founded Semantics. Details can be found in [YK03a].

Note that signature inheritance is not subject to overriding, so *every* inheritable molecule is inherited to subclasses and class instances. If multiple molecules are inherited to a class member or a subclass, then all of them are considered to be true.

Inheritance of Boolean methods is similar to the inheritance of methods and attributes that return non-Boolean values. Namely,

- `t#s and s[m *=>]` entail `t[m=>]`.
- `t##s and s[m *=>]` entails `t[m *=>]`.
- `t#s and s[*m]` entails `t[m]` unless overridden.
- `t##s and s[*m]` entails `t[*m]` unless overridden.

2.12.0.3 ***Complex molecules.*** SILK molecules can be combined into complex molecules in two ways:

- By grouping.
- By nesting.

Grouping applies to molecules that describe the same object. For instance,

```
t[m1 -> v1] and t[m2 => v2] and t[m3 {6:9} => v3] and t[m4 -> v4]
```

is, by definition, equivalent to

```
t[m1 -> v1 and m2 => v2 and m3 {6:9} => v3 and m4 -> v4]
```

Molecules connected by the `or` connective can also be combined using the usual precedence rules:

```
t[m1 -> v1] and t[m2 => v2] or t[m3 {6:9} => v3] and t[m4 -> v4]
```

becomes

```
t[m1 -> v1 and m2 => v2 or m3 {6:9} => v3 and m4 -> v4]
```

The `and` connective inside a complex molecule can also be replaced with a comma, for brevity. For example,

```
t[m1 -> v1, m2 => v2]
```

Nesting applies to molecules in the following "chaining" situation, which is a common idiom in object-oriented databases:

```
t[m -> v] and v[q -> r]
```

is by definition equivalent to

```
t[m -> v[q -> r]]
```

Nesting can also be used to combine membership and subclass molecules with value and signature molecules in the following situations:

```
t#s and t[m -> v]
t##s and t[m -> v]
```

are equivalent to

```
 t[m -> v]#s
 t[m -> v]##s
```

respectively.

Molecules can also be ***nested inside predicates*** and terms with a semantics similar to nesting inside other molecules. For instance, `p[a ->c]` is considered to be equivalent to `p(a) and a[b ->c]`. Deep nesting and, in fact, nesting in any part of another molecule or predicate is also allowed. Thus, the formulas

```
p(f(q,a[b -> c]),foo)
a[b -> foo(e[f -> g])]
a[foo(b[c -> d]) -> e]
a[foo[b -> c] -> e]
a[b -> c](q,r)
```

are considered to be equivalent to

```
p(f(q,a),foo) and a[b -> c]
a[b -> foo(e)] and e[f -> g]
a[foo(b) -> e] and b[c -> d]
a[foo -> e] and foo[b -> c]
a[b -> c] and a(q,r)
```

respectively. Note that molecule nesting leads to a completely ***compositional syntax***, which in our case means that molecules are allowed in any place where terms are allowed. (Not all of these nestings might look particularly natural, e.g., `a[b ->c](q,r)` or `p(a[b ->c](?X))`, but there is no good reason to reject these nestings and thus complicate the syntax either.)

Issue 2.13: Nesting molecules inside predicates can sometimes be unintuitive. We might want to disallow this unless explicitly reified. □

Issue 2.14: Is precedence within nested molecules fully specified? Are there any problems with reification? □

2.12.0.4 ***Path expressions.*** Path expressions are useful shorthands that are widely used in object-oriented and Web languages. In a logic-based language, a path expression sometimes allows writing formulas more concisely by eliminating multiple nested molecules and explicit variables. SILK defines path expressions only as replacements for value molecules, since this is where this shorthand is most useful in practice.

A ***path expression*** has the form

```
t.t₁.t₂.  ...  .tₙ
```

or

```
t!t₁!t₂!  ...  !tₙ
```

The former corresponds to non-inheritable molecules and the latter to inheritable ones. In fact, "." and "!" can be mixed within the same path expression.

A path expression can occur anywhere where a term is allowed to occur. For instance, `a[b -> c.d]`, `a.b.c[e -> d]`, `p(a.b)`, and `X=a.b` are all legal formulas. The semantics of path expressions in the body of a rule and in its head are similar, but slightly different. This difference is explained next.

In the *body* of a rule, an occurrence of the first path expression above is treated as follows. The conjunction

```
t[t₁ -> ?Var₁] and ?Var₁[t₂ -> ?Var₂] and
...  and ?Varₙ₋₁[tₙ -> ?Varₙ]
```

is added to the body and the occurrence of the path expression is replaced with the variable `?Var`$_n$. In this conjunction, the variables `?Var`$_1$, ..., `?Var`$_n$ are *new* and are used to represent intermediate values. The second path expression is treated similarly, except that the conjunction

```
t[t₁ *-> ?Var₁] and ?Var₁[t₂ *-> ?Var₂] and
...  and ?Varₙ₋₁[tₙ *-> ?Varₙ]
```

is used. For instance, `mary.father.mother = sally` in a rule body is replaced with

```
mary[father -> ?F] and ?F[mother -> ?M] and ?M = sally
```

In the *head* of a rule, the semantics of path expressions is reduced to the case of a body occurrence as follows: If a path expression, ρ, occurs in the head of a rule, it is replaced with a new variable, `?V`, and the predicate `?V=`ρ is conjoined to the body of the rule. For instance,

```
p(a.b) :- body ;
```

is understood as

```
p(?V) :- body and ?V=a.b ;
```

Note that since molecules can appear wherever terms can, path expressions of the form `a.b[c -> d].e.f[g -> h].k` are permitted. They are conceptually similar to XPath [BBC+07] expressions with predicates that control the selection of intermediate nodes in XML documents. Formally, such a path expression will be replaced with the variable `?V` and will result in the addition of the following conjunction:

```
a[b -> ?X[ c-> d]] and ?X[ e-> ?Y]
        and ?Y[f->?Z[g - >h]] and ?Z[k -> ?V]
```

It is instructive to compare SILK path expressions with XPath. SILK path expressions were originally proposed for F-logic [KLW95] several years before XPath. The purpose was to extend the familiar notation in object-oriented programming languages and to adapt it to a logic-based language. It is easy to see that the "*" idiom of XPath can be captured with the use of a variable. For instance, `b/*/c` applied to object `e` is expressed as `e.b.?X.c`. The ".." idiom of XPath is also easy to express. For instance, `a/../b/c` applied to object `d` is expressed as `?_[?_ -> d.a].b.c`. On the other hand, there is no counterpart for the `//` idiom of XPath. The reason is that this idiom is not well-defined when there are cycles in the data (for instance, `a[b -> a]`). However, recursive descent into the object graph can be defined via recursive rules.

2.13 Reification

The ***reification layer*** allows SILK to treat certain kinds of formulas as terms and therefore to manipulate them, pass them as parameters, and perform various kinds of reasoning with them. In fact, the HiLog layer already allows certain formulas to be reified. Indeed, since any HiLog term is also a HiLog atomic formula, such atomic formulas are already reifiable. However, the reification layer goes several steps further by supporting reification of an arbitrary rule or formula that can occur in the rule head or rule body (provided that it does not contain explicit quantifiers — see below).

Formally, if F is a formula that has the syntactic form of a rule head, a rule body, or of a rule then F is also considered to be a term. This means that such a formula can be used wherever a term can occur.

Note that a reified formula represents an objectification of the corresponding formula. This is useful for specifying ontologies where objects represent theories that can be true in some worlds, but are not true in the present world (and thus those theories cannot be asserted in the present world). Examples include the effects of actions: effects of an action might be true in the world that will result after the execution of an action, but they are not necessarily true now.

In general, reification of formulas can lead to logical paradoxes [Per85]. The form of reification used in SILK does not cause paradoxes, but other unpleasantries can occur. For instance, the presence of a truth axiom (`true(?X) <==> ?X`) can render innocent looking rulebases inconsistent. However, as shown in [YK03b], the form of reification in SILK does not cause paradoxes as long as

- rule heads do not contain classical negation; and
- a rule head cannot be a variable, i.e., as long as the rules of the form `?X :- body` (which are legal in HiLog) are disallowed.

We therefore adopt the above restrictions for all layers of SILK.

Editor's Note 2.11: Need to rethink the above restrictions. In particular, `neg` in the heads should be allowed. □

Without special care, reification might introduce syntactic ambiguity, which arises due to the nesting conventions for molecules. For instance, consider the following molecule:

```
a[b -> t]
```

Suppose that `t` is a reification of another molecule, `c[d -> e]`. Since we have earlier said that any formula suitable to appear in the rule body can also be viewed as a term, we can expand the above formula into

```
a[b -> c[d -> e]]
```

But this is ambiguous, since earlier we defined the above as a commonly used object-oriented idiom, a syntactic sugar for

```
a[b -> c] and c[d -> e]
```

Similarly, if we want to write something like `t[b -> c]` where `t` is a reification of `f[g -> h]` then we cannot write `f[g -> h][b -> c]` because this nested molecule is a syntactic sugar for `f[g -> h] and f[b ->c]`. To resolve this ambiguity, we introduce the reification operator, `${...}`, whose only role is to tell the parser that a particular occurrence of a nested molecule is to be treated as a term that represents a reified formula rather than as syntactic sugar for the object-oriented idiom.

Note that the explicit reification operator is not required for HiLog predicates because there is no ambiguity. For instance, we do not need to write `${p(?X)}` below (although it *is* permitted and is considered the same as `p(?X)`):

```
a[b -> p(?X)]
```

This is because `a[b -> p(?X)]` does *not* mean `a[b -> p(?X)] and p(?X)`, since the sugar is used only for nested molecules. In contrast, explicit reification is needed below, if we want to reify `p(?X[foo -> bar])`:

```
a[b -> p(?X[foo -> bar])]
```

Otherwise `p(?X[foo -> bar])` would be treated as syntactic sugar for

```
a[b -> p(?X)] and ?X[foo -> bar]
```

Therefore, to reify `p(?X[foo -> bar])` in the above molecule one must write this instead:

```
a[b -> ${p(?X[foo -> bar])}]
```

Example 2.3: Reification in SILK is very powerful and yet it does not add to the complexity of the language. The following fragment of a knowledge base models an agent who believes in the modus ponens rule:

```
john[believes->${p(a)}] ;
john[believes->${p(?X)==>q(?X)}] ;
// modus ponens
john[believes->?A] :-
                john[believes->${?B==>?A}] and john[believes->?B] ;
```

Since the agent believes in `p(a)` and in the modus ponens rule, it can infer `q(a)`. Note that in the above we did not need explicit reification of `p(a)`, since no ambiguity can arise. However, we used the explicit reification anyway, for clarity. □

Syntactic rules. Currently SILK does not permit explicit quantifiers under the scope of the reification operator, because the semantics for reification given in [YK03b, KLP+04] does not cover this case. So not every formula can be reified. More specifically, the formulas that are allowed under the scope of the reification operator are:

- The formulas that are allowed in the rule head or quantifier-free formulas in the rule body.
- Quantifier-free rules.

The implication of these restrictions is that every term that represents a reification of a SILK formula has only free variables, which can be bound outside of the term. Each such term can therefore be viewed as a (possibly infinite) set of reifications of the ground instances of that formula.

Editor's Note 2.12: Remove the above restrictions and extend syntax/semantics. □

Issue 2.15: Can quantified formulas be reified? Some systems we want to interoperate with support this. □

Issue 2.16: Review how Common Logic handles reification. We want to work as gracefully as possible with higher-order features. □

2.14 Skolemization

It is often necessary to be able to specify existential information in the head of a rule or in a fact. Due to the limitations of the logic programming paradigm, which trades expressive power for execution efficiency, such information cannot be specified directly. However, existential variables in the rule heads can be *approximated* through the technique known as Skolemization [CL73]. The idea behind Skolemization is that in a formula of the form $\forall \mathtt{Y}_1 \ldots \mathtt{Y}_n \; \exists \mathtt{X} \; \ldots (\varphi)$ the existential variable `X` can be removed and replaced everywhere in φ with the function term $\mathtt{f(Y}_1\ldots\mathtt{Y}_n\mathtt{)}$, where `f` is a *new* function symbol that does not occur anywhere else in the specification.

Rationale: For any query, the original rulebase is unsatisfiable if and only if the transformed rulebase is unsatisfiable [CL73]. This implies that the query to the original rulebase can be answered if and only if it can be answered when posed against the Skolemized rulebase. However, from the point of view of logical entailment, the Skolemized rulebase is stronger than the original one, and this is why we say that Skolemization only *approximates* existential quantification, but is not equivalent to it. □

Skolemization is defined for formulas in *prenex normal form*, i.e., formulas where all the quantifiers are collected in a prefix to the formula and apply to the entire formula. A formula that is not in the prenex normal form can be converted to one in the prenex normal form by a series of equivalence transformations [CL73].

SILK supports Skolemization by providing special constants `_#` and `_#1`, `_#2`, `_#3`, and so on. As with other constants in SILK, these symbols can be used both in argument positions and in the position of a function. For instance, `_#(a,_#,_#2(c,_#2))` is a legal function term.

Each occurrence of the symbol `_#` denotes a new constant. Generation of such a constant is the responsibility of the SILK compiler. For instance, in `_#(a,_#,_#2(c,_#2))`, the two occurrences of `_#` denote two different constants that do not appear anywhere else. In the first case, the constant is in the position of a function symbol. The numbered Skolem constants, such as `_#2` in our example, also denote a new constant that does not occur anywhere else in the rulebase. However, the different occurrences of the same numbered symbol in *the same rule* denote the *same* new constant. Thus, in the above example the two occurrences of `_#2` denote the same new symbol. Here is a more complete example:

```
holds(a,_#1) and between(1,_#1,5) ;
between(minusInf,_#(?Y),?Y) :- timepoint(?Y) and ?Y != minusInf ;
```

In the first line, the two occurrences of `_#1` denote the same new Skolem constant, since they occur in the scope of the same rule. In the second line, the occurrence of `_#` denotes a new Skolem function symbol. Since we used `_#` here, this symbol is distinct from any other constant. Note, however, that even if we used `_#1` in the second rule, that symbol would have denoted a distinct new function symbol, since it occurs in a separate rule and there is no other occurrence of `_#1` in that rule.

Semantics: The Skolem constants in SILK are in some ways analogous to the blank nodes in RDF. However, they have semantics suitable for a rule-based language and it has been argued in [YK03b] that the Skolem semantics is superior to RDF, which relies on existential variables in the rule heads [Hay04]. □

Editor's Note 2.13: Discuss the relationship of skolemization to derived equality. □

2.15 Aggregation

SILK supports SQL-like aggregation over the results of a SILK query. The general syntax is

```
    ?Result = operator{?Var [GroupingVarList] | Query }
```

where *operator* can be `silk:max`, `silk:min`, `silk:avg`, `silk:count`, `silk:sum`, `silk:collectset`, or `silk:collectbag`.

Example 2.4:

```
// count employees
?- ?employeeCount = silk:count{?who | ?who # Employee} ;

// average salary of all employees
?- ?avgSalary = silk:avg{?salary | ?who # Employee[salary(?year)->?salary]} ;
```

□

The last two operators return lists of the instantiations of `?Var` that satisfy *Query* with (`silk:collectbag`) and without (`silk:collectset`) possible duplicates.

Example 2.5:

```
// years for which salary information is available
?- ?years = silk:collectbag{?year | ?who[salary(?year)->?salary]} ;

// unique years for which salary information is available
?- ?years = silk:collectset{?year | ?who[salary(?year)->?salary]} ;
```

□

The grouping variables provide functionality similar to `GROUP BY` in SQL. They have the effect that the aggregation produces one list of results per every instantiation of the variables in *GroupingVarList* for which *Query* has a solution. The variable `?Result` gets successively bound to each such list (one list at a time).

Example 2.6:

```
// each employee's average salary along with the value of the grouping variable ?who
?- ?avgSalary = silk:avg{?salary[?who] | ?who # Employee[salary(?year)->?salary]} ;

// total salary by year
?- ?yearlyPayroll = silk:sum{?salary[?year] | ?who # Employee[salary(?year)->?salary]} ;
```

□

2.16 SILK and XML Schema Data Types

SILK supports the primitive XML Schema data types [BM04]. The syntax is the same as in N3 and RIF. For instance, `"abc"^^xsd:string`, `"2005-07-18"^^xsd:date`, `"2008-01-03T15:55:40.34"^^xsd:dateTime`, `"123.56"^^xsd:decimal`, `"321"^^xsd:integer`, `"23e5"^^xsd:float`, and so on. Some frequently used data types have convenient abbreviations, e.g. `"abc"` for strings, `123.56` for decimals, `345` for integers, and `23e5` for floats.

2.17 Overview of the Semantics of SILK

A single point of reference for the model-theoretic semantics of SILK will be given in a separate document. Here we will only give an overview and point to the papers where the semantics of the different layers were defined separately.

First, we note that the semantics of the Lloyd-Topor layers — both monotonic and nonmonotonic — is transformational and was given in Sections 2.5 and 2.7. Similarly, the Courteous layer is defined transformationally and is described in [Gro99, WGK$^+$09].

The model theory of NAF is given by the well-founded semantics as described in [VRS91]. The model theory behind HiLog is described in [CKW93] and F-logic is described in [KLW95]. The semantics of inheritance that is used in SILK is defined in [YK03a]. The model theory of reification is given in [YK03b] and was further extended to reification of rules in [KLP$^+$04].

The semantics of the Equality layer is based on the standard semantics (for instance, [CL73]) but is modified by the ***unique name assumption***, which states that syntactically distinct terms are unequal. This modification is described in [KLW95], and we summarize it here. First, without equality, SILK makes the unique name assumption. With equality, the unique name assumption is modified to say that terms that cannot be proved equal with respect to `:=:` are assumed to be unequal. In other words, SILK makes a closed world assumption about explicit equality.

Other than that, the semantics of `:=:` is standard. The interpretation of this predicate is assumed to be an equivalence relation with ***congruence properties***. A layman's term for this is "substitution of equals by equals." This means that if, for example, `t:=:s` is derived for some terms `t` and `s` then, for any formula φ, it is true if and only if ψ is true, where ψ is obtained from φ by replacing some occurrences of `t` with `s`.

Overall, the semantics of SILK has a nonmonotonic flavor even without NAF and its extension layers. This is manifested by the use of the unique name assumption (modified appropriately in the presence of equality) and the treatment of constraints. To explain the semantics of constraints, we first need to explain the idea of ***canonical models***.

In classical logic, all models of a set of formulas are created equal and are given equal consideration. Nonmonotonic logics, on the other hand, carefully define a subset of models, which are declared to be ***canonical*** and logical entailment is considered only with respect to this subset of models. Normally, the canonical models are so-called ***minimal models***, but not all minimal models are canonical.

Any rule set that does not use the features of the NAF layer and its extensions is known to have a unique minimal model, which is also its canonical model. This is an extension of the well-known fact for Horn clauses in classical logic programming [Llo87]. With NAF, a rule set may have multiple incomparable minimal models, and it is well-known that not all of these models appropriately capture the intended meaning of the rules. However, it turns out that one such model can be distinguished, and it is called the ***well-founded model*** [VRS91]. A formula is considered to be true according to the SILK semantics if and only if it is true in that one single model, and the formula is false if and only if it is false in that model.

Now, in the presence of constraints, the semantics of SILK is defined as follows. Given a rulebase, first its canonical model is determined. In this process, all constraints are *ignored*. Next, the constraints are checked in the canonical model. If *all* of them are true, the rulebase is said to be consistent. If at least one constraint is false in the canonical model, the constraint is said to be violated and the rulebase is said to be inconsistent.

2.18 SILK Predicates

SILK defines a number of predicates, which are named using CURIEs in the `silk` namespace.

Rationale: In general, we prefer use of SILK-defined predicates to new language syntax. □

2.18.1 Querying External Data Sources

SILK will often use data from external data sources. A number of predicates are defined to access such data.

2.18.1.1 SPARQL Endpoints SILK supports both SELECT and CONSTRUCT queries on SPARQL endpoints.

`silk:sparqlQuery(?outputs, ?inputs, ?uri, ?template)` executes the SPARQL SELECT query `?template`, after substituting varibles in the list `?inputs`, using the endpoint `?uri`, binding the variables in the list `?outputs`.

Example 2.7: `silk:sparqlQuery([?river1,?river2],`
`[’’], "http://localhost:8080/sparql/parliament",`
`"SELECT DISTINCT ?river1 ?river2 WHERE { ?river1`
`<http://example.org/rivers#flowsInto> ?river2 }")` □

`silk:sparqlConstruct(?inputs, ?uri, ?template)` executes the SPARQL CONSTRUCT query `?template`, after substituting variables in the list `?inputs`, using the endpoint `?uri` and loads the resulting graph using the SILK RDF/OWL parser.

Example 2.8: `silk:sparqlConstruct([''], "http://localhost:8080/sparql/parliament", "CONSTRUCT { ?r rdf:type <http://example.org/rivers#River> } WHERE { ?r rdf:type <http://example.org/rivers#River> }")` □

2.18.1.2 ODBC Data Sources

SILK supports access to ODBC data sources including relational databases and spreadsheets.

`silk:odbcOpen(?dsn)` opens the ODBC data source with name `?dsn` to allow subsequent `silk:odbcQuery`s.

Example 2.9: `silk:odbcOpen(silk_emergency_data) ;` □

`silk:odbcQuery(?outputs, ?inputs, ?dsn, ?template)` executes the SQL query `?template`, after substituting variables in the list `?inputs`, using the ODBC data source with name `?dsn`, binding the variables in the list `?outputs`.

Example 2.10: `silk:odbcQuery([?ContactEmail, ?Message], [?riverSymbol], silk_emergency_data, "SELECT emergency_contact, message from FirstResponders where river = ?")` □

2.18.1.3 Web Services

SILK currently provides access to RESTful web services. SOAP web services are expected to be supported in the future.

`silk:restWebService(?outputs, ?inputs, ?uri, ?xpath)` invokes the RESTful web service `?uri`, after substituting variables in the list `?inputs`, selects values using the XPath expression `?xpath`, and binds them to variables in the list `?outputs`.

Example 2.11: `silk:restWebService([?Number], [?Person], "http://localhost:8090/people/?name=$1$", "//Person/phoneNumber/text()")` □

2.18.1.4 Other SILK Engines

SILK can query another running SILK Engine.

`silkb:querySilkInstance(?outputs, ?bindingOrder, ?inputs, ?url, ?template)` executes the SILK query `?template`, after substituting variables in the list `?inputs`, in the SILK Engine identified by `?url`, then binds the variables in the list `?outputs` in the order specified by `?bindingOrder`.

Example 2.12: `silkb:querySilkInstance([?z], ["?z"], [?x], "http://example.org/silk#engine1", "?- x(%1%, ?z);");` □

2.18.2 Built-ins

SILK provides built-in functions for datatype conversions, string operations, etc.

2.18.2.1 SWRL Built-ins SILK currently provides signatures and implementations for the built-in procedures supported by SWRL [HPSB+04].

2.18.2.2 RIF Built-Ins Future SILK implementations will provide signatures and implementations for the RIF Built-in Predicates and Functions [PBK08].

2.18.2.3 XPath and XQuery Functions and Operators XQuery and XPath Functions and Operators [MMW07] were the basis for most of the SWRL and RIF built-ins. Signatures and implementations will be provided using their original URIs. Operators may be used directly by the SILK infix operators.

2.18.3 External Actions

SILK can invoke procedures that affect the outside world. The following predicates are currently defined.

`silkb:writeLn(?input)` writes the string `?input` to stdout, followed by a newline.

Example 2.13: `silkb:writeLn("hello world!")` □

`silkb:sendEmail(?smtpServer, ?from, ?to, ?subject, ?text)` sends email from address `?from` to address `?to` with subject `?subject` and body `?text` via host `?smtpServer`.

Example 2.14: `silkb:sendEmail("smtp.bbn.com", "dkolas@bbn.com", "mdean@bbn.com", "greetings", "Hello Mike!")` □

2.18.4 Logging

The following external predicates provide an interface to the Java logging service Log4J, which reports messages for a specified logger and hierarchical level in accordance with a dynamic external configuration. These can be used to report contraint violations and other conditions.

`silkb:fatal(loggerName, message)` reports a fatal error.
`silkb:error(loggerName, message)` reports an error.
`silkb:warn(loggerName, message)` reports a warning.
`silkb:info(loggerName, message)` reports an information message.
`silkb:debug(loggerName, message)` reports a debugging message.
`silkb:trace(loggerNme, message)` reports a trace message.

Example 2.15: `silkb:error("silk", "an error") ;` □

2.18.5 List of All SILK Predicates

This section lists all the predicates in the `silk` and `silkb` namespaces, with descriptions and/or references to other sections.

`silk:arg`. See section 2.9.

`silk:arith(?var, ?value)` binds variable `?var` to `?value`.

`silk:avg`. See section 2.15.

`silk:binding`. See section 2.9.

`silk:cancel`. See section 2.8.5.

`silk:clause`. TBD.

`silk:collectbag`. See section 2.15.

`silk:collectset`. See section 2.15.

`silk:count`. See section 2.15.

`silkb:debug`. See section 2.18.4.

`silkb:error`. See section 2.18.4.

`silk:ExternalPredicate`. See section 2.9.

`silkb:fatal`. See section 2.18.4.

`silk:in`. See section 2.9.

`silkb:info`. See section 2.18.4.

`silk:javaClass`. See section 2.9.

`silk:max`. See section 2.15.

`silk:min`. See section 2.15.

`silk:odbcOpen`. See section 2.18.1.2.

`silk:odbcQuery`. See section 2.18.1.2.

`silk:opposes`. See section 2.8.4.

`silk:out`. See section 2.9.

`silk:overrides`. See section 2.8.3.

`silk:PersistentQuery`. See section 2.4.1.

`silkb:querySilkInstance`. See section 2.18.1.4.

`silk:restWebService`. See section 2.18.1.3.

`silk:sparqlConstruct`. See section 2.18.1.1.

`silk:sparqlQuery`. See section 2.18.1.1.

`silk:sum`. See section 2.15.

`silk:symbolToURI(?symbol, ?uri)` converts the symbol `?symbol` to the URI `?uri`.

`silkb:sendEmail`. See section 2.18.3.

`silkb:trace`. See section 2.18.4.

`silk:type`. See section 2.9.

`silkb:warn`. See section 2.18.4.

`silkb:writeLn`. See section 2.18.3.

2.19 SILK Grammars

A grammar provides a formal description of a language. The following subsections describe SILK in terms of a human-readable grammar for reference and an executable machine-readable grammar and API for implementations. The latter are needed because parser generation tools impose restrictions on grammars to facilitate efficient and unambiguous parsing.

Editor's Note 2.14: Some changes to these grammars can be expected as SILK continues to evolve. □

2.19.1 SILK Human-Readable Grammar

The following is an Extended BNF grammar for SILK intended for human consumption.

```
    Input  ::=  ( Statement ";" )* ¡EOF¿
Statement  ::=  Rule
             |  Query
             |  Mutex
             |  PrefixDeclaration
     Rule  ::=  ( Label )? Head ( ":-" Body )?
    Label  ::=  "{"Term "}"
     Head  ::=  Formula
     Body  ::=  Formula
  Formula  ::=  Term
             |  "(" Formula ")"
             |  Quantifier "(" Formula ")"
             |  "neg" Formula
             |  "naf" Formula
             |  Formula "and" Formula
             |  Formula "or" Formula
             |  Formula "==¿" Formula
             |  Formula "¡==" Formula
             |  Formula "¡==¿" Formula
     Term  ::=  ¡variable¿
             |  Constant
             |  "(" Term ")"
             |  Function
             |  Molecule
             |  PathExpression
             |  List
             |  ReifiedStatement
             |  Aggregate
             |  Term "=" Term
             |  Term "!=" Term
             |  Term ":=:" Term
             |  Term "+" Term
```

```
                                  |    Term "-" Term
                                  |    Term "*" Term
                                  |    Term "/" Term
                                  |    Term "¡" Term
                                  |    Term "¡=" Term
                                  |    Term "¿" Term
                                  |    Term "¿=" Term
                       TermList  ::=   Term ( "," Term )*
                       Constant  ::=   Number
                                  |    ¡symbol¿
                                  |    ¡string¿ ( "^^" ¡curie¿ )?
                                  |    URI
                         Number  ::=   ¡integer¿
                                  |    ¡decimal¿
                                  |    ¡float¿
                            URI  ::=   ¡fulluri¿
                                  |    ¡curie¿
                       Function  ::=   Term "(" TermList ")"
                       Molecule  ::=   ValueMolecule
                                  |    BooleanValuedMolecule
                                  |    ClassMembershipMolecule
                                  |    SubclassMolecule
                                  |    SignatureMolecule
                                  |    BooleanSignatureMolecule
                                  |    ComplexMolecule
                  ValueMolecule  ::=   Term "[" ValueMoleculeInternal "]"
          BooleanValuedMolecule  ::=   Term "[" BooleanValuedMoleculeInternal "]"
        ClassMembershipMolecule  ::=   Term "#" Term
               SubclassMolecule  ::=   Term "##" Term
              SignatureMolecule  ::=   Term "[" SignatureMoleculeInternal "]"
       BooleanSignatureMolecule  ::=   Term "[" BooleanSignatureMoleculeInternal "]"
                ComplexMolecule  ::=   Term "[" ComplexMoleculeInternal ( "," ComplexMoleculeInte
          ValueMoleculeInternal  ::=   Term ( "-¿" | "*-¿" ) ( Term | Set )
  BooleanValuedMoleculeInternal  ::=   Term
      SignatureMoleculeInternal  ::=   Term ( CardinalityConstraints )? ( "=¿" | "*=¿" ) Term
BooleanSignatureMoleculeInternal ::=   Term "=¿"
        ComplexMoleculeInternal  ::=   ValueMoleculeInternal
                                  |    BooleanValuedMoleculeInternal
                                  |    ClassMembershipMolecule
                                  |    SubclassMolecule
                                  |    SignatureMoleculeInternal
```

```
                              |     BooleanSignatureMoleculeInternal
                    Set     ::=     "{"TermList "}"
  CardinalityConstraints    ::=     "{"¡integer¿ ":" ( ¡integer¿ | "*" ) "}"
          PathExpression    ::=     Term ( ( "." | "!" ) Term )*
                    List    ::=     "[" TermList ( "|" Term )? "]"
              Quantifier    ::=     ( "forall" | "exist" ) VariableList
            VariableList    ::=     ¡variable¿ ( "," ¡variable¿ )*
       ReifiedStatement     ::=     "$""{"Statement "}"
               Aggregate    ::=     Constant "{"Variable ( "[" VariableList "]" )? "|" Formula "}
                   Query    ::=     "?-" Formula
                   Mutex    ::=     "!-" Term "and" Term ( "|" Formula )?
       PrefixDeclaration    ::=     ":-" "prefix" ¡symbol¿ "=" URI
```

2.19.2 Restrictions for SILK Layers

These grammars represent the composition of all the layers of SILK. They are necessarily somewhat over-permissive in that some syntactically valid rulesets may not be semantically valid.

The different layers of SILK impose restrictions on this grammar as follows:

Editor's Note 2.15: Summarize restrictions from previous sections in terms of the human-readable grammar. Consider use of a table for easy comparison. □

2.19.3 SILK LL(k) Grammar

The following isan executable JavaCC LL(k) grammar for SILK based on the human-readable grammar above. LL(k) grammars, used by JavaCC and ANTLR, preclude use of "left recursion" such as `Formula ::= Formula "or" Formula`. This is generally resolved by replacing references to the recursive non-terminal with a new non-terminal that include productions for the common prefix and possible continuations.

```
      Input   ::=   ( Statement ( ¡eos¿ | ( "." ¡EOF¿ ) ) )* ¡EOF¿
  Statement   ::=   Statement1
 Statement1   ::=   Rule
                |   Query
                |   Mutex
                |   PrefixDeclaration
  Aggregate   ::=   Constant "{"Variable ( "[" VariableList "]" )? "|" Body "}"
   Constant   ::=   NumericValue
                |   Uri
                |   ¡symbol¿
                |   ¡string¿ ( "^^" ¡curie¿ )?
```

```
     NumericValue  ::=  IntegerValue
                    |   ¡decimal¿
                    |   ¡float1¿
     IntegerValue  ::=  ¡integer¿
              Uri  ::=  ¡fulluri¿
                    |   ¡curie¿
PrefixDeclaration  ::=  ":-" "prefix" ¡symbol¿ "=" Uri
             Term  ::=  TermRelOp ( ( "=" | "!=" | ":=:" ) TermRelOp )*
        TermRelOp  ::=  TermAdd ( ( "¡" | "¡=" | "¿" | "¿=" ) TermAdd )*
          TermAdd  ::=  TermSubtract ( "+" TermSubtract )*
     TermSubtract  ::=  TermMultiply ( "-" TermMultiply )*
     TermMultiply  ::=  TermDivide ( "*" TermDivide )*
       TermDivide  ::=  Term1 ( "/" Term1 )*
                        Term0  (  "("  Function0  ")"  |  "["  Molecule0  "]"  |  "#"
            Term1  ::=  ClassMembershipMolecule  |  "##"  SubclassMolecule  |  "!"
                        PathExpression | "." PathExpression )*
            Term0  ::=  Aggregate
                    |   Constant
                    |   Variable
                    |   List
                    |   ReifiedStatement
                    |   "(" Formula ")"
         Variable  ::=  ¡variable¿
        Function0  ::=  TermList
         TermList  ::=  ( Term ( "," Term )* )?
             List  ::=  "[" TermList ( "|" Term )? "]"
             Rule  ::=  ( Label )? ( Rule1 )
            Label  ::=  "{"Term "}"
            Rule1  ::=  Body ( ":-" Body )?
     VariableList  ::=  ¡variable¿ ( "," ¡variable¿ )*
             Head  ::=  Formula
             Body  ::=  Formula
          Formula  ::=  FormulaOr
       Quantifier  ::=  "exist"
                    |   "forall"
       FormulaImp  ::=  Formula0 ( ( ¡rimp¿ | ¡limp¿ | ¡bidi¿ ) Formula )*
       FormulaAnd  ::=  FormulaImp ( ( "and" ) FormulaImp )*
        FormulaOr  ::=  FormulaAnd ( "or" FormulaAnd )*
            Query  ::=  "?-" Body
         Formula0  ::=  "neg" Formula0
                    |   "naf" Formula0
```

	|	Quantifier VariableList "(" Formula ")"
	|	Term
Mutex	::=	"!-" Term ("and") Term ("|" Body)?
Molecule0	::=	Molecule2 ("," Molecule2)*
ClassMembership-Molecule	::=	Term
SubclassMolecule	::=	Term
Molecule2	::=	("*")? Term (Molecule3)?
Molecule3	::=	"-¿" (Term | Set)
	|	"*-¿" (Term | Set)
	|	(CardinalityConstraints)? ("*")? "=¿" (Term)?
Set	::=	"{"TermList "}"
CardinalityConstraints	::=	"{"IntegerValue ":" (IntegerValue | "*") "}"
PathExpression	::=	Term
ReifiedStatement	::=	"$""{"Statement1 "}"

Editor's Note 2.16: Consider adding an alternative SILK LALR Grammar section 2.17.4. □

2.19.4 SILK Meta Model

The following is a meta model for SILK expressed in SILK:

```
<>[rdfs:comment->"SILK Meta Model represented in SILK",
   owl:versionInfo->"$Id: SILK.silk 1202 2009-12-21 07:18:28Z mdean $"] ;

// TODO:  should these be in the silk namespace, silkmm, or something else?

// TODO:  would prefer to use {1} for fixed cardinality

Model[statement {0:*} *=> Statement] ;

Statement[label {0:1} *=> Term,
          uniqueName {0:1} *=> Term,
          annotation {0:*} *=> Annotation ] ;

  // indentation follows class hierarchy

  Rule ## Statement[head {0:1} => Formula,
                    body {0:1} => Formula] ;

  Query ## Statement[query {1:1} => Formula] ;

  Mutex ## Statement /* [ ] */ ;
  PrefixDeclaration ## Statement['prefix' {0:1} => Symbol, // prefix is a reserved token
                                 uri {0:1} => FullURI] ;

Formula /* [ ] */ ;

  Term ## Formula /* [ ] */ ;  // a Term can be used anywhere a Formula can

    Constant ## Term /* [value {1:1} *=> top literal] */ ;

      Symbol ## Constant[value {1:1} => xsd:string] ;

      TypedConstant ## Constant[datatype {1:1} *=> xsd:anyURI] ;

        SilkString ## TypedConstant[value {1:1} => xsd:string] ;

        SilkNumber ## TypedConstant ;

          SilkInteger ## SilkNumber[value {1:1} => xsd:integer] ;
                                    // datatype -> xsd:integer

          SilkDecimal ## SilkNumber[value {1:1} => xsd:decimal] ;
                                    // datatype -> xsd:decimal

          SilkFloat ## SilkNumber[value {1:1} => xsd:float] ;
                                  // datatype -> xsd:float

        URI ## TypedConstant[value {1:1} *=> xsd:anyURI] ;
             // datatype *-> xsd:anyURI

          FullURI ## URI ;

          CURIE ## URI['prefix' {1:1} => Symbol, // prefix is a reserved token
                       localName {1:1} => xsd:string] ;

    Variable ## Term[name {1:1} => xsd:string] ;

    Function ## Term[predicate {1:1} => Term,
                     argument {0:*} => Term] ;

    List ## Term[value *=> Term] ;

    Molecule ## Term ;

      ValueMolecule ## Molecule[predicate {1:1} *=> Term,
                                value {1:1} *=> Term] ;
        BooleanValuedMolecule ## ValueMolecule[value {1:1} => xsd:boolean] ;

      SetValuedMolecule ## Molecule[predicate {1:1} *=> Term,
                   value {0:*} => Term] ;

      ClassMembershipMolecule ## Molecule[member {1:1} => Term,
                                    class {1:1} => Term] ;

      SubclassMolecule ## Molecule[subclass {1:1} => Term,
                             superclass {1:1} => Term] ;

      SignatureMolecule ## Molecule[method {1:1} *=> Term,
                   type {1:1} *=> Term,
                              minCardinality {0:1} *=> Term,  // Integer or Variable
                              maxCardinality {0:1} *=> Term] ; // Integer or Variable

        BooleanSignatureMolecule ## SignatureMolecule /* [type -> xsd:boolean,
                                                  maxcardinality -> 1] */ ;

      ComplexMolecule ## Molecule[subject {1:1} => Term,
                            molecule {0:*} => Molecule] ;

    PathExpression ## Term[molecule {1:*} => Molecule] ;

    ReifiedStatement ## Term[statement {1:1} => Statement] ;

    Aggregate ## Term[operator {1:1} => Constant,
                  aggregationVariable {1:1} => Variable,
                  groupingVariable {0:*} => Variable,
      query {1:1} => Query] ;

    CompoundTerm ## Term[connective {1:1} => Connective,
                     left {1:1} => Term,
                     right {1:1} => Term] ;

  CompoundFormula ## Formula[connective {1:1} => Connective,
             left {1:1} => Formula,
     right {1:1} => Formula] ;

  NegatedFormula ## Formula[type {1:1} => Negation,
              formula {1:1} => Formula] ;

  QuantifiedFormula ## Formula[quantifier {0:*} => Quantifier,
                formula {1:1} => Formula] ;

Quantifier[variable {1:*} => Variable] ;
```

```
  ForAll ## Quantifier ;

  Exist ## Quantifier ;

Negation ;

  Neg # Negation ;

  Naf # Negation ;

Annotation /* [ ] */ ;

Connective ;
```

2.19.5 SILK Java API

A Java Application Programming Interface (API) has been developed for SILK and is expected to be the primary interface used by most rule editing and execution tools.

The SILK API is composed of a set of component APIs:

- The Abstract Syntax API provides a set of interfaces based on the SILK grammar, Section 2.19.
- The Parser API supports a SILK parser and importing of various representations (e.g. RDF/XML).
- The Serializer API supports writing of SILK and export to other representations.
- The Engine API provides methods to control a SILK reasoning engine.
- The Query API provides a JDBC-like interface to query results.
- The Transform API provides a common interface to implementations of Lloyd-Topor and hypermonotonic transformations.
- The Checker API provides a standard interface for checking SILK statements for language (Section 2.19.2) and/or implementation restrictions.

The interface hierarchy in the Abstract Syntax API closely mirrors the SILK Meta Model. In general, a Model contains Statements including Rules which contain head and/or body Formulas which contain Terms.

Editor's Note 2.17: Consider adding a UML diagram showing both inheritance and containment. □

Javadoc for the SILK API is included in SILK software releases available at http://silk.semwebcentral.org.

2.20 SILK Conventions

The utility of a computer language is generally enhanced by conventions regarding its use.

Editor's Note 2.18: This is a placeholder for a section that will include naming conventions, suggested graphical representations (color and black-and-white), web-rendering guidelines, a "pronunciation guide" for reading SILK aloud, and (references to) a structured/controlled English vocabulary. □

Issue 2.17: These should consider W3C and other internationalization and accessibility guidelines. □

2.20.1 SILK Media Type

The nonstandard Internet media type (MIME type) `application/x-silk` may be used for content negotiation. Related media types are application/rdf+xml and application/rif+xml.

Issue 2.18: If a standard media type is desired, we could try to register `text/silk` or `application/silk`. □

2.21 Additional Features for SILK

The following features were not included in SWSL, but most were listed as possible extensions. These features will be added to SILK, in decreasing priority order.

1. **Predicates and functions with named arguments**. This extension allows terms and predicates of the form `p(foo -> 1, bar -> 2)`. The order of the arguments in such terms is immaterial. Variables are not allowed over the argument names; otherwise, unification has quadratic complexity.

   Issue 2.19: Note: independence of the order is very hard to implement unless we limit the arg names to constants (or, at least, variable-free terms). □

2. **Procedural attachments, state changes à la Transaction Logic, situated logic programs**. A ***procedural attachment*** is a predicate or a method that is implemented by an external procedure (e.g., in Java or Python). Such a procedure can have a side effect on the real world (e.g., sending an email message or activating a device) or it can receive information from the outside world. First formalizations of these ideas in the context of database and rule based languages appeared in [MW80, CGK89]. These ideas were more recently explored in [Gro04b] in the

context of e-commerce. Transaction Logic [BK98] provides a seamless integration of these concepts into the logic.

An attached procedure can be specified by a link statement, which associates a predicate or a method with an external program. The exact details of the syntax have not been finalized, but the following is a possibility:

```
attachment relation/Arity
           name-of-java-procedure(integer,string,...)
```

This syntax can be generalized to include object-oriented methods.

Editor's Note 2.19: Procedural attachments should include a standard set of builtin functions, including the XQuery and XPath Functions and Operators [MMW07] on XML Schema datatypes that are the basis for most builtins for SWRL, FLORA-2, and RIF. These could alternatively be made methods on datatype classes, as in FLORA-2. □

Another necessary extension involves *update primitives* - primitives for changing the underlying state of the knowledge. These primitives can add or delete facts, and even add or delete rules. A declarative account of such update operations in the context of a rule-based language is given by Transaction Logic [BK98]. This logic also can also be used to represent *Event-Condition-Action* rules [BKC94].

Editor's Note 2.20: See sections 2.9 and 2.18. □

3. **Database-style Constraints.** Constraints play a very important role in database and knowledge base applications. As a future extension, SILK will have database-style constraints. Database constraints are different in nature from restrictions used in Description Logic. Whereas restrictions in Description Logic are part of the same logical theory as the rest of the statements and are used to *derive* new statements, constraints in databases are not used to derive new information. Instead, they serve as tests of correctness for the canonical models of the knowledge base. In this framework, canonical models (e.g., the well-founded model [VRS91]) are first computed without taking constraints into account. These models are then checked against the constraints. The models that do not satisfy the constraints are discarded. In the case of the well-founded semantics, which always yields a single model, testing satisfaction of the constraints validates whether the knowledge base is in a consistent state.

4. **Event sublanguage**.

   Editor's Note 2.21: Expand based on slides 30-32 from B. Grosof ISWC 2006 Reaction RuleML talk. □

5. **If-then-else**. The `if` *`test`* `then` *`test1`* is sometimes more convenient and familiar than the `==>` operator. More important, however, is the fact that the more complete idiom, `if` *`test`* `then` *`test1`* `else` *`test2`*, is known to be very useful and common in rule-based languages. Although the `else`-part can be expressed with negation as failure, this is not natural and most well-developed languages support the *if-then-else* idiom directly. This idiom may be added to SILK later.

   Editor's Note 2.22: FLORA-2 includes if-then-else. □

6. **"rest"-variables**. The "rest" notation à la Common Logic [Com07] can be useful in metaprogramming. A rest-variable binds to a list of variables or terms and it always occurs as the last variable of a term. During unification with another term, such a variable binds to a list of arguments of that term beginning with the argument corresponding to the variable through the rest of the term (hence the name of such variables). For instance, in the following term, `?R` is a rest-variable:

   ```
   p(?X,?Y | ?R)
   ```

   If this term is unified with `p(?Z,f,?Z,q)`, then `?X` binds to `?Z`, `?Y` to `f`, and `?R` to the list `[?Z,q]`.

7. **Constraint solving.** Constraint solving (a.k.a. constraint logic programming) is an important knowledge representation paradigm to be included in SILK.

8. **Non-ground identity relation**, `==`. This predicate is true if the arguments are identical up to variable renaming. This predicate is not declarative but can be very useful, as demonstrated by its extensive use in logic programming.

The following features will be considered for inclusion in future versions of the SILK language and/or supporting rulesets. These have not yet been prioritized.

- **Modules**. Management and use of large knowledge bases will require additional naming and structuring mechanisms. These are expected to leverage work on FLORA-2 modules and CycL microtheories [Cyc02]. It should be possible to express overrides among such modules.
- **Process descriptions**. It should be possible to represent and reason over process descriptions, including defaults and exceptions. This is expected to leverage work from OWL-S [MBH$^+$04] and the Semantic Web Services Framework.
- **Time**. XML Schema provides basic `date`, `time`, and `dateTime` datatypes. OWL-Time adds intervals. Allen Relations are relevant for qualitative temporal reasoning.

- **Money**. This should include support for conversions and ISO 4217 Currency Codes.
- **Constraints beyond non-equality**. This is likely to incorporate work on Constraint Logic Programming.
- **Percents and ratios**. Look at how these are used in spreadsheets. Support for a rational datatype represented as a quotient of 2 integers, as in Common Lisp, may be helpful for preserving precision.
- **Spreadsheets**. SILK should be able to import data directly from spreadsheets. Consider also the use of spreadsheet-like metaphors for rule authoring.
- **SPARQL named graphs**. The ability to name collections of axioms and facts supports modularity and meta-programming.
- **RDF, RDBMS, XQuery, Web Services, and API interfaces**. SILK should be able to import facts directly from existing data sources.
- **Geospatial**. GeoRSS provides an RDF vocabulary for basic geometries based on the Open Geospatial Consortium's Geography Markup Language (GML). The Region Connection Calculus (RCC8) is relevant for qualitative spatial reasoning. It would be helpful to have more intuitive relation names than, e.g., NTPPi. Hybrid techniques will likely be necessary for efficient quantitative geospatial reasoning.
- **Named entities**. People, organizations, and locations are commonly referenced in rules. Definitions from the NIST Message Understanding Conference (MUC) and Automatic Content Extraction (ACE) evaluations are widely supported by information extraction products. Support should accommodate both human presentations and unique identifiers, as well as entity disambiguation (e.g. owl:sameAs and owl:differentFrom). FOAF is relevant for dealing with People. Support for various social networking algorithms and applications may also be desirable.
- **Meta-reasoning**. Some additional capabilities to reason over rules and modules will likely be required. This is still ill-defined.
- **Classical First Order Logic**. Classical FOL will likely be required for some aspects of process description and other applications. See Section 3. It should be possible to flag at an appropriate grain size (ruleset down to axiom) which language dialect is being used. Reasoning with FOL rules may be incomplete.
- **Structured comments and annotations**. It should be possible to associate additional information with rules and other entities. This should include workflow conventions such as TBD or @@. If possible, whitespace and other markup should be preserved. Compatibility with Wikis

and other collaborative authoring environments is desirable. Look at RIF annotations. CycL knowledge base documentation conventions [Cyc02] may be relevant here. Also consider augmenting textual comments with Javadoc-like conventions. Some annotations may be semantically significant. It should also be possible to associate comments with sets of rules.

- **Persistent internal identifiers**. It should be possible to associate persistent unique identifiers or labels with rules and other entities, to track naming and other changes over time. UUID URNs [LMS05] could be used for these identifiers.
- **Part/whole**. Simple part-whole relations in OWL Ontologies provides a basic OWL vocabulary.
- **Distinguished inequality symbols**. Possibly distinguish between user-asserted and derived inequalities. This could involve predicates and/or infix operators.
- **Equation solving and quantitative reasoning**. Consider integration of Wolfram|Alpha.
- **Rule macros**. Allow the definition and use of rule macros.
- **Implicit context**. Provide some mechanism for adding implicit additional terms, e.g. `on(mars)` or `at(time)`, to a specified set of rules, perhaps with some block scoping mechanism. This is somewhat analagous to the Pascal `with` statement.

Editor's Note 2.23: Expand links above into references and add additional references as these features are actually incorporated. □

3 SILK FOL

SILK FOL (First Order Logic) is a full first-order logic language that shares many of the elements of SILK. It exists primarily to facilitate reuse of ontologies from other knowledge representations based on FOL, such as Cyc and Common Logic. Large portions of SILK FOL can be simulated in SILK using the *hypermonotonic transform* as discussed in Section 3.4.

3.1 Overview of SILK FOL

SILK FOL is a *layered* language. Unlike OWL, the layers are not organized based on the expressive power and computational complexity. Instead, each layer includes a number of new concepts that enhance the *modeling power* of the language. This is done in order to make it easier to learn the language and to help understand the relationship between the different features. Furthermore,

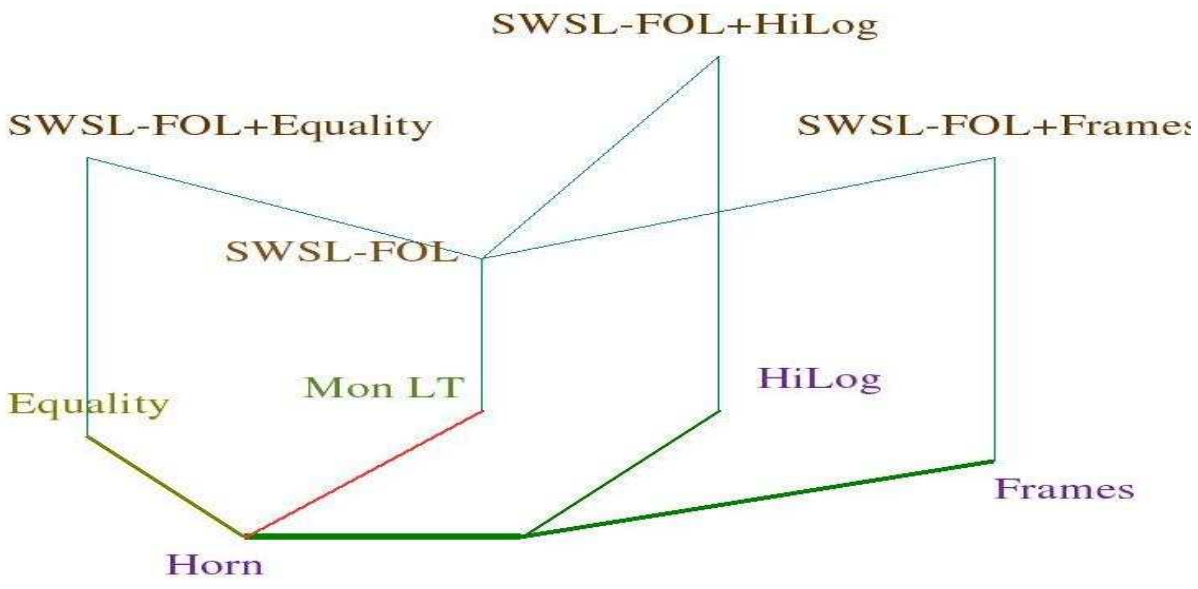

Figure 2: The Layers of SILK FOL

most layers that extend the core of the language are *independent* from each other — they can be implemented all at once or in any partial combination.

The layered structure of SILK FOL is depicted in Figure 2.

Editor's Note 3.1: Update Figure 2 to replace SWSL-FOL with SILK FOL. □

The bottom of Figure 2 shows those layers of SILK FOL that have monotonic semantics and therefore can be extended to full first-order logic. Above each layer of SILK Rules, the figure shows the corresponding SILK FOL extension. The most basic extension is *SILK FOL*. The other three layers, *SILK FOL+Equality*, *SILK FOL+HiLog*, and *SILK FOL+Frames* extend *SILK FOL* both syntactically and semantically. Some of these extensions can be further combined into more powerful FOL languages. We discuss these issues in Section 3.2.

3.2 SILK FOL Syntax

SILK FOL includes all the connectives used in first-order logic. First-order negation is denoted by `neg` (in contrast to the default negation of SILK, which is denoted by `naf`), and first-order implications are denoted by `<==` and `==>` (in contrast to `:-`, the nonmonotonic implication in SILK).

It follows from the above that SILK and SILK FOL share significant portions of their syntax. In particular, every connective used in SILK FOL can also be used in SILK. However, not every first-order formula in SILK FOL is a rule and the rules in SILK are not first-order formulas (because of "`:-`"). Therefore, neither SILK FOL is a subset of SILK nor the other way around. Furthermore, even though the classical connectives `neg` and `==>`/`<==` can occur in SILK, they are embedded into an overall nonmonotonic language and their semantics cannot be said to be exactly first-order.

Formally, ***SILK FOL*** consists of the following formulas:

- First-order atomic formulas
- If φ and ψ are SILK FOL formulas then so are φ `and` ψ, φ `or` ψ, `neg` φ, φ `==>` ψ, φ `<==` ψ, and φ `<==>` ψ.
- If φ is a SILK FOL formula and `X` is a variable, then the following are also SILK FOL formulas: `exist ?X` (φ) and `forall ?X` (φ). SILK FOL allows to combine quantifiers of the same sort, so `exist ?X,?Y` (φ) is the same as `exist ?X exist ?Y` (φ).

As in the case of SILK, we will use the semicolon (";") to designate the end of a SILK FOL formula.

SILK defines three extensions of SILK FOL. The first extension adds the equality operator, `:=:`, the second incorporates the object-oriented syntax from the Frames layer, and the third does the same for the HiLog layer.

Formally, ***SILK FOL+Equality*** has the same syntax as SILK FOL, but, in addition, the following atomic formulas are allowed:

- *term* `:=:` *term*

SILK FOL+Frames has the same syntax as SILK FOL except that, in addition, the following is allowed:

- SILK molecules, as defined in the Frames Layer of SILK, are valid SILK FOL formulas.
- The path expressions defined of the SILK Frames syntax are not used in SILK FOL. In SILK, path expressions are interpreted differently in the rule head and body. Since SILK FOL does not distinguish the head of a rule from its body, the path expression syntax is not well-defined in this context.

SILK FOL+HiLog extends SILK FOL by allowing HiLog terms and HiLog atomic formulas instead of first-order terms and first-order atomic formulas.

Each of these extensions is not only a syntactic extension of SILK FOL but also a semantic extension. This means that if φ and ψ are formulas in SILK FOL then $\varphi \models \psi$ in SILK FOL if and only if the same holds in SILK FOL+Equality, SILK FOL+Frames, and SILK FOL+HiLog. We will say that SILK FOL+Equality, SILK FOL+Frames, and SILK FOL+HiLog are ***conservative semantic extensions*** of SILK FOL.

SILK FOL+HiLog and SILK FOL+Frames can be combined both syntactically and semantically. The resulting language is a conservative semantic extension of both SILK FOL+HiLog and SILK FOL+Frames. Similarly, SILK FOL+Equality and SILK FOL+Frames can be combined and the resulting language is a conservative extension of both. Interestingly, combining SILK FOL+Equality with SILK FOL+HiLog leads to a conservative extension of SILK FOL+HiLog, but *not* of SILK FOL+Equality! More precisely, if φ and ψ are formulas in SILK FOL+Equality and $\varphi \models \psi$ then the same holds in SILK FOL+HiLog. However, there are formulas such that $\varphi \models \psi$ holds in SILK FOL+HiLog but not in SILK FOL+Equality [CKW93].

3.3 Overview of the Semantics of SILK FOL

The semantics of the first-order sublanguage of SILK FOL is based on the standard first-order model theory and is monotonic. The only new elements here are the higher-order extension that is based on HiLog [CKW93] and the frame-based extension based on F-logic [KLW95]. The respective references provide a complete model theory for these extensions, which extends the standard model theory for first-order logic.

3.4 Hypermonotonic Mapping: Combining SILK and SILK FOL

In this section, we discuss how to **combine** knowledge expressed in SILK with knowledge expressed in SILK FOL.

SILK is especially well suited for representing available knowledge and desired patterns of reasoning, including nonmonotonic reasoning. The rules paradigm has very efficient implementations, and there is vast experience in using rule systems.

SILK FOL is especially well suited for reasoning about disjunctive information, reasoning by cases, contrapositive reasoning, ets.

SILK and SILK FOL overlap largely in syntax, and SILK includes almost all of the connectives of SILK FOL. The deeper issue, however, is the semantic relationship between SILK and SILK FOL.

For several purposes it is desirable to *combine* knowledge expressed in the SILK form with knowledge expressed in the SILK FOL form. One important such purpose is:

- LP rules "on top of" FOL ontologies. "On top of" here means that some of the predicates mentioned in the set of rules are defined via ontological knowledge expressed in FOL. Such FOL ontologies can often be viewed as "background" knowledge.

For example, the predicates might be classes or properties defined via OWL DL axioms, i.e., expressed in the Description Logic fragment of FOL.

In terms of semantics, it is desirable to have reasoning in SILK *respect* as much as possible the information contained in such background FOL ontologies.

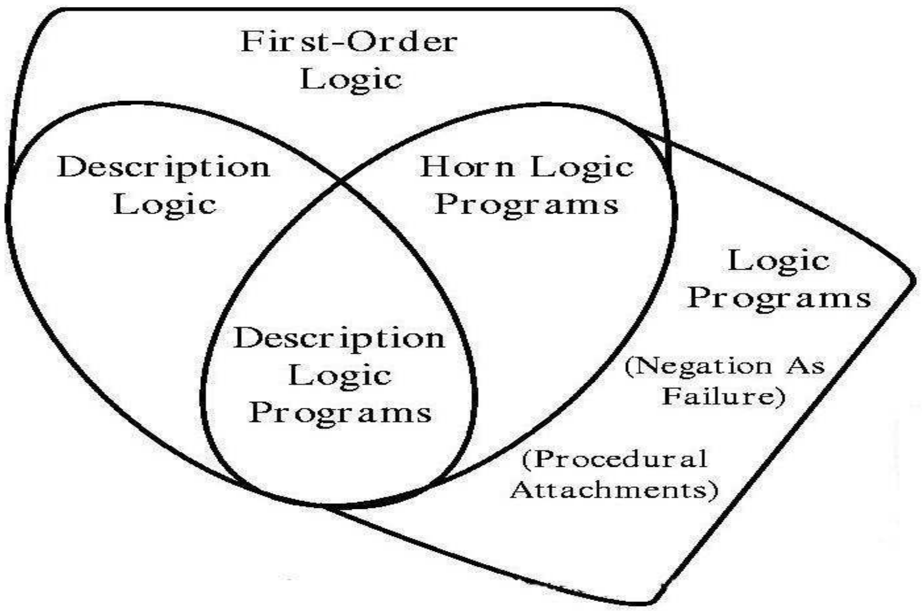

Figure 3: The relationships among different formalisms

In particular, it is desirable to enable sufficient completeness in the semantic combination to ensure that the conclusions drawn in SILK will be (classically) *not inconsistent* with the SILK FOL ontologies.

Ideally, there would be one well-understood overall knowledge representation formalism that subsumes both SILK and SILK FOL. This would provide the general theoretical basis for combining arbitrary SILK knowledge with arbitrary SILK FOL knowledge. Unfortunately, finding such an umbrella formalism is still an open issue for basic research. Instead, the current scientific understanding provides only a limited theoretical basis for combining SILK knowledge with SILK FOL knowledge. On the bright side, there are limited expressive cases for which it is well-understood theoretically how to do such combination.

The Venn diagram of relationships between the different formalisms, given in Figure 3 illustrates the most salient aspects of the current scientific understanding.

The shield shape represents first-order logic-based formalisms. The (diagonally-rotated) bread-slice shape shows the expressivity of the logic programming based paradigms. These overlap partially — in the Horn rules subset. FOL includes expressiveness beyond the overlap, notably: positive disjunctions; existentials; and entailment of non-ground and non-atomic conclusions. Likewise, LP includes expressiveness beyond the overlap, such as negation-as-failure, which is logically nonmonotonic. Description Logic (cf. OWL DL), depicted as an oval shape, is a fragment of FOL.

Horn FOL is another fragment of FOL. Horn LP is a slight weakening of

Horn FOL. “Weakening” here means that the conclusions from a given set of Horn premises that are entailed according to the Horn LP formalism are a subset of the conclusions entailed (from that same set of premises) according to the Horn FOL formalism. However, the set of ground atomic conclusions is the same in the Horn LP as in the Horn FOL. For most practical purposes (e.g., relational database query answering), Horn LP is thus essentially similar in its power to the Horn FOL.

Horn LP is a fragment of both FOL and nonmonotonic LP — i.e., of both SILK and SILK FOL. Horn LP is thus a limited “bridge” that provides a way to pass information — either premises, or ground-atomic conclusions — from FOL to LP, or vice versa. Knowledge from FOL that is in the Horn LP subset of expressiveness can be easily combined with general LP knowledge. Vice versa, knowledge from LP that is in the Horn LP subset of expressiveness can be easily combined with general FOL knowledge. Description Logic Programs (DLP) [GHVD03] represent a fragment of Horn LP. It likewise acts as a “bridge” between Description Logic (i.e., OWL DL) and LP.

Note that, technically, LP uses a different logical connective for implication (“`:-`” in SILK syntax) than FOL uses. When we speak of Horn LP as a fragment of FOL, we are viewing this LP implication connective as mapped into the FOL implication connective (also known as *material implication*).

3.4.0.1 ***Horn LP as “bridge”.*** To summarize, there is some initial good news about semantic combination:

- The Horn LP case is a “bridge” between SILK and SILK FOL.
- The DLP case is a “bridge” between SILK and OWL DL.

3.4.0.2 ***Builtin predicates.*** Another case of well behaved semantic combination is for ***builtin predicates*** that are purely informational, e.g., that represent arithmetic comparisons or operations such as less-than or multiplication. Technically, in LP these can be viewed as procedural attachments. But alternatively, they can be viewed as predicates that have fixed extensions. Their semantics in both FOL and LP can thus be viewed essentially as virtual knowledge base consisting of a set of ground facts. This thus falls into the Horn LP fragment.

3.4.0.3 ***Hypermonotonic reasoning as “bridge”.*** Recently, a new theoretical approach called ***hypermonotonic reasoning*** [Gro04a] has been developed to enable a case of “bridging” between (nonmon) LP and FOL that is considerably more expressive than Horn LP.

We will now describe in more detail some preliminary results about this hypermonotonic reasoning approach that bear upon the relationship of LP to FOL and thus upon how to combine LP knowledge with FOL knowledge.

Courteous LP (including its fragment: LP with negation-as-failure) can be viewed as a weakening of FOL, under a simple mapping of Courteous LP rules/conclusions into FOL. "Weakening" here means that for a given set of premises, the set of conclusions entailed in the Courteous LP formalism is in general a subset of the set of conclusions entailed by the FOL formalism. In other words:

- *(Courteous) LP is sound but incomplete relative to FOL.*

This fundamental relationship between the formalisms provides an augmentation to the theoretical basis for combining knowledge in LP (i.e., SILK) with knowledge in FOL.

Consider a set of rules S in LP and a set of formulas B in FOL. Let T be a translation mapping from the language of S to the language of B. S is said to be ***hypermonotonic*** with respect to B and T when S is sound but incomplete relative to B, under the mapping T. That is, when the conclusions entailed in S from a given set of premises P are in general always a subset of the conclusions entailed in B from the translated premises of S.

Define CLP2 to be the fragment of the Courteous LP formalism in which explicit negation-as-failure is omitted (i.e., prohibited). Each rule and mutex in CLP2 can be mapped quite straightforwardly and intuitively to a clause in FOL: simply replace the LP implication connective ("`:-`" in SILK syntax) by the FOL implication connective. Observe that this is the same mapping/translation that was considered in relating the Horn LP to FOL. Each ground-literal conclusion in CLP2 can also be mapped, in the same fashion, into a ground-literal in FOL.

The restriction on Courteous LP to avoid explicit negation-as-failure is not very onerous, essentially since the great majority of use cases in which explicit negation-as-failure is employed can be reformulated during manual authoring of rules so as to avoid it as a construct. More generally, the mapping can be extended, by complicating it a bit, to permit explicit negation-as-failure.

Going in the reverse direction, every clause in FOL can also be mapped into CLP2, in such a way that the resulting CLP information is a weakening of the FOL clause that nevertheless preserves much of the strength of the FOL clause. This reverse-translation mapping from FOL to CLP is complicated somewhat by the *directional* nature of the LP implication connective. "Directional" here means having a direction from body towards head. Each LP rule can be viewed as a directed clause. Consider a FOL clause C that consists of a disjunction of m literals:

- (universal closure of:) L1 or ... or Lm.

Here, each Li is an atom or a classically-negated atom. When mapping c to CLP2, there are m possible choices of one for each possible choice of which literal is to be made head of the LP rule. Each possible choice corresponds to a different rule — the LP rule in which literal Li is chosen as head has the form:

- `Li :- neg L1, ..., neg Li-1, neg Li+1, ..., neg Lm ;`

Altogether, the FOL clause C is mapped into a set of m LP rules:

- `L1 :- neg L2, neg L3, ..., neg Lm ;`
- `L2 :- neg L1, neg L3, ..., neg Lm ;`
- ...
- `Lm :- neg L1, neg L2, ..., neg Lm-1 ;`

where neg (neg A) is replaced equivalently by A. This set of rules is called the "omni-directional" set of rules for that clause — or, more briefly, the *"omni rules"* for that clause.

In general, FOL axioms need not be clausal since they may include existential quantifiers. However, often *skolemization* can be performed to represent such existentials in a manner that preserves soundness (as is usual for skolemization). A refinement of the reverse translation mapping above is to exploit such skolemization in order to relax the requirement of clausal form. We use such skolemization particularly for head existentials.

Issue 3.1: Discuss mapping a clause into omni vs. omni + exclusion vs. just exclusion. □

3.4.0.4 *Automatic weakened translation of FOL ontologies into SILK.* In the ontologies aspect of SILK, it is desirable to have a "bridging" technique to automatically translate FOL ontologies into SILK FOL in such a manner as to preserve soundness (from an FOL viewpoint) but to be nevertheless fairly strong (i.e., capture much of the strength/content of the original FOL axioms). The precise algorithm used to obtain the SILK translation for a given axiom in SILK FOL is as follows:

Input: a formula F in SILK FOL.
Output: a set of rules R, expressed in SILK.

1. Translate `F` into formula `F1` in Prenex Normal Form.
2. Skolemize `F1` to get `F2`, which is in Skolem Normal Form.
3. Write `F2` as a set S of clauses.
4. For each clause `C` in `S`, produce the omnidirectional set of rules for `C` (as defined above).

`R` then is the union of all the omnidirectional sets of rules produced at Step 4.

3.5 Parsing SILK FOL

The SILK grammar has been relaxed to incorporate SILK FOL. Specifically, a SILK FOL formula is represented as a body-less rule. Additional syntactic constraints can be checked to limit a rule set to only SILK or SILK FOL rules.

Rationale: There's a tremendous amount of overlap between the SILK and SILK FOL grammars and APIs. Maintaining 2 separate sets is impractical. □

3.6 Additional Features for SILK FOL

The following feature was not included in SWSL-FOL, but was listed as a possible extension; it will be included in SILK FOL.

- **Predicates and functions with named arguments**. This extension allows the terms and predicates of the form `p(foo -> 1, bar -> 2)`. The order of the arguments in such terms is immaterial. Variables are not allowed over the argument names; otherwise, unification has quadratic complexity.

References

[BBB+05] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. SWSL: Semantic Web Services Language. Technical report, W3C, April 2005. http://www.w3.org/Submission/SWSF-SWSL/.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. Technical report, W3C, January 2007. `http://www.w3.org/TR/xpath20/`.

[BK98] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[BK08] H. Boley and M. Kifer. RIF Framework for logic dialects. W3C Working Draft. `http://www.w3.org/TR/rif-fld/`, July 2008.

[BKC94] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri). Technical report, Internet Engineering Task Force, January 2005. `http://www.ietf.org/rfc/rfc3986.txt`.

[BM04] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. Recommendation 28 October 2004, W3C, 2004.

[BM08] M. Birbeck and S. McCarron. CURIE Syntax 1.0: A syntax for expressing compact URIs. Technical report, W3C, May 2008. `http://www.w3.org/TR/curie/`.

[CGK89] D. Chimeti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for ldl. In *Int'l Conference on Very Large Data Bases*, pages 195–203, 1989.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[CL73] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Com07] Common Logic Working Group. Common Logic (CL): A framework for a family of logic-based languages. Technical report, ISO, 2007. ISO 24707. `http://metadata-stds.org/24707/index.html`.

[Cyc02] Cycorp. Ontological engineering handbook. Manual, 2002. `http://www.cyc.com/doc/handbook/oe/oe-handbook-toc-opencyc.html`.

[DG05] M. Duerst and M. Guignard. Internationalized resource identifiers (iris). Technical report, Internet Engineering Task Force, January 2005. `http://www.ietf.org/rfc/rfc3987.txt`.

[DS04] M. Dean and G. Schreiber. Owl web ontology language reference. Technical report, W3C, February 2004. W3C Recommendation. `http://www.w3.org/TR/owl-ref/`.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Int'l Conference on Very Large Data Bases*, pages 273–284, Santiago, Chile, 1994. Morgan Kaufmann, San Francisco, CA.

[GDG+] B. Grosof, M. Dean, S. Ganjugunte, S. Tabet, , and C. Neogy. SweetRules: An open source platform for semantic web business rules. Web site. `http://sweetrules.projects.semwebcentral.org/`.

[GHVD03] B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *12th International Conference on the World Wide Web (WWW-2003)*, May 2003.

[Gro99] B.N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report RC 21472, IBM, July 1999.

[Gro04a] B. Grosof. Hypermonotonic reasoning: Unifying nonmonotonic logic programs with first order logic. In *Workshop on Principles and Practice of Semantic Web Reasoning (PPWSR04)*, September 2004. `http://www.mit.edu/~bgrosof/#HypermonFromPPSWR04InvitedTalk`.

[Gro04b] B. Grosof. Representing e-commerce rules via situated courteous logic programs in ruleml. *Electronic Commerce Research and Applications*, 3(1):2–20, 2004. `http://www.mit.edu/~bgrosof/#ecra-sclp-ruleml`.

[Hal] The Halo project. Web site. `http://www.projecthalo.com/`.

[Hay04] P. Hayes. Rdf model theory. Technical report, W3C, February 2004. W3C Recommendation. `http://www.w3.org/TR/rdf-mt/`.

[HPSB$^+$04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. Member Submission 21 May 2004, W3C, 2004. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`.

[KLP$^+$04] M. Kifer, R. Lara, A. Polleres, C. Zhao U. Keller, H. Lausen, and D. Fensel. A logical framework for web service discovery. In *ISWC 2004 Semantic Web Services Workshop*. CEUR Workshop Proceedings, November 2004.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, 1987.

[LMS05] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. Technical report, Internet Engineering Task Force, July 2005. `http://www.ietf.org/rfc/rfc4122.txt`.

[MBH$^+$04] David Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. Technical report, W3C, November 2004. W3C Member Submission. `http://www.w3.org/Submission/OWL-S/`.

[MMW07] A. Malhotra, J. Melon, and N. Walsh. Xquery 1.0 and xpath 2.0 functions and operators. Technical report, W3C, January 2007. `http://www.w3.org/TR/xpath-functions/`.

[MW80] D. Maier and D.S. Warren. Incorporation computed relations in relational databases. Technical Report 80/17, Department of Computer Science, SUNY at Stony Brook, December 1980.

[NS97] N.Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *Int'l Conference on Logic Programming*, 1997.

[Ont] Ontoprise, GmbH. Ontobroker. http://www.ontoprise.com/.

[PBK08] A. Polleres, H. Boley, and M. Kifer. RIF Datatypes and built-ins. W3C Working Draft. `http://www.w3.org/TR/rif-dtb/`, July 2008.

[Per85] D. Perlis. Languages with self-reference i: Foundations. *Artificial Intelligence*, 25:301–322, 1985.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[WGK+09] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *Int'l Conference on Logic Programming*, July 2009.

[YK03a] G. Yang and M. Kifer. Inheritance in rule-based frame systems: Semantics and inference. *Journal on Data Semantics*, 2800:69–97, 2003.

[YK03b] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the Semantic Web. *Journal on Data Semantics, LNCS 2800*, 1:69–98, September 2003.

[YKWZ08] G. Yang, M. Kifer, H. Wan, and C. Zhao. FLORA-2: User's manual. The FLORA-2 Web Site, 2008. `http://flora.sourceforge.net/docs/floraManual.pdf`.