SQUALL: a High-Level Language for Querying and Updating the Semantic Web

Sébastien Ferré

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1985 – Septembre 2011

SQUALL: a High-Level Language for Querying and Updating the Semantic Web

Sébastien Ferré*
ferre@irisa.fr

Abstract: Languages play a central role in the Semantic Web. An important aspect regarding their design is syntax as it plays a crucial role in the wide acceptance of the Semantic Web approach. Like for programming languages, an evolution can be observed from low-level to high-level designs. High-level languages not only allow more people to contribute by abstracting from the details, but also makes experienced people more productive, and makes the produced documents easier to share and maintain. We introduce SQUALL, a high-level language for querying and updating semantic data. It has a strong adequacy with RDF, an expressiveness very similar to SPARQL 1.1, and a controlled natural language syntax that completely abstracts from low-level notions such as bindings and relational algebra. We first give an informal presentation of SQUALL through examples, comparing it with SPARQL. We then formally define the syntax and semantics of SQUALL as a Montague grammar, and its translation to SPARQL.

Key-words: Semantic Web, controlled natural language, query language, update language, expressiveness, Montague grammar

---

SQUALL : un langage de haut niveau pour interroger et mettre à jour le Web sémantique

Résumé : *Les langages jouent un rôle central dans le Web sémantique. Un aspect important de leur conception est la syntaxe car elle joue un rôle crucial pour l'acceptation large du Web sémantique. Comme pour les langages de programmation, on peut observer une évolution des langages de bas-niveau vers des langages de haut-niveau. Les langages de haut-niveau ne permettent pas seulement à plus de personnes de contribuer en s'abstrayant de nombreux détails, mais rendent aussi les utilisateurs avancés plus productifs, et rendent les documents produits plus facile à partager et maintenir. Nous proposons SQUALL, un langage de haut-niveau pour interroger et mettre à jour des données sémantiques. Il a une forte adéquation avec RDF, une expressivité très similaire à SPARQL 1.1 et une syntaxe de type langage naturel contrôlé qui s'abstrait complètement des notions de bas-niveau tels que les substitutions ou l'algèbre relationnelle. Nous donnons tout d'abord une présentation informelle de SQUALL à travers des exemples et en comparant à SPARQL. Nous définissons ensuite formellement la syntaxe et la sémantique de SQUALL par une grammaire de Montague, ainsi que sa traduction en SPARQL.*

Mots clés : *Web sémantique, langage naturel contrôlé, langage de requêtes, langage de mises-à-jour, expressivité, grammaire de Montague*

---

* Équipe LIS, Université de Rennes 1

1 Introduction

Formal languages play a central role in many domains of computer science. This is obviously true in programming and algorithmics since the beginning, and other important examples are relational databases, and more recently the Semantic Web. Formal languages play the role of an interface between people and machines, and they can do so only because they precisely define the relation between syntax and semantics. This is a key distinction with natural languages.

The Semantic Web [AvH04, HKR09] is founded on a number of formal languages, used to represent: data (RDF), ontologies (RDFS, OWL), rules (SWRL), queries and updates (SPARQL 1.1 Update). In this paper, we focus on queries and updates, but our approach would naturally extend to ontologies and rules. Important aspects of query and update languages are: expressiveness (*what can be said*), and usability (*how it can be said* and *who can learn it*). SPARQL 1.1 Update (SPARQL for short) is very expressive [PAG06] as it includes relational algebra, and recently included aggregates, subqueries, negation, and property paths. However, its usability is limited because it exhibits low-level notions from relational algebra (e.g., bindings, join, union), and logic (e.g., variables, connectors, quantifiers). An ideal candidate for combining expressiveness and usability is natural language. However, full natural languages have a weak adequacy to Semantic Web formalisms [HBEV04], because they are too expressive and too ambiguous. While the NLP approach is required for extracting data from existing documents, it is not required for queries and updates, because they are produced interactively.

Our objective is to define a formal query and update language as a fragment of a natural language. Like Montague, we think that *"There is no important theoretical difference between natural languages and the artificial languages of logicians"* [Mon70]. This opinion is supported by the existence of Controlled Natural Languages (CNL) [Sma08, FKS06]. The main advantage of CNLs is to improve usability by reusing the cognitive capabilities of people, and therefore reducing their learning effort, while retaining the properties of formal languages. To the best of our knowledge, no existing CNL fulfills adequacy and interoperability with RDF(S) and SPARQL. ACE [FKS06] has its own underlying formalism (Discourse Representation Constructs), and SOS or Rabbit cover OWL ontologies and assume linguistic knowledge [SKC+08].

In this paper, we introduce SQUALL, a Semantic Query and Update High-Level Language. It qualifies as a CNL for querying and updating semantic data. Its contribution is not about expressiveness as it is roughly equivalent to SPARQL. The contribution of SQUALL is to provide a high-level and natural syntax that completely abstracts from low-level notions such as bindings or relational algebra, and still, that fulfills adequacy with Semantic Web formalisms. SQUALL shares the motivation for a nicer syntax with notations like Turtle and N3, but is rich enough to be a standalone language. For backward compatibility, SQUALL contains the Turtle notation as a subset. Unlike a number of CNLs, SQUALL does not require any domain-specific linguistic knowledge: e.g., knowing that "person" is a noun, whose plural is "people", and that "knows" is a transitive verb, whose passive is "known". This means that SQUALL is fully and directly interoperable with other notations for RDF(S) and SPARQL. This also means that SQUALL has to be learned, like other formal languages. However, we think that it is easier to learn, and that it makes it easier to formulate complex queries and updates: e.g., "for which researcher ?X, DBLP tell-s that every publication whose author is ?X and whose year $\geq$ 2000 has at least 2 author" is a valid query in SQUALL.

In Section 2, we first give preliminaries about the Semantic Web and Montague grammars. Section 3 is devoted to the description of the syntax and semantics of SQUALL, through a number of use cases for queries and updates, and through a comparison with the SPARQL syntax. In Section 4, we formally define the syntax and semantics of SQUALL, where semantics is given in terms of an intermediate logical language. Section 5 provides a translation from this intermediate language to SPARQL, therefore providing a concrete semantics as well as a possible implementation. Section 6 sketches the implementation along compiler techniques. Section 7 discusses the position of SQUALL w.r.t. existing CNLs for the Semantic Web, and the above language properties. Section 8 concludes this paper.

2 Preliminaries

We recall basic facts about the Semantic Web and Montague grammars. The Semantic Web provides, through RDF, the data model underlying both SPARQL and SQUALL. Montague grammars provide the theoretical framework in which we formally define the syntax and semantics of SQUALL.

2.1 Semantic Web

The Semantic Web (SW) is founded on several representation languages, such as RDF, RDFS, and OWL, which provide increasing inference capabilities [HKR09]. The two basic units of these languages are *resources* and *triples*. A resource can be either a URI (Uniform Resource Identifier), a literal (e.g., a string, a number, a date), or a *blank node*, i.e., an anonymous resource. A URI is the absolute name of a *resource*, i.e., an entity, and plays the same role as a URL w.r.t. web pages. Like URLs, a URI can be a long and cumbersome string (e.g., `http://www.w3.org/1999/02/22rdfsyntaxns#type`), so that it

is often denoted by a qualified name (e.g., `rdf:type`), where `rdf:` is the RDF namespace. In the notation N3, the default namespace `:` can be omitted for qualified names that do not collide with reserved keywords.

A triple (*s p o*) is made of 3 resources, and can be read as a simple sentence, where *s* is the subject, *p* is the verb (called the predicate), and *o* is the object. For instance, the triple (`Bob friend Alice`) says that "Bob has a friend Alice", where `Bob` and `Alice` are the bare qualified names of two individuals, and `friend` is the bare qualified name of a property, i.e., a binary relation. The triple (`Bob rdf:type man`) says that "Bob has type man", or simply "Bob is a man". Here, the resource `man` is used as a class, and `rdf:type` is a property from the RDF namespace. The triple (`man rdfs:subClassOf person`) says that "man is subsumed by person", or simply "every man is a person". The set of all triples of a knowledge base forms a RDF graph.

Query languages provide on semantic web knowledge bases the same service as SQL on relational databases. They generally assume that implicit triples have been inferred and added to the base. The most well-known query language, SPARQL, reuses the `SELECT FROM WHERE` shape of SQL queries, using graph patterns in the `WHERE` clause.

2.2 Montague Grammars

Montague grammars [DWP81] are an approach to natural language semantics that is based on formal logic and λ-calculus. It is named after the American logician Richard Montague, who pioneered this approach [Mon70]. A Montague grammar is a context-free generative grammar, where each rule is decorated by a λ-term that denotes the semantics of the syntactic construct defined by the rule. For example, the following rule gives the syntax and semantics of sentences like "there is a man":

$$S \rightarrow \textbf{there is } NP \quad \{\ np\ \lambda x.\textbf{true}\ \}$$

Here, **there** and **is** are keywords, or grammatical words, of the language; *S* (for sentence) and *NP* (for noun phrase) are syntagms. The semantics is given between curly brackets, and is defined in a fully compositional style, i.e., the semantics of a construct is always a composition of the semantics of sub-constructs. The semantics of a sub-construct is given by the lowercase name of the corresponding syntagm. Every λ-term is typed, and the type of the semantics of a given syntagm is always the same. By convention, we name λ-variables according to their type: x, y, z for RDF *resources* (a.k.a. Montague *entities*), p for RDF properties, t for reified statements (both are special kinds of resources), d for *descriptions*, and s for *sentences*. Sentences are the intention of truth values (a.k.a. Montague *propositions*), and descriptions are the intention of sets of resources (a.k.a. Montague *properties*), i.e., of functions from resources to sentences. In the above rule, x denotes a resource, **true** denotes a sentence, and *np* denotes the intention of a set of descriptions, i.e., a function from descriptions to sentences (a.k.a. Montague *property of properties*). Therefore, the whole λ-term denotes a sentence. The constants used in those λ-terms are the constructors of the intermediate language (e.g., **true**, **and**, **exists**), and RDF terms. The λ-terms obtained by composition can be simplified according to λ-calculus, through β-reduction (e.g., $\lambda x.t\ u =_\beta t[x \leftarrow u]$), and η-expansion (e.g., $t =_\eta \lambda x.(t\ x)$).

3 SQUALL by Examples

The syntax and semantics of SQUALL is informally given through examples, somewhat like when teaching a foreign language. The examples are taken from a list of use cases from a previous comparison of RDF query languages [HBEV04], which we extend to updates. Each use case tests for a language feature, and the resulting set of features closely matches the expressivity of SPARQL 1.1 Update, and is therefore useful to evaluate the expressiveness and compliance of SQUALL. For each example, the query/update is given in plain English, in SQUALL, and in SPARQL.

In the design of SQUALL, we have strived to fulfill a number of language properties, which we reuse from the comparison of RDF query languages [HBEV04] as well as from a report on CNLs for the Semantic Web [Sma08], and to which we add the property of abstraction:

Expressiveness. Expressiveness indicates how powerful queries and updates can be formulated in a given language.

Compliance. A language is compliant with the underlying knowledge formalism if it is *consistent* and *complete* with this formalism (e.g., RDFS, OWL, SWRL). *Consistency* means that everything that can be expressed in the language has a counterpart in the underlying formalism. *Completeness* means that the underlying formalism is fully covered by the language.

Safety. A language is considered safe, if every valid query returns a finite result, and every valid update has a finite effect.

Abstraction. A language is said abstract if it avoids to expose details or to make distinctions, that are only relevant to its implementation. Queries and updates of such a language should express *what* is to be searched or done, rather than *how*.

Flexibility. The flexibility of a language measures how constrained is the syntax, and whether a meaning can be expressed in several equivalent ways.

Usability. The usability of a language depends on how much training is required to learn it. The evaluation of usability should distinguish reading and writing.

3.1 Lexical Conventions

The lexical conventions for terms and variables are the same as in Turtle, N3 and SPARQL: `<http://www.aifb.unikarlsruhe.de/>` is a full URI, `rdf:type` is a qualified name, `:author` is a qualified name with default namespace, `_:12` is a blank node, `"Hello!"` is a plain literal, `"hello"@en` is a plain literal with a language tag, `"42"^^xsd:integer` is a typed literal, and `?X` is a variable. Like in N3, `42` is a valid representation of an integer literal, and `author` is a valid (bare) qualified name of the default namespace because it does not collide with a keyword of the language. In SQUALL examples, we assume the usual namespaces `rdf:`, `rdfs:`, `owl:`, and the domain vocabulary as the default namespace so that bare URIs can be used for most classes and properties. For reasons of space and legibility, we will use capital letters (like A, B) instead of full URIs to denote individuals (e.g., publications, persons). The keywords of SQUALL are grammatical words of English: e.g., **is**, **of**, **and**, **every**. They are displayed in bold font in examples to clearly distinguish them from bare qualified names.

SQUALL does not require any kind of domain-specific linguistic knowledge, and therefore makes no distinction between nouns and verbs at the lexical level (it does however at the syntactic level). A more essential distinction, which is readily available in a RDFS dataset, is between unary predicates (e.g., RDFS classes), and binary predicates (e.g., RDF properties).

3.2 Simple Sentences

A triple like A `worksFor` B forms the simplest sentence, where A is the subject, `worksFor` is the predicate, and B is the object. A simple sentence (S) is analyzed as a noun phrase (NP), the subject, followed by a verb phrase (VP). The VP is itself analyzed as a binary predicate (P2) followed by a NP, the object. From the definition of a triple, a NP can be a RDF term or a variable, and a P2 can be a property URI or a variable.

The Turtle notation extends simple sentences. A P2 can also be **a**, a shorthand for `rdf:type`; **has** `author`, a synonym for `author`; and **is** `author` **of**, the inverse of `author`. A NP or a P2 can also be the pair of square brackets `[ ]`, an implicit variable. A VP can be put between the square brackets to qualify the implicit variable. For example, the sentence A **is** `author` **of** `[ year 2010 ]` is equivalent to the two triples `?X author` A and `?X year 2010`. Finally, conjunctive coordination of sentences, VPs, or object NPs can be formed by using separators, respectively dots, semi-colons, or commas. A coordinated object NP distributes over its predicate: the verb phrase (`author` A`,` B) is equivalent to (`author` A`;` `author` B). A coordinated VP distributes over its uncoordinated subject NP: the sentence (`?X author` A`;` `author` B) is equivalent to (`?X author` A`.` `?X author` B). In SQUALL examples, the Turtle notation is avoided in favor of a more natural syntax, e.g., relatives instead of square brackets, and the coordinating word **and** instead of separators.

3.3 Queries

We present the syntax and semantics of SQUALL queries through the 14 use cases used for a comparison of the expressiveness of RDF query languages [HBEV04]. Each use case tests a different feature such as union, quantifiers, or reification. We compare SQUALL to the proposal for SPARQL 1.1[1] because SPARQL is now the reference query language in the Semantic Web, and because only its version 1.1 covers all use cases.

Path Expressions. This is the first and simplest use case. It involves the crossing of two properties, one from publications to their authors, and another from authors to their names.

English	*Return the names of the authors of publication A*
SQUALL	**what is the** `name` **of an** `author` **of** A
SPARQL	`SELECT ?n WHERE {` A `:author [ :name ?n ] }`

This example introduces new syntactic constructs. A noun phrase (NP) can be the question word **what**, which plays the role of `SELECT ?n`, but where the variable `?n` is left implicit. A verb phrase (VP) can be the copula **is** followed by a NP. The articles **a**/**an** and **the** are determiners (Det) that form NPs when followed by a nominal group (NG). In questions, there is no semantic difference between the definite and indefinite articles. Here, the two NGs are made of a binary predicate (P2),

[1] `http://www.w3.org/TR/sparql11-query/`

here `name` or `author`, and a NP, separated by the keyword **of**, and providing an inverse reading of the predicate. The NP (**an** `author`) is a shorthand for (**an** `author` **of** `[ ]`).

The keywords **is**, **a**/**an**, and **the** do not contribute to the semantics of the question, and only the English grammar makes them necessary. In Russian, for example, **is** is optional, and **a**/**an**/**the** do not even exist: the above question could then be shortened to: **what** `name` **of** `author` **of** *A*.

Union. This case corresponds to the set union of relational algebra.

English	*Return the labels of all topics and (union) the titles of all publications*
SQUALL	**what is the** `label` **of a** `topic` **or the** `title` **of a** `publication`
SPARQL	`SELECT ?x WHERE {{ [ a :topic ] :label ?x } UNION { [ a :publication ] :title ?x } }`

The low-level `UNION` is replaced by the coordinating word **or**. Like the coordinating word **and**, it can be applied between any two phrases of same kind, here two noun phrases (NP). Therefore, the above question can be seen as a factorization of: **what is the** `label` **of a** `topic` **or is the** `title` **of a** `publication`. The union of NPs has been distributed over the copula verb **is**. To avoid ambiguity, coordinations must be distributed in a determined order. First, a coordination of arguments (e.g., subject, object) must distribute before a coordination of predicates. Second, coordinations of arguments must distribute from left to right, i.e., the subject first, then the object. A nominal group (NG) can be an unary predicate (P1), here the class URI `publication`.

Difference. This case corresponds to the set difference of relational algebra.

English	*Return the labels of all topics that are not titles of publications*
SQUALL	**what is the** `label` **of a** `topic` **and not the** `title` **of a** `publication`
SPARQL	`SELECT ?x WHERE { [ a :topic] :label ?x. FILTER NOT EXISTS { [ a :publication ] :title ?x }}`

This use case is covered since SPARQL 1.1. The low-level `NOT EXISTS` is replaced by the negation **not**. It can apply to any phrase, here a noun phrase. A negated phrase produces no bindings, and therefore acts as a constraint. **not** follows the same distribution rules as **and** and **or**, and has higher priority than **and**, which has higher priority than **or**.

Optional. Optional patterns correspond to left joins in the relational algebra [PAG06].

English	*What are the names and, if known, the e-mail of the authors of all available publications ?*
SQUALL	**what is the** `name` **of an** `author` **and if defined, what is the** `email` **of this** `author`
SPARQL	`SELECT ?n ?e WHERE { ?p :author ?a. ?a :name ?n. OPTIONAL { ?a :email ?e } }`

We introduce the keyword **if defined,** that can be applied in front of any phrase. Here, it applies to a question, and **and if defined,** acts as an additional coordinating word between two questions. An optional necessarily succeeds, but may not produce any additional binding. **if defined,** follows the same distribution rules as **not**, **and**, **or**, and has lower priority than them.

The two occurences of **what** are necessary as they correspond to two different things, and hence to two different implicit variables. Each predicate at the head of a nominal group also introduces an implicit variable that can be refered to later by the demonstrative **this**. In case of ambiguity, the most recent occurence is chosen. If necessary, explicit variables can be used: the pair **an** `author` – **this** `author` can be replaced by the pair **an** `author ?X` – `?X`. Here, the variable `?X` is put in apposition to the predicate `author`. RDF terms can also be used in apposition to the head of a NG: e.g., **the** `professor` *A* **is an** `author` **of** *B*.

Quantification. Quantifiers are one of the least well-handled features of query languages. When they can be expressed, it is generally in a tedious way. For example, the universal quantifier "every" requires a combination of joins and differences in SPARQL 1.1.

English	*Return the persons who are authors of all publications*
SQUALL	**what is an** `author` **of every** `publication`
SPARQL	`SELECT ?a WHERE { ?a a :person. FILTER NOT EXISTS { ?p a :publication. FILTER NOT EXISTS { ?p :author ?a }}}`

every is a quantifier that can be used as a determiner, in place of an article. Other predefined quantifiers are the logical quantifiers **some**, **no**, **only**, and the cardinality quantifiers **at least** n, **at most** n, **exactly** n, where n is an integer. This list is reminiscent of the quantified restrictions in Description Logics (DL) [BCM+03]. More quantifiers could easily be defined, such as **at least** p**%**, where p is a percentage. To avoid ambiguities, the scope of a quantifier always includes the predicate of the sentence, even if it is at the left of the quantifier, and extends to the end of the non-coordinated sentence. Therefore, the sentence (**every** `professor` **is an** `author` **of a** `publication`) means that "for every professor X, there is a publication Y such that Y has author X".

Aggregation. Quantification is already a kind of aggregation, where the aggregated value is a truth value. SPARQL 1.1 introduces the classical aggregation operators `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`.

English	*Count the number of authors of a publication A*
SQUALL	**what is the** `count` **of the** `author` **of** *A*
SPARQL	`SELECT COUNT(?x) WHERE {` *A* `:author ?x }`

An aggregation forms a nominal group (NG) that uses the same syntax as the inverse reading of a binary predicate. The key difference, here, is that `count` is an aggregator instead of a property URI. It does not map from each author of *A* to a count, but from the set of all authors of *A* to a count.

The query can also be expressed as: **how many** `person` **is an** `author` **of** `X`. The keyword **how many** is a question determiner that corresponds to `SELECT COUNT(?x)` in SPARQL.

Grouping. Grouping comes as a refinement of aggregation, where the aggregator should be applied to each group instead of to the whole set.

English	*Return the total number of publications, when it is greater than 10, by author affiliation*
SQUALL	**what is the** `count` **of the** `publication` **per the** `affiliation` **of the** `author` **of this** `publication` **where this** `count` $>$ `10`
SPARQL	`SELECT COUNT(?p) WHERE { ?p :author [ :affiliation ?f ] } GROUP BY ?f HAVING (COUNT(?p) > 10)`

An aggregation can be followed by **per** and a list of noun phrases constituting dimensions against which the aggregated items are projected.

The coordinating word **where** is equivalent to **and** in queries, but it coordinates only sentences and it has a lower priority than other coordinations. Here, it introduces a constraint on each count. This constraint makes use of the built-in predicate $>$ ("is greater than"). Built-in predicates can be used syntactically exactly like RDF predicates (classes and properties), where the subject is passed as the first argument, and the object as the second argument (additional arguments are handled in the next use case about reification). This favors abstraction because it is a matter of modelling to choose between an intensionally-defined built-in predicate, and an extensionally-defined RDF predicate.

Reification. This use case is about the ability to refer to a statement. This is useful for constraining or questionning the circumstancial aspects of a statement (e.g., date, time, source). In SPARQL, this is possible only if the statement has been encapsulated into a named graph.

English	*Return the person who classified the publication A*
SQUALL	**what is the** `creator` **of that** *A* **has** `topic` **some thing**
SPARQL	`SELECT ?p WHERE { GRAPH ?g {` *A* `:topic ?t }. ?g :creator ?p }`

In SQUALL, every statement can be refered to, and a set of statements (a graph) can simply be formed by linking each of them to a same resource. The keyword **that** applies to a sentence (S), or a coordination of sentences, and forms a noun phrase (NP) that denotes the statement of those sentences.

The same query can be formulated as: **at** `creator` **what,** `X` **has** `topic` **some thing**, or more simply: **at which** `creator,` `X` **has some** `topic`. Here, `topic` is the predicate of the independent clause, instead of the predicate of the dependent clause. The keyword **at** followed by a property URI, here `creator`, forms a preposition. A preposition followed by a NP forms a prepositional phrase (PP). PPs can occur at any position in the sentence before, between, or after any of the subject, predicate or object. In fact, the subject NP can be seen as a shorthand for (**at** `rdf:subject NP`), and similarly for the predicate and the object NP. The prepositions, including `rdf:subject`, `rdf:predicate`, and `rdf:object`, describe the properties that link the statement to the predicate argument. The free ordering of PPs is useful to control the ordering and scope of quantifiers. Prepositional phrases allow for the natural representation of n-ary relations, while retaining compliance with binary relations of the underlying RDF data model. Each additional argument to the predicate is represented by a triple from the statement to the argument. For example, the SQUALL sentence (*A* `worksFor` *B* **at** `startDate 2001` **at** `status professor` means that "*A* works for *B* since 2001 as a professor". Additional arguments also allow for n-ary built-in predicates, like in the sentence **at which** `position,` **the** `title` **of** *A* `contains "semantic"`, where the built-in predicate `contains` has 3 arguments: the text (as subject), the pattern (as object), and the position.

The keyword **thing** serves as a void nominal group (NG), and denotes the class of all resources (like `rdfs:Resource`). The keyword **which** is a question determiner, and corresponds to a SELECT in SPARQL. **what** is a shorthand for **which thing**. To avoid the use of **thing** like in (**has** `topic` **some thing**) or (**at** `creator` **which thing**), the determiner can be moved in front of the property, which then plays the role of the head of the NG: (**has some** `topic`) or (**at which** `creator`). Assuming URIs for basic notions such as "person", "time", "place", and "reason", we can define the following syntactic sugar for grammatical words: **who** = **which** `person`; **when** = **at which** `time`; **where** = **at which** `place`; **why** = **at which** `reason`; **because of** = **at** `reason`; **because** = **at** `reason` **that**.

Namespaces. This use case is about the ability to access the URI string of a resource, and to perform string matching.

English	*Return all resources whose namespace starts with "http://www.aifb.unikarlsruhe.de/"*
SQUALL	**what is a** `rdfs:Resource` **whose** `uri matches "^http://www.aifb.unikarlsruhe.de/"`
SPARQL	`SELECT ?x WHERE { ?x a rdfs:Resource. FILTER regex(str(?x), "^http://www.aifb.unikarlsruhe.de/")}`

This query uses two built-in predicates: `uri` is a binary predicate from resources to their URI (when defined) as a string literal, and `matches` is a binary predicate from strings to the regular expressions that they match. The keyword **whose** introduces a relative clause (Rel), and is followed by a binary property (P2) and a verb phrase (VP). Relative clauses are part of nominal groups (NG), and come after the head and the optional apposition of the NG.

Given that everything in a RDF graph is a resource, the above question can also be expressed in SQUALL as: **whose** `uri matches "^http://www.aifb.unikarlsruhe.de/"`. Here, the keyword **whose** followed by a binary property (P2) forms a question NP, which is equivalent to: **the** `uri` **of what**.

Language. This use case is about the language tag of plain literals.

English	*Return the German label of the topic whose English label is "Database Management"*
SQUALL	**what has** `lang "de"` **and is the** `rdfs:label` **of the** `topic` **that has a** `rdfs:label` **whose** `str` **is** `"Database Management"` **and whose** `lang` **is** `"en"`
SPARQL	`SELECT ?gl WHERE { [ a :topic ] rdfs:label ?gl, ?el. FILTER ( lang(?gl) = "de" && lang(?el) = "en" && str(?el) = "Database Management" ) }`

This query uses two additional built-in predicates: `lang` is a binary predicate from plain literals to their language tags, and `str` is a binary predicate from literals to their string representation. The keyword **that** introduces a relative clause (Rel), and is followed by a verb phrase (VP). It is distinguished from the other **that**, which introduces a dependent clause, because it is followed by a verb phrase instead of a sentence. In other natural languages, there are often two distinct words: e.g., "qui/que" in French, "kiu/ke" in Esperanto. The end of the SQUALL query could have been formulated in a more concise but less explicit way: ... **the** `topic` **whose** `rdfs:label` **is** `"Database Management"@en`.

Literals and Datatypes. This use case is about the distinction between the lexical and semantic value of a typed literal. Two distinct lexical values may denote a same semantic value, e.g., `"8"^^xsd:integer` and `"08"^^xsd:integer`.

English	*Return all publications whose page number is the integer value 8*
SQUALL	**which** `publication` **has** `pageNumber 8`
SPARQL	`SELECT ?p WHERE { ?p :pageNumber 8 }`

The integer `8` is a custom notation for the semantic value of `"8"^^xsd:integer`, and matches equivalent lexical values such as `"08"^^xsd:integer`. The built-in predicate `str` gives access to the lexical value of typed literals.

English	*Return all publications whose page number is the lexical value "08"*
SQUALL	**which** `publication` **has a** `pageNumber` **whose** `str` **is** `"08"`
SPARQL	`SELECT ?p WHERE { ?p :pageNumber ?n. FILTER ( str(?n) = "08" ) }`

Collections and Containers. This use case is about collections and containers. There is little support for accessing their contents in existing languages.

English	*Return the last author of publication A*
SQUALL	**what is the** `last` **of the** `authorList` **of** *A*
SPARQL	`SELECT ?p WHERE {` *A* `:authorList [ rdf:rest* [ rdf:first ?p; rdf:rest rdf:nil ] ] }`

`authorList` is a property from publications to their author list, and `last` is a built-in binary predicate from lists to their last element. The expression **the** `last` **of** `?L` could be expanded as: **the** `rdf:first` **of a** `rdf:List` **whose** `rdf:rest` **is** `rdf:nil` **and that is the opt trans** `rdf:rest` **of** `?L`. Here, the reflexive and transitive closures are used to reach sublists at arbitrary depth.

Recursion. Recursion is generally absent from query languages, probably because it threatens stability with infinite loops. Another reason is that recursion is most often the duty of the reasoning mechanisms of the knowledge store: e.g., inference rules (Datalog [CGT89], SWRL), transitive properties (OWL). Recursion is introduced in SPARQL 1.1 through property path expressions, which provide a limited form of recursion that does not threaten stability, and still cover a number of use cases.

English	*Return all subtopics of topic "Information Systems", recursively*
SQUALL	**what is a trans** `subtopic` **of** `InformationSystems`
SPARQL	`SELECT ?x WHERE { :InformationSystems :subtopic+ ?x }`

Here, the property `subtopic` would better be defined as transitive (`subtopic` **is a** `owl:TransitiveProperty`) in the knowledge store. However, the underlying store may not have reasoning facilities, or one may want to use the transitive closure of a property that is not intrisically transitive (e.g., `friend`). We provide property modifiers as prefixes of binary predicates (P2): **trans** for transitive closure, **opt** for reflexive closure, and **sym** for symmetric closure. Those keywords can be seen as shorthand for adverbs: resp., "transitively", "optionally", "symmetrically".

Entailment. This use case corresponds to inference through a RDF schema, a OWL ontology, or a set of rules. Therefore, it is more the responsability of the knowledge store, than the responsability of the query engine. However, for the purpose of evaluation, let us assume the store has no inference capability.

English	*Return all instances that are members of the class Publication*
SQUALL	**what has a** `rdf:type` **that opt trans** `rdfs:subClassOf` `publication`
SPARQL	`SELECT ?p WHERE { ?p rdf:type [ rdfs:subClassOf* :publication ] }`

The explicit use of `rdf:type` and `rdfs:subClassOf` (used as a verb), plus the reflexive and transitive closure of the latter, are sufficient to answer this question. Assuming inference capability, it is enough to state (**what is a** `publication`) in SQUALL, and (`SELECT ?p WHERE { ?p a :publication }`) in SPARQL.

Closed Questions. We add a 15th use case for closed questions, which correspond to ASK-queries in SPARQL. The keyword **whether** introduces a closed question, and is followed by a coordinated sentence.

English	*Has publication X an author that works for Y ?*
SQUALL	**whether** *X* **has an** `author` **that** `worksFor` *Y*
SPARQL	`ASK` { *X* `author [ worksFor` *Y* `] }`

3.4 Updates

SQUALL updates use the same constructs as queries, and are distinguished from them because they do not contain any question word. They correspond to declarative sentences (assertions), whereas queries correspond to interrogative sentences (questions). In this section, SQUALL is compared to SPARQL/Update[2], a W3C submission that extends SPARQL. We reuse the examples given in this submission as a set of use cases.

Insertion. This use case is about the insertion of two RDF triples into the default graph of the RDF store.

English	*Add a book A with title "A new book", and with author B*
SQUALL	**there is a** `book` *A* **whose** `title` **is** `"A new book"` **and whose** `author` **is** *B*
SPARQL	`INSERT DATA` { *A* `a :book ; :title "A new book" ; :author` *B* }

The keyword **there is** acts as a global existential quantifier, and is followed by a noun phrase (NP) to form a sentence. The scope of the quantifier extends to the end of the sentence. Here, the book *A* is assumed to be new. If it is already present in the store (through existing triples), the following SQUALL sentence is more natural yet equivalent: **the** `book` *A* **has** `title "A new book"` **and has** `author` *B*.

Deletion. This use case combines a deletion and an insertion to correct a book title. The requested change happens in the named graph *G*.

English	*Correct the title of book A by replacing "Compiler Desing" by "Compiler Design", in the graph G*
SQUALL	**at** `graph` *G*, *A* **has** `title "Compiler Design"` **and not** `"Compiler Desing"`
SPARQL	`MODIFY` *G* `DELETE` { *A* `:title "Compiler Desing"` } `INSERT` { *A* `:title "Compiler Design"` }

Deletion of a triple (hence a statement) is simply expressed with negation. Note how (*A* **has** `title`) is factorized between the two titles. The prepositional phrase (**at** `graph` *G*) specifies the graph to be used for the two statements (one insertion and one deletion) in the sentence. Here, the URI `graph` denotes a binary predicate going from statements to their graph (named or default). It may be implemented either explicitly as a normal RDF property, or implicitly through the notion of default and named graphs.

Group of updates. This use case is about the ability to perform a group of updates, and corresponds to the keyword WHERE in SPARQL/Update.

English	*Delete all records of books with date before year 2000*
SQUALL	**for every** `book ?b` **that** `?p ?v` **and whose** `date < "2001-01-01"^^xsd:date,` **not** `?b ?p ?v`
SPARQL	`DELETE { ?b ?p ?v } WHERE { ?b a :book ; ?p ?v ; :date ?date. FILTER ( ?date < "2000-01-01"^^xsd:date ) }`

The keyword **for** introduces a global quantifier, and is followed by a determined noun phrase (NP) and a sentence (S) to form a sentence. The quantifier is given by the determiner, and its scope extends to the end of the coordinated sentence. Global quantifiers are the normal form in mathematics, logic, and also in notation N3, but are less common in natural language.

The WHERE-clause, which determines the number of updates to be performed, is introduced in SQUALL either by the quantifiers **every** or **the** (global or local to a NP), or by the coordination word **where** between two sentences. Therefore, equivalent formulations are:

[2] `http://www.w3.org/Submission/SPARQL-Update/`

- **every** `book` **that** `?p ?v` **and whose** `date < "2001-01-01"^^xsd:date` **not** `?p ?v`,
- **not** `?b ?p ?v` **where a** `book ?b ?p ?v` **and the** `date` **of** `?b < "2001-01-01"^^xsd:date`.

Copy. This use case copies records from one named graph G to another named graph $G2$.

English	*Copy every record from graph G to graph G2*
SQUALL	**every thing that** `?p ?v` **at graph** G `?p ?v` **at graph** $G2$
SPARQL	`INSERT INTO` $G2$ { `?b ?p ?v` } `WHERE` { `GRAPH` G { `?b ?p ?v` } }

In case of ambiguity, a prepositional phrase, here (**at** `graph` G) binds to the preceding verb rather than the following verb.

Move. This use case moves records from one named graph to another named graph. Each move combines an insertion and a deletion.

English	*Move every record from graph G to graph G2*
SQUALL	**every thing that** `?p ?v` **at graph** G `?p ?v` **at graph** $G2$ **and not** G
SPARQL	`INSERT INTO` $G2$ { `?b ?p ?v` } `WHERE` { `GRAPH` G { `?b ?p ?v` } }; `DELETE FROM` G { `?b ?p ?v` } `WHERE` { `GRAPH` G { `?b ?p ?v` } }

Here, SPARQL/Update needs two updates with a duplication of the WHERE-clause. The flexible syntax of SQUALL avoids this duplication, and also maximally factorizes the insertion and deletion. The noun phrase ($G2$ **and not** G) distributes over the rest of the verb phrase, which is therefore equivalent to (`?p ?v` **at** `graph` $G2$ **and not** `?p ?v` **at** `graph` G).

Considering the reification of triples as statements, and the representation of graphs through a binary predicate, another formulation is: **every** `rdf:Statement` **whose** `graph` **is** G **has** `graph` $G2$ **and not** G.

4 Formal Definition of the Syntax and Semantics of SQUALL

In this section, we formally define the syntax and semantics of SQUALL in the style of Montague grammars. This provides a translation from the concrete and natural syntax of SQUALL to an intermediate logical language, rather than directly in terms of an existing query language for the Semantic Web. This is a common practice in the compilation of high-level programming languages, which has a number of advantages. First, it makes the definition of semantics easier to write and understand because at a more abstract level. Second, it gives freedom in the choice of the implementation. For instance, the operational semantics of the intermediate language can be given by translating it to an existing language, e.g., SPARQL; by interpreting it in a relational algebra engine; or by using continuation passing-style, like in Prolog. In Section 5, we sketch a solution in the first approach.

We first define a core language corresponding to RDF triples as simple sentences. We then detail a number of orthogonal extensions: relational algebra, queries, quantifiers, reification and n-ary relations, built-in predicates, and aggregations with grouping. The Montague grammars given in this paper provides a self-contained specification of the syntax and semantics of SQUALL.

4.1 Triples

A triple *s p o* can be seen as a sentence. The tradition in linguistics [BJL+99] is to analyse *s* and *o* as noun phrases (*NP*), *p o* as a verb phrase (*VP*), and the whole triple as a sentence (*S*). In a Semantic Web context, a *NP* can be a term (*Term*), i.e., one of a URI, a literal, or a variable. A unary predicate (*P1*) can be used as an intransitive verb, and a binary predicate (*P2*) can be used as a transitive verb followed by an object *NP*. Verbs are ended by the third person mark −(**e**)**s** to clearly identify the predicate of the sentence, and to avoid later ambiguity. In a Semantic Web context, a *P1* is typically a class

URI, and a *P2* is typically a property URI. Variables can also be used as predicates.

$$\begin{array}{lcl} S & \rightarrow & NP\ VP\ \{\ np\ vp\ \} \\ NP & \rightarrow & Term\ \{\ \lambda d.(d\ term)\ \} \\ VP & \rightarrow & P1{-}(\mathbf{e})\mathbf{s}\ \{\ \lambda x.(p1\ x)\ \} \\ & | & P2{-}(\mathbf{e})\mathbf{s}\ NP\ \{\ \lambda x.(np\ \lambda y.(p2\ x\ y))\ \} \\ P1 & \rightarrow & ClassURI\ \{\ \lambda x.(\mathbf{type}\ x\ uri)\ \} \\ & | & Var\ \{\ \lambda x.(\mathbf{type}\ x\ var)\ \} \\ P2 & \rightarrow & PropertyURI\ \{\ \lambda x.\lambda y.(\mathbf{stat}\ x\ uri\ y)\ \} \\ & | & Var\ \{\ \lambda x.\lambda y.(\mathbf{stat}\ x\ var\ y)\ \} \end{array}$$

In the semantics, (**stat** $x\ p\ y$) denotes a triple statement $x\ p\ y$, and (**type** $x\ c$) is a shorthand for (**stat** x `rdf:type` c). We also define (**thing** x) as a shorthand for (**type** x `rdfs:Resource`). The semantics of a term, when used as a *NP*, is the set of descriptions (i.e., λd) of which the term is an instance (i.e., $d\ term$). The semantics of a verb phrase ($P2{-}\mathbf{s}\ NP$) is the description of the resources x (i.e., λx) such that the description of resources y that are connected to x through the property $p2$ (i.e., $\lambda y.(p2\ x\ y)$), is an instance of np. For instance, the sentence "*A* know-s *B*" is parsed as $S(NP(Term(A)), VP(P2(URI(\text{know})), NP(Term(B))))$, and translates to $(\lambda d.(d\ A)\ \lambda x.(\lambda d.(d\ B)\ \lambda y.(\mathbf{stat}\ x$ `know` $y)))$, which reduces to (**stat** A `know` B). This complexity makes sense when introducing quantifiers (see Section 4.5).

The Turtle notation extends the syntax of sentences from triples to simple graphs. It allows (a) to express several triples in a single sentence, (b) to conjunctively combine sentences, verb phrases, and object noun phrases, and (c) to avoid the use of some variables. For backward compatibility, SQUALL includes the Turtle constructs.

$$\begin{array}{lcl} VP & \rightarrow & \mathbf{a}\ NP\ \{\ \lambda x.(np\ \lambda y.(\mathbf{type}\ x\ y))\ \} \\ & | & P2\ NP\ \{\ \lambda x.(np\ \lambda y.(p2\ x\ y))\ \} \\ & | & \mathbf{has}\ P2\ NP\ \{\ \lambda x.(np\ \lambda y.(p2\ x\ y))\ \} \\ & | & \mathbf{is}\ P2\ \mathbf{of}\ NP\ \{\ \lambda x.(np\ \lambda y.(p2\ y\ x))\ \} \\ NP & \rightarrow & B\ \{\ \lambda d.(\mathbf{exists}\ (\mathbf{and}\ b\ d))\ \} \\ P2 & \rightarrow & B\ \{\ \lambda x.\lambda y.(\mathbf{exists}\ \lambda p.(\mathbf{and}\ (b\ p)\ (\mathbf{stat}\ x\ p\ y)))\ \} \\ B & \rightarrow & [\ ]\ \{\ \mathbf{true}\ \} \\ & | & [\ VP\ ]\ \{\ \mathbf{init}\ vp\ \} \\ S & \rightarrow & S_1\ .\ S_2\ \{\ \mathbf{and}\ s_1\ s_2\ \} \\ VP & \rightarrow & VP_1\ ;\ VP_2\ \{\ \lambda x.(\mathbf{and}\ (vp_1\ x)\ (vp_2\ x))\ \} \\ NP & \rightarrow & NP_1\ ,\ NP_2\ \{\ \lambda d.(\mathbf{and}\ (np_1\ d)\ (np_2\ d))\ \} \end{array}$$

4.2 Relational Algebra

For queries, SQUALL provides the algebraic operators of SPARQL [PAG06]: **and** for joins, **or** for unions, **not** for differences and negations, **if defined/maybe** for optional patterns. Like in Turtle, the coordination **and** can be replaced by a *dot* for sentences (S), and by a *comma* for *NP*s.

$$\begin{array}{lcl} \Delta & \rightarrow & \mathbf{not}\ \Delta_1\ \{\ \mathbf{not}\ \delta_1\ \} \\ & | & \Delta_1\ \mathbf{and}\ \Delta_2\ \{\ \mathbf{and}\ \delta_1\ \delta_2\ \} \\ & | & \Delta_1\ \mathbf{or}\ \Delta_2\ \{\ \mathbf{or}\ \delta_1\ \delta_2\ \} \\ & | & \mathbf{if\ defined/maybe},\ \Delta_1\ \{\ \mathbf{option}\ \delta_1\ \} \end{array}$$

In this rule, Δ stands for any syntagm so that algebraic operators can coordinate all kinds of constructs: e.g., "*A* or *B* work-s and cite-s *X*", which is equivalent to "*A* work-s and cite-s *X* or *B* work-s and cite-s *X*". For a same constructor to apply in constructs having different types, it is necessary to add rewriting rules like the following: **and** $\delta_1\ \delta_2\ \alpha \rightsquigarrow$ **and** $(\delta_1\ \alpha)\ (\delta_2\ \alpha)$, where α stands for any argument: e.g., an entity x if $\Delta = VP$, a description d if $\Delta = NP$.

4.3 Headed NPs, Relatives, and Auxiliary Verbs

This section augments the syntax to make it more natural and flexible. A Turtle *NP* like "[a woman; is author of *X*]" often specifies a type, here "woman", that can be made the head of the *NP*. The head is preceded by a determiner (Det), and followed by an optional term as apposition (App), and a coordination of relatives (Rel): e.g., "a woman ?A that is an author of *X*". The syntagms *NG1* and *NG2* (nominal groups) describe the possible heads, and the relative position of apposition and relatives (AR), which are both optional. *Rel* defines the possible relatives. Headed noun phrases allow for new verbal

forms based on the auxiliary verbs **is** and **has**: e.g., "is a woman", "has an author". The syntagms *NG1*, *AR*, *Rel*, *AP* denote descriptions, the syntagm *NG2* denote binary predicates, while determiners denote relations between descriptions.

$$
\begin{array}{lcl}
NP & \rightarrow & Det\ NG1\ \ \{\ \lambda d.(det\ (\textbf{init}\ ng1)\ d)\ \} \\
 & | & Det\ NG2\ \textbf{of}\ NP \\
 & & \quad \{\ \lambda d.(np\ \lambda x.(det\ (\textbf{init}\ (ng2\ x))\ d))\ \} \\
Det & \rightarrow & \textbf{a(n)}\ \ \{\ \lambda d_1.\lambda d_2.(\textbf{exists}\ (\textbf{and}\ d_1\ d_2))\ \} \\
 & | & \textbf{the}\ \ \{\ \lambda d_1.\lambda d_2.(\textbf{the}\ d_1\ d_2)\ \} \\
NG1 & \rightarrow & \textbf{thing}\ AR\ \ \{\ \textbf{and thing}\ ar\ \} \\
 & | & P1\ AR\ \ \{\ \textbf{and}\ p1\ ar\ \} \\
NG2 & \rightarrow & P2\ AR\ \ \{\ \lambda x.\lambda y.(\textbf{and}\ (p2\ x\ y)\ (ar\ y))\ \} \\
AR & \rightarrow & App\ Rel\ \ \{\ \textbf{and}\ app\ rel\ \} \ |\ App\ \ \{\ app\ \} \\
App & \rightarrow & Term\ \ \{\ \lambda x.(\textbf{eq}\ x\ term)\ \} \ |\ \epsilon\ \ \{\ \lambda x.\textbf{true}\ \} \\
Rel & \rightarrow & \textbf{that}\ VP\ \ \{\ \textbf{init}\ vp\ \} \\
 & | & \textbf{that}\ NP\ P2\text{−}\textbf{(e)s}\ \ \{\ \textbf{init}\ \lambda x.(np\ \lambda y.(p2\ y\ x))\ \} \\
 & | & \textbf{such that}\ S\ \ \{\ \textbf{init}\ \lambda x.s\ \} \\
 & | & Det\ NG2\ \textbf{of which}\ VP \\
 & & \quad \{\ \textbf{init}\ \lambda x.(det\ (ng2\ x)\ vp)\ \} \\
 & | & \textbf{whose}\ NG2\ VP \equiv \textbf{the}\ NG2\ \textbf{of which}\ VP \\
VP & \rightarrow & \textbf{is}\ AP\ \ \{\ ap\ \} \ |\ \textbf{is}\ Rel\ \ \{\ rel\ \} \\
 & | & \textbf{has}\ Det\ P2\ AR\ \ \{\ \lambda x.(det\ (p2\ x)\ ar)\ \} \\
AP & \rightarrow & Term\ \ \{\ \lambda x.(\textbf{eq}\ x\ term)\ \} \\
 & | & \textbf{a(n)/the}\ NG1\ \ \{\ ng1\ \} \\
 & | & \textbf{a(n)/the}\ NG2\ \textbf{of}\ NP \\
 & & \quad \{\ \lambda x.(np\ \lambda y.(ng2\ y\ x))\ \} \\
S & \rightarrow & S_1\ \textbf{where}\ S_2\ \ \{\ \textbf{where}\ s_1\ s_2\ \}
\end{array}
$$

The meaning of the function **init** is clarified in Section 4.6 about reification. At this point, it is equivalent to the identity function. The constructor **exists** is an existential quantification over an entity. The constructor **eq** represents equality (a binary predicate). The keywords **the** and **where** are propagated to the semantics as the constructors **the** and **where**, because their interpretation differs whether they occur in the scope of a query or an update (see Section 5).

4.4 Queries

Question words distinguish declarative sentences (updates) from interrogative sentences (queries): **whether** introduces a closed question, whose semantics encapsulates the sentence in the constructor **ask**; **what** can be used in place of any *NP* to form open questions. The constructor **select** applies to a description, and indicates that the instances of this description should be returned as query results. Multi-dimensional queries are expressed by using several occurences of **what**: e.g., "what is the author of what". Some question words are used as determiners. **which** has the same effect as **what** while allowing a restriction on returned resources: e.g., "which woman is an author of X". **how many** provides the easy expression of the most common aggregation, counting: e.g., "how many person is an author of X". The constructor **count** applies to a description, and indicates that the number of its instances should be returned as a query result.

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{whether}\ S_1\ \ \{\ \textbf{ask}\ s_1\ \} \\
NP & \rightarrow & \textbf{what} \equiv \textbf{which thing} \\
 & | & \textbf{whose}\ NG2\ AR \equiv \textbf{the}\ NG2\ AR\ \textbf{of what} \\
Det & \rightarrow & \textbf{which}\ \ \{\ \lambda d_1.\lambda d_2.(\textbf{select}\ (\textbf{and}\ d_1\ d_2))\ \} \\
 & | & \textbf{how many}\ \ \{\ \lambda d_1.\lambda d_2.(\textbf{count}\ (\textbf{and}\ d_1\ d_2))\ \}
\end{array}
$$

In order for the question constructors not to appear in the scope of non-question constructors, we introduce rewriting rules like the following: **and** (**select** $\lambda x.s_1$) $s_2 \rightsquigarrow$ **select** $\lambda x.$(**and** s_1 s_2); **exists** $\lambda x.$(**select** $\lambda y.s$) $\rightsquigarrow$ **select** $\lambda y.$(**exists** $\lambda x.s$). Similar rules are to be defined for **count**, other algebraic operators (like **and**) and other quantifiers (like **exists**), which are presented in Section 4.2 and 4.5. Also, we ensure that no **select** falls in the scope of **count** by adding the additional rewriting rule: **count** $\lambda x.$(**select** $\lambda y.s$) $\rightsquigarrow$ **select** $\lambda y.$(**count** $\lambda x.s$)

A sentence is a valid query if its semantics contains either one **ask** or any number of **select** and possibly one **count**. A sentence is a valid update if its semantics contains none of the question constructors. Otherwise, the sentence is invalid, and a syntax error should be returned.

4.5 Quantifiers

Quantifiers are commonplace in natural languages in the form of determiners, whereas they are notoriously difficult to express in SPARQL or SQL [HJ95]. The quantifier **exists** applies to a description, and checks that its extension is not empty. The quantifier **forall** applies to two descriptions, and checks that the extension of the first is included in the extension of the second. The constructor **atleast** i applies to a description, and checks that its extension has at least i elements. A possible use of them is: "every person is an author of at least 10 publication".

$$
\begin{array}{rcl}
Det & \rightarrow & \textbf{some} \ \{ \ \lambda d_1.\lambda d_2.(\textbf{exists} \ (\textbf{and} \ d_1 \ d_2)) \ \} \\
& | & \textbf{every} \ \{ \ \lambda d_1.\lambda d_2.(\textbf{forall} \ d_1 \ d_2) \ \} \\
& | & \textbf{no} \ \{ \ \lambda d_1.\lambda d_2.(\textbf{not} \ (\textbf{exists} \ (\textbf{and} \ d_1 \ d_2))) \ \} \\
& | & \textbf{at least} \ i \ \{ \ \lambda d_1.\lambda d_2.(\textbf{atleast} \ i \ (\textbf{and} \ d_1 \ d_2)) \ \} \\
S & \rightarrow & \textbf{for} \ NP \ , \ S \ \{ \ np \ \lambda x.s \ \} \\
& | & \textbf{there is} \ NP \ \{ \ np \ \lambda x.\textbf{true} \ \}
\end{array}
$$

The grammar rules are defined so that the scope of quantifiers are leftmost-outermost, and are restricted to the scope of the related verb. Therefore, "every man love-s some woman" means there is possibly a different woman for each man; while in "there is a woman that every man love-s" means there is a single woman.

The keywords **for** and **there is** introduce global quantifiers, in a style closer to mathematical logic. Their scope extends to the end of the sentence, which may be a coordinated sentence: e.g., "for every publication ?X, ?X has an author ?A and ?A cite-s ?X". Relatives introduced by **such that** can be used in a global existential quantification: "there is a person ?X such that no publication has the author ?X".

4.6 Reification and N-ary Predicates

RDF triples can be reified as statements. Therefore, it is useful to allow statements about statements. The keyword **that** turns a sentence into a noun phrase, changing the focus from the truth of the sentence to the statements involved in the sentence. The variable t is used for denoting individual statements. Here it becomes necessary to explicit two arguments shared by all constructs, and that were left implicit so far (thanks to η-equivalence of λ-calculus): $\overline{a}$ is a list of additional arguments to the predicate of the sentence, and g denotes a graph, i.e., a set of statements. Together, they form the context in which a sentence is interpreted.

In the sentence "A say-s that B is an author of X", the object of the property "say" is the statement of the triple (X `author` B). In the semantics, the main clause abstracted over its object and receiving its own context ($\lambda t.(\textbf{stat} \ A \ \texttt{say} \ t \ \overline{a} \ g)$) is passed as a graph to the dependent clause (**stat** X `author` B). This generalizes the GRAPH construct in SPARQL, where a graph would be restricted to the form $\lambda t.(\textbf{stat} \ t \ \texttt{graph} \ G)$, where `graph` would be the binary predicate from statements to graphs, and G would denote a SPARQL graph (a URI or a variable).

The other rules define prepositional phrases (PP), and their inclusion in sentences. A PP is introduced by **at**, and is followed by a URI and a NP. It modifies a sentence by adding an argument to its context (constructor **arg**). Each argument is a pair ($uri, term$), where uri defines the role of the argument w.r.t. the statement, much like "rdf:subject", "rdf:predicate", and "rdf:object"; and $term$ is the argument itself. Those arguments provide means for expressing n-ary relationships, without introducing dummy blank nodes. For example, "A is an author of X at rank 1" means that A is the first author of X; and "A sell-s X at year 2009 at amount 10" means that A sold 10 units of X in 2009.

The remaining rules say that prepositional phrases can occur anywhere in a sentence, and the syntagms OP (object phrase) and CP (complement phrase) are introduced to retain the leftmost-outermost scope rule for quantifiers. Therefore, the rigidity of this rule is balanced by the free ordering of PPs, the possible inversion between subject and object (**of**), and the global quantifiers introduced by **for**. An example of query that uses a number of those constructs is: "at which venue every professor that work-s at place X is a speaker at some time".

$$
\begin{array}{lcl}
NP & \rightarrow & \textbf{that}\ S\ \ \{\ \lambda d.\lambda\overline{a}.\lambda g.(s\ ()\ \lambda t.(d\ t\ \overline{a}\ g))\ \} \\
PP & \rightarrow & \textbf{at}\ URI\ NP\ \ \{\ \lambda s.(np\ \lambda z.(\textbf{arg}\ uri\ z\ s))\ \} \\
 & | & \textbf{at}\ Det\ URI\ AR \equiv \textbf{at}\ URI\ Det\ \textbf{thing}\ AR \\
S & \rightarrow & PP\ S\ \ \{\ pp\ s\ \} \\
VP & \rightarrow & PP\ VP\ \ \{\ \lambda x.(pp\ (vp\ x))\ \} \\
 & | & P1{-}(\textbf{e})\textbf{s}\ CP\ \ \{\ \lambda x.(cp\ (p1\ x))\ \} \\
 & | & P2{-}(\textbf{e})\textbf{s}\ OP\ \ \{\ \lambda x.(op\ (p2\ x))\ \} \\
 & | & \textbf{is}\ AP\ CP\ \ \{\ \lambda x.(cp\ (ap\ x))\ \} \\
 & | & \textbf{has}\ Det\ P2\ AR\ CP \\
 & & \quad \{\ \lambda x.(det\ \lambda y.(cp\ (p2\ x\ y))\ ar)\ \} \\
OP & \rightarrow & PP\ OP\ \ \{\ \lambda d.(pp\ (op\ d))\ \} \\
 & | & NP\ CP\ \ \{\ \lambda d.(np\ \lambda y.(cp\ (d\ y)))\ \} \\
CP & \rightarrow & PP\ CP\ \ \{\ \lambda s.(pp\ (cp\ s))\ \}\ \mid\ \epsilon\ \ \{\ \lambda s.s\ \} \\
Rel & \rightarrow & \textbf{at which}\ URI\ AR\ S \\
 & & \quad \{\ \textbf{init}\ \lambda x.(\textbf{and}\ (ar\ x)\ (\textbf{arg}\ uri\ x\ s))\ \}
\end{array}
$$

It is now possible to clarify and define the constructors **init** and **arg** that have already been used. The constructor **init** reinitializes the list of arguments in some construct: $\textbf{init} = \lambda d.\lambda x.\lambda\overline{a}.(d\ x\ ())$. This is useful to restrict the passing of arguments to the predicate of a sentence, and to avoid its propagation to noun groups and relatives for instance. The constructor **arg** adds an argument to the current list of arguments, waiting to be passed to the main predicate of the sentence: $\textbf{arg} = \lambda uri.\lambda z.\lambda s.\lambda\overline{a}.(s\ ((uri, z), \overline{a}))$.

In order to pass the context down to arguments of algebraic operators, quantifiers, and question constructors, it is necessary to define rewriting rules like the following: $\textbf{true}\ \overline{a}\ g \rightsquigarrow \textbf{true}$, $\textbf{and}\ s_1\ s_2\ \overline{a}\ g \rightsquigarrow \textbf{and}\ (s_1\ \overline{a}\ g)\ (s_2\ \overline{a}\ g)$, $\textbf{exists}\ d\ \overline{a}\ g \rightsquigarrow \textbf{exists}\ \lambda x.(d\ x\ \overline{a}\ g)$.

4.7 Built-in Predicates and Aggregations

Built-in predicates can be used as unary predicates (*Pred1URI*) like class URIs, and as binary predicates (*Pred2URI*) like property URIs. A built-in predicate expects a list of arguments, which may include the subject and object. We add a rule for sentences based on a built-in predicate that expects neither a subject nor an object (*Pred0URI*).

The head of a noun phrase (*NG1*) can be an aggregator followed by the set of what should be aggregated, and optionally followed by a list of grouping dimensions introduced by **per**. Each dimension is an *AP* that specifies the set of possible values for this dimension: e.g., "what is the count of the publication ?P per the year of ?P, and the affiliation of an author of ?P". This construction produces bindings for the aggregated value and each dimension.

$$
\begin{array}{lcl}
P1 & \rightarrow & Pred1URI\ \ \{\ \lambda x.(\textbf{pred1}\ uri\ x)\ \} \\
P2 & \rightarrow & Pred2URI\ \ \{\ \lambda x.\lambda y.(\textbf{pred2}\ uri\ x\ y)\ \} \\
S & \rightarrow & Pred0URI{-}(\textbf{e})\textbf{s}\ CP\ \ \{\ cp\ (\textbf{pred0}\ uri)\ \} \\
NG1 & \rightarrow & AggregURI\ \textbf{of}\ AP\ (\textbf{per}\ AP_i^+)? \\
 & & \quad \{\ \lambda x.(\textbf{aggreg}\ uri\ x\ ap\ (ap_i)_i)\ \}
\end{array}
$$

4.8 Handling of Ambiguity

The price for the natural and flexible syntax of SQUALL is ambiguity, i.e., the fact that some sentences can be parsed in different ways possibly leading to different semantics. In SQUALL, ambiguities are resolved by the following rules:

1. when forming a construct Δ from one or two constructs of same syntagm Δ (e.g., coordinating 2 *NP*s, modifying a sentence with a *PP*), algebraic operators have priority (in decreasing priority order: **not**, **and**, **or**, **if defined**/**maybe**, **where**) over sentence modifiers (*PP* as a prefix, and global quantifiers **for** *NP*). Punctuation has lowest priority, and right-associativy is used for binary coordinations;
2. smaller syntagms have priority over larger syntagms, i.e., in decreasing priority order: *P1*, *P2*, *Det*, *Rel*, *NG1*, *NG2*, *AP*, *NP*, *PP*, *CP*, *OP*, *VP*, *S*;
3. in case of ambiguity between forming two constructs of same syntagm, the shorter construct is chosen.

Round brackets can be used for every syntagms to escape those rules. Rule 2 implies that "a man or woman" is interpreted as "a (man or woman)" rather than "(a man) or woman", as *NG1* has priority over *NP*.

5 Translation to SPARQL

An operational semantics can be given to SQUALL by translating its intermediate language to SPARQL. Given a SQUALL sentence S with semantics s, the translation of S in the intermediate language is the formula $f = s\ ()\ g_0$, i.e., the sentence initialized with an empty list of arguments, and some default graph g_0. In order to simplify this formula, we remove some of the constructors of the intermediate language by giving them a definition. For example, the triple $x\ p\ y$ passed to the constructor **stat** is reified by stating the existence of a statement resource t, which is connected to its subject, predicate, object, and arguments through roles. Each connection is represented using the new constructor **triple** that represents a non-reifiable triple. Those connections are completed by stating that the statement t belongs to the given graph g. The constructor **eq** is defined by the built-in predicate $=$. The constructors **atleast** and **count** are defined in terms of the aggregator `COUNT`, and the built-in predicate $\leq$. The constructor **fold** is the classical iterator on lists. A term (**fold** $f\ e\ (x, \overline{x})$) reduces to (**fold** $f\ (f\ e\ x)\ \overline{x}$), and a term (**fold** $f\ e\ ()$) reduces to e.

$$
\begin{aligned}
&\textbf{stat} = \lambda x.\lambda p.\lambda y.\lambda \overline{a}.\lambda g.(\textbf{exists}\ \lambda t.(\textbf{fold}\\
&\quad \lambda q.\lambda(uri, z).(\textbf{and}\ q\ (\textbf{triple}\ t\ uri\ z))\ (g\ t)\\
&\quad ((\text{rdf:subject}, x), (\text{rdf:predicate}, p), (\text{rdf:object}, y), \overline{a})))\\
&\textbf{eq} = \lambda x.\lambda y.(\textbf{pred2}\ =\ x\ y)\\
&\textbf{atleast} = \lambda i.\lambda d.(\textbf{exists}\ \lambda x.(\\
&\quad \textbf{and}\ (\textbf{aggreg}\ \texttt{COUNT}\ x\ d)\ (\textbf{pred2}\ \leq\ i\ x)))\\
&\textbf{count} = \lambda d.(\textbf{select}\ \lambda x.(\textbf{aggreg}\ \texttt{COUNT}\ x\ d))
\end{aligned}
$$

After applying those definitions, the only remaining constructors are: **triple**, **pred0**, **pred1**, **pred2**, **aggreg**, **true**, **not**, **and**, **or**, **option**, **where**, **exists**, **forall**, **the**, **ask**, **select**. We define in the following their translation to the queries and updates of SPARQL. Those translations are chosen to be concise, and not to be optimal in any way. The SPARQL translation of a formula f is denoted by $[f]$, and $[X \mid f]_Q$ is an auxialiary translation for multi-dimensional queries. The two other auxiliary translations are $[f]_G$ for producing graph patterns (generation of bindings), and $[f]_U$ for producing updates (insertion and deletion of triples). An update is a triple (I, D, G), where I is a graph to be inserted, D is a graph to be deleted, and G is a graph pattern.

$$
\begin{aligned}
&[\textbf{ask}\ f] = \texttt{ASK}\ [f]_G\\
&[\textbf{select}\ d] = [?x \mid d\ ?x]_Q\\
&[f] = \texttt{MODIFY INSERT}\ \{I\}\ \texttt{DELETE}\ \{D\}\\
&\quad \texttt{WHERE}\ G \quad \text{where } (I, D, G) = [f]_U\\
&[X \mid \textbf{select}\ d]_Q = [X\ ?x \mid d\ ?x]_Q\\
&[X \mid f]_Q = \texttt{SELECT}\ X\ \texttt{WHERE}\ [f]_G
\end{aligned}
$$

Formulas with constructors **ask** and **select** translate to the corresponding SPARQL queries, while other formulas translate to SPARQL updates. Every occurence of a SPARQL variable $?x$ assumes the generation of a fresh variable name. Those variables are used to instantiate description parameters of question, quantifier, and aggregation constructors.

$$
\begin{aligned}
&[\textbf{triple}\ s\ p\ o]_G = s\ p\ o\ .\\
&[\textbf{pred0}\ pred\ \overline{a}\ g]_G = \texttt{FILTER}\ pred(\overline{a})\\
&[\textbf{pred1}\ pred\ x\ \overline{a}\ g]_G = \texttt{FILTER}\ pred(x, \overline{a})\\
&[\textbf{pred2}\ pred\ x\ y\ \overline{a}\ g]_G = \texttt{FILTER}\ pred(x, y, \overline{a})\\
&[\textbf{aggreg}\ agg\ x\ d\ (d_i)_i\ \overline{a}\ g]_G =\\
&\quad \{\ \texttt{SELECT}\ (?z_i)_i\ (agg(?y)\ \text{AS}\ ?x)\\
&\quad\ \texttt{WHERE}\ [\textbf{fold and}\ (d\ ?y)\ (d_i\ ?z_i)_i]_G\\
&\quad\ \texttt{GROUP BY}\ (?z_i)_i\ \}\\
&[\textbf{true}]_G = \{\ \}\\
&[\textbf{and}\ f_1\ f_2]_G = \{\ [f_1]_G\ [f_2]_G\ \}\\
&[\textbf{or}\ f_1\ f_2]_G = \{\ [f_1]_G\ \texttt{UNION}\ [f_2]_G\ \}\\
&[\textbf{option}\ f]_G = \texttt{OPTIONAL}\ [f]_G\\
&[\textbf{not}\ f]_G = \texttt{FILTER NOT EXISTS}\ [f]_G\\
&[\textbf{where}\ f_1\ f_2]_G = [\textbf{and}\ f_1\ f_2]_G\\
&[\textbf{exists}\ d]_G = [d\ ?x]_G\\
&[\textbf{forall}\ d_1\ d_2]_G = [\textbf{not}\ (\textbf{exists}\ (\textbf{and}\ d_1\ (\textbf{not}\ d_2)))]_G\\
&[\textbf{the}\ d_1\ d_2]_G = [\textbf{exists}\ (\textbf{and}\ d_1\ d_2)]_G
\end{aligned}
$$

Built-in predicates translate to SPARQL filters, and aggregations translate to SPARQL aggregative sub-queries. Arguments can be used for n-ary predicates, but there is no counterpart in SPARQL for aggregations. Algebraic constructors

translate to their SPARQL counterpart, and quantifiers all translate to the implicit SPARQL existential quantifier and negation.

$$
\begin{array}{l}
[\textbf{triple}\ s\ p\ o]_U = (s\ p\ o\ .,\epsilon,\epsilon) \\
[\textbf{true}]_U = (\epsilon,\epsilon,\epsilon) \\
[\textbf{and}\ f_1\ f_2]_U = (I_1\ I_2, D_1\ D_2, \{\ G_1\ G_2\ \}) \\
[\textbf{not}\ f]_U = (D, I, G) \quad \text{where } (I, D, G) = [f]_U \\
[\textbf{where}\ f_1\ f_2]_U = (I_1, D_1, \{\ G_1\ [f_2]_G\ \}) \\
\quad \text{where } (I_1, D_1, G_1) = [f_1]_U \\
[\textbf{exists}\ d]_U = [d\ ?x]_U \\
[\textbf{forall}\ d_1\ d_2]_U = [\textbf{where}\ (d_2\ ?x)\ (d_1\ ?x)]_U \\
[\textbf{the}\ d_1\ d_2]_U = [\textbf{where}\ (d_2\ ?x)\ (d_1\ ?x)]_U
\end{array}
$$

Compare the translation of **where** with graph patterns. In updates, it introduces a graph pattern, whereas in a graph pattern, it refines it like **and**. The same can be said for the quantifier **the**.

6 Implementation

We have implemented a SQUALL interpreter, and experimented it on top of a custom RDFS store. It could easily be adapted to another RDFS store, or as a compiler to low-level querying and update language, like SPARQL 1.1 Update. The implementation is organized along the principles of programming language compilers [ASU86], as a sequence of phases: lexical analysis, syntactic analysis, translation to an intermediate representation, optimization, and finally either interpretation or translation to a target language. The lexical analysis recognizes keywords, URIs in their various forms (full, qualified, bare), literals, and variables. The syntactic analysis is based on descending parser, fed with the context-free grammar of SQUALL. The intermediate representation is based on a logical language made of connectors, quantifiers, and relation closures, whose semantics is defined in the same relational algebra as SPARQL. The translation to this logical language is done in the compositional style of Montague's semantics, which has been successfully applied to the translation of natural languages into logic [DWP81]. This is the most important phase as it reduces the high-level and flexible syntax of SQUALL to a low-level and stricter language, while preserving semantics. The result of the translation can easily be converted to an existing language such as SPARQL as they share the same semantics and level of detail. Alternately, the intermediate representation can directly be interpreted to perform updates or to answer queries, using techniques similar to other RDF query languages. Parsing and translations are better addressed by functional and logical programming languages, because they are all but manipulation of symbolic structures. We have implemented it in a functional language of the ML family, Objective Caml[3].

7 Discussion and Future Work

SQUALL qualifies as a Controlled Natural Language (CNL) because it is a formal language with a (controlled) natural syntax. However, it departs from existing CNLs in two ways. First, SQUALL is fully compliant with the RDF data model, and does not require augmenting RDF in any way, unlike other approaches [FS07]. Second, SQUALL does not require any linguistic knowledge, e.g., the plural of nouns or the passive form of verbs, and can thus be applied on existing RDF data without additional annotation. The consequence of this is a slighty less natural syntax, but also a simpler grammar. Therefore, SQUALL stands at an intermediate position between CNLs and Semantic Web languages. It has the semantics, expressiveness, safety and interoperability of SPARQL, and the abstraction, flexibility and usability of CNLs.

Another difference between SQUALL and other CNLs is that SQUALL targets queries and updates, while most CNLs for the Semantic Web target ontologies [SKC+08], like SOS and Rabbit. ACE is more general, and supports both assertions and questions. Descriptions are also covered by SQUALL, as they form a subset of updates. Most OWL axioms and SWRL rules could also be expressed in SQUALL, but they often conflict with group of updates. For instance, should the sentence (**every** `professor` **whose** `affiliation` **is** A **is an** `author` **of** B) be understood as an update, applying to the *already* known professors, or as an inference rule, also applying to *future* professors?

The syntax and semantics of SQUALL suggests some usage for naming classes and properties, and for conceptualization. As the grammatical words **is**, **of**, **has** are part of the SQUALL grammar, it becomes possible, and advisable, to lemmatize URI fragments. For instance, a property like `hasFather` can be renamed as `father`, and `subclass` could be used as the inverse of `subClassOf`. When representing n-ary relations, prepositions can be used to explicit the role of additional arguments, instead of introducing blank nodes. For example, instead of (`Chutney hasIngredient [ ingredient greenMango; amount "500g" ]`) [HKR09], where it is not clear what is the object of `hasIngredient`, we suggest the SQUALL sentence `Chutney` **has** `ingredient greenMango` **at** `amount "500g"`. In the latter, the prepositional phrase about the amount is optional, and

[3] `http://caml.inria.fr/`

does not interfere with the main subject-predicate-object structure. To find the ingredients of Chutney, it suffices to ask the question: **what is an** `ingredient` **of** `Chutney`.

In order to improve the expressivity and compliance of SQUALL, we plan to integrate query modifiers, and to add support for handling collections and containers. Query modifiers alter the results of a query by removing duplicates (DISTINCT), ordering results (ORDER BY), or controlling the amount of results (LIMIT and OFFSET).

A CNL is not sufficient for a successful user interaction, and our main objective is to integrate SQUALL into Logical Information Systems (LIS) [FH11, HFD11]. If it is much easier to read a CNL compared to a formal language, it is still difficult for a user to write assertions and questions. The first encountered difficulty is learning the syntax and the disambiguation rules, but the main difficulty is about the vocabulary, which is strongly dependent on the dataset at hand. Predictive parsers like Ginseng [BKK05] are helpful in this regard. However, this approach is not sufficient to avoid empty results for questions, and empty results are only discovered once the question has been entered entirely. For updates, suggestions of predictive parsers are hardly informative because they do not take into account existing data. The LIS approach is to support exploratory search [Mar06], where at each step a valid query or update is presented to the user, and dataset-aware suggestions are given to further refine the query or update, and to support "at a glance" sense making.

8 Conclusion

SQUALL is a Semantic Query and Update High-Level Language that provides a controlled natural syntax on top of SPARQL 1.1 Update, while preserving adequacy, interoperability, and expressiveness. Its syntax and semantics are formally defined as a Montague grammar. The semantics of SQUALL sentences are formulated in a logical intermediate language. We have sketched a translation from this intermediate language to SPARQL, thus providing an operational semantics and possible implementation for SQUALL.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AvH04] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. MIT Press, 2004.

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BJL+99] Douglas Biber, Stig Johansson, Geoffrey Leech, Susan Conrad, and Edward Finegan. *Longman grammar of spoken and written English*. Pearson Education Limited, 1999.

[BKK05] A. Bernstein, E. Kaufmann, and C. Kaiser. Querying the semantic web with Ginseng: A guided input natural language search engine. In *Work. Information Technology and Systems (WITS)*, 2005.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[DWP81] D. R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague Semantics*. D. Reidel Publishing Company, 1981.

[FH11] S. Ferré and A. Hermann. Semantic search: Reconciling expressive querying and exploratory search. In L. Aroyo and C. Welty, editors, *Int. Semantic Web Conf.*, LNCS 7031, pages 177–192. Springer, 2011.

[FKS06] N. E. Fuchs, K. Kaljurand, and G. Schneider. Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In G. Sutcliffe and R. Goebel, editors, *FLAIRS Conference*, pages 664–669. AAAI Press, 2006.

[FS07] N. E. Fuchs and R. Schwitter. Web-annotations for humans and machines. In E. Franconi, M. Kifer, and W. May, editors, *European Semantic Web Conference*, LNCS 4519, pages 458–472. Springer, 2007.

[HBEV04] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of RDF query languages. In S.A. McIlraith *et al.*, editor, *Int. Semantic Web Conf.*, LNCS 3298, pages 502–517. Springer, 2004.

[HFD11] A. Hermann, S. Ferré, and M. Ducassé. Guided creation and update of objects in rdf(s) bases. In M. A. Musen and Ó. Corcho, editors, *Int. Conf. Knowledge Capture (K-CAP)*, pages 189–190. ACM, 2011.

[HJ95] P.-Y. Hsu and D. S. Parker Jr. Improving SQL with generalized quantifiers. In P. S. Yu and A. L. P. Chen, editors, *Int. Conf. Data Engineering*, pages 298–305. IEEE Computer Society, 1995.

[HKR09] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[Mar06] G. Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[Mon70] R. Montague. Universal grammar. *Theoria*, 36:373–398, 1970.

[PAG06] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In I. F. Cruz *et al*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 30–43. Springer, 2006.

[SKC+08] R. Schwitter, K. Kaljurand, A. Cregan, C. Dolbear, and G. Hart. A comparison of three controlled natural languages for OWL 1.1. In K. Clark and P. F. Patel-Schneider, editors, *Workshop on OWL: Experiences and Directions (OWLED)*, volume 258. CEUR-WS, 2008.

[Sma08] P. Smart. Controlled natural languages and the semantic web. Technical report, School of Electronics and Computer Science University of Southampton, 2008.