SparqlBlocks: Using Blocks to Design Structured Linked Data Queries

Miguel Ceriani[1,2] and Paolo Bottoni[1]
[1]Sapienza, University of Rome, Italy
[2]Instituto Tecnológico de Buenos Aires, Argentina
{ceriani,bottoni}@di.uniroma1.it

Abstract *While many Linked Data sources are available, the task of building structured queries on them is still a challenging one for users who are not conversant in the specialised query languages required for their effective use. A key hindering factor is the lack of intuitive user interfaces for these languages. The block programming paradigm is becoming popular for the development of visual interfaces that are easy to use and guaranteed to generate syntactically correct programs, promoting a gradual and modular approach to the task of programming. We exploit these features of the block paradigm to develop SparqlBlocks, a visual language and an integrated user interface in which both Linked Data queries and results are represented as blocks, supporting a modular and exploratory approach to query design. By integrating the presentation of queries and results, reuse of results in the refinement of queries is promoted, as well as the exploration of both the data and the structure of Linked Data sources. SparqlBlocks has been evaluated with 11 users literate in computer science but with small to no expertise in querying Linked Data. After a tutorial, all the users were able to build at least a simple query and all but two were able to build nontrivial queries.*

1. Introduction

Linked Data [1] — the structured data available online — are increasing both in quantity and diversity [2]. A key advantage of the Linked Data model is its support for serendipitous exploration and reuse of existing data. In practice, though, exploring and querying Linked Data is not trivial and requires knowledge of RDF [3] (the basic data model), SPARQL [4] (the standard query language), and a number of schemas and ontologies (the domain/dataset specific data models).

Existing experimental tools for non-experts (see for example [5, 6]), while being effective for some cases, do not support the user much in an incremental process of query design, as reusing intermediate queries and results for new queries is not easy.

As block programming languages [7, 8] — in which coding occurs by dragging and connecting fragments shaped like jigsaw puzzle pieces — have been successfully adopted to introduce programming to non-experts, we decided to leverage the block programming paradigm to design Linked Data queries, supporting an exploratory approach to query design in which language affordances are visually exposed and syntax errors are avoided. Hence, users are not required to know in advance the details of RDF and SPARQL, while the flexibility and expressiveness of a complex query language are preserved.

A specific challenge in querying Linked Data is supporting their heterogeneous nature: no expert can achieve perfect knowledge of the structure and semantics of all the data they may need to use from multiple sources; systems should thus help in discovering structure and semantics of data sources. To deal with this, we build on the block programming approach, presenting a novel paradigm for interactive queries in which both the queries and their results are integrated and interoperable in the workspace. Results are available as blocks that can be used as part of existing queries (to refine them) or to build new (follow-up) queries. Queries are dynamically executed as they are built or modified. As a result, the supporting environment fosters an exploratory approach such that users may start querying datasets without knowing their specific organization and gather progressively more detailed information.

The described approach has been implemented in SparqlBlocks[1] [9, 10], a visual language and an associated visual environment for designing and executing queries on Linked Data sources. The target user of the system is anyone interested in building queries beyond simple data browsing, on one or multiple datasets. The tool may also work as an educational aid for learning Linked Data technologies.

In this paper, we describe in detail the language and user interface of SparqlBlocks and report on a controlled user evaluation aimed at proving that the SparqlBlocks environment may be used to build nontrivial queries on Linked Data without prior knowledge of RDF, SPARQL, or even of the data source's content and structure.

Paper organisation. After introducing the technological background in Section 2, we present and discuss related work in Section 3. In Section 4 we describe the SparqlBlocks visual language and environment, reviewing their specific requirements and features, while in Section 5 we describe its implementation. An analysis from the perspectives of cognitive dimensions and query affordances is given in Section 6, followed in Section 7 by the description of preliminary informal

DOI reference number: 10.18293/VLSS2017-011

[1]http://sparqlblocks.org/

feedback from users and how, based on that feedback, the design evolved. The tool has been evaluated in a user study, described in Section 8, that involved 11 users without prior knowledge of either RDF, SPARQL, or the structure and content of the used dataset. The experimental results are shown both from a quantitative and a qualitative point of view in Sections 9 and 10, respectively. As pointed out in the conclusive discussion in Section 11, the results are encouraging, as users were satisfied by the user interface and able to build nontrivial queries, but they also reveal a number of possible improvements for the tool.

2. Background

We introduce the basics of the RDF data model, the SPARQL query language, and DBpedia, a dataset that will be used for examples and evaluation. We then briefly describe the block programming paradigm.

2.1. Data Model: RDF

In the Resource Description Framework (RDF) [3], the data model proposed by the W3C for Linked Data, knowledge is represented via *RDF statements* about *resources*, which are meant to represent anything in the "universe of discourse", e.g. documents, people, physical objects, abstract concepts. An RDF statement is represented by an *RDF triple*, composed of *subject* (a resource), *predicate* (specified by a resource as well, called a *property*), and *object* (a resource or a *literal*, i.e. a value from a basic type). An *RDF graph* is a set of RDF triples.

A literal is a simple value that can be either a language-tagged string — a string associated with a language tag that identifies the (natural) language of the label — or a typed literal — a value expressed with a string and an associated type that may be any IRI, but which is usually one of the basic datatypes defined by W3C for the XML Schema Definition Language (XSD) 1.1 [11].

Resources are identified by an *Internationalized Resource Identifier* (IRI) [12], a generalization of the *Uniform Resource Identifier* (URI) [13] for retrieving content in an HTTP context. A resource may have one or more *types*, which are also resources. If a resource is used as type for other ones, it is called a *class*. A human-readable version of a resource's name is a string literal, called its *label*. A resource may have one or more labels. Thanks to language tags, a resource can have labels in different languages.

An *RDF dataset* is a set of *named RDF graphs*, i.e., RDF graphs associated with an IRI, the *graph name*, along with a single *default graph*, an RDF graph without a name. Named RDF graphs are used to represent data associated with specific contexts. Usually the default graph of an RDF dataset is either the union of all the named graphs or holds meta-data about the named graphs.

RDF graphs and RDF datasets can be serialized through different concrete RDF syntaxes (Turtle, JSON-LD, XML/RDF). A common feature of multiple RDF syntaxes (used also in SPARQL) is that *prefixes* can be used in place of the initial part of an IRI, which represents specific namespaces for vocabularies or sets of resources. For example, the two IRI namespaces for standard RDF concepts[2] are usually referred to via the prefixes `rdf:` and `rdfs:`, as in `rdf:type`, which is the property used to associate a resource with its type(s) and in `rdfs:label`, which is the property used to associate a resource with its label(s).

Figure 1 shows a graphical depiction of an RDF graph where resources are represented as ovals, literals as rectangles, and triples as labelled arrows connecting them (from subject to object, while the label represents the predicate). The labels for resources and predicates (that are resources too) are IRIs in prefix notation, while the labels for literals are the lexical representation of the literals.

In Listing 1, the same RDF graph is represented using Turtle, an RDF syntax offering prefix notation. Turtle allows authors to avoid repeating the subject of a sequence of triples when it is the same. The semicolon (;) separates predicate/object pairs that apply to the same subject. The dot (.) separates blocks of triples having a common subject.

```
@base            <http://www.w3.org/> .
@prefix rdf:     <1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:    <2000/01/rdf-schema#> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix dbo:     <http://dbpedia.org/ontology/> .

dbpedia:Mount_Everest
    rdf:type dbo:Mountain ;
    rdfs:label "Mount␣Everest"@en ;
    dbo:elevation 8848 ;
    dbo:locatedInArea dbpedia:Nepal ;
    dbo:locatedInArea dbpedia:China .

dbpedia:K2
    rdf:type dbo:Mountain ;
    rdfs:label "K2"@en ;
    dbo:elevation 8611 ;
    dbo:locatedInArea dbpedia:China .

dbpedia:Nepal
    rdf:type dbo:Country ;
    rdfs:label "Nepal"@en .

dbpedia:China
    rdf:type dbo:Country ;
    rdfs:label "China"@en .
```

Listing 1. Turtle code for the RDF graph in Figure 1.

2.2. Query Language: SPARQL

SPARQL [4] is the standard query language for RDF datasets[3], based on the notion of *triple pattern*, an RDF triple in which each component can be replaced by a variable. A *basic graph pattern* is a set of triple patterns associated with

[2]http://www.w3.org/1999/02/22-rdf-syntax-ns# and http://www.w3.org/2000/01/rdf-schema#

[3]For conciseness, rather than describing SPARQL syntax in detail, we just show the basics of SPARQL semantics and then a few elements of the syntax. This should be sufficient to both (1) have an idea of the language and (2) understand the SPARQL queries that are shown in the paper in comparison with SparqlBlocks syntax.

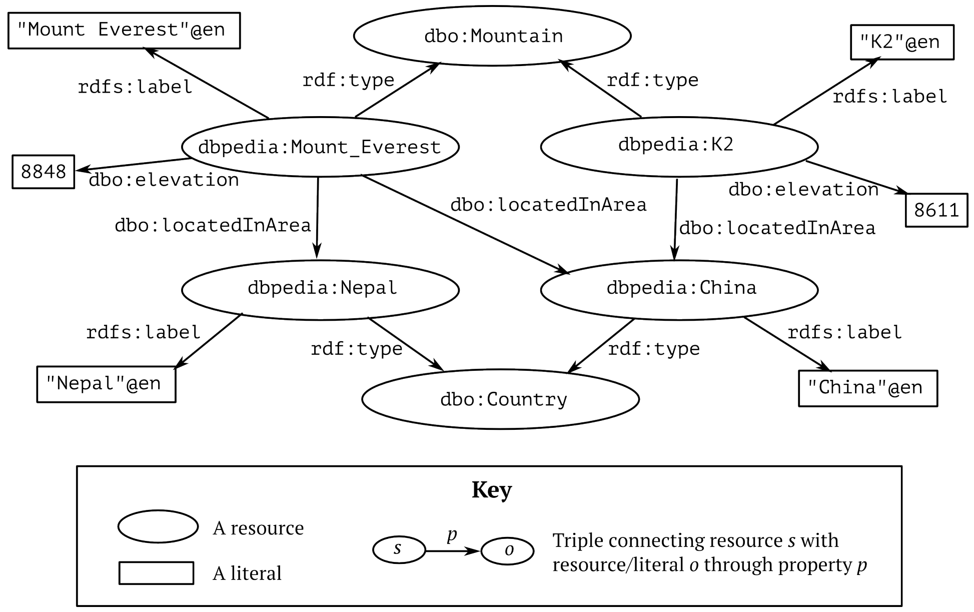


Figure 1. Part of the RDF graph served by the SPARQL endpoint of DBpedia.

a specific input graph (the default graph, a graph named by a IRI, or a generic named graph referred to via a variable). When executing a SPARQL query, the basic graph pattern is matched against the input RDF dataset and the result is a multiset of tuples, each tuple corresponding to a binding for each of the variables. Relations generated through basic graph patterns can be filtered, composed, or grouped using relational operators. The result of a SPARQL SELECT query (one of the available query types and the one that will be considered in this work) is a multiset of tuples that can be optionally ordered, making it a list of tuples.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height .
}
ORDER BY DESC(?height)
LIMIT 3
```

Listing 2. SPARQL corresponding to blocks in Figure 11.

Table 1. Results of query in Listing 2.

?mount	?height
<http://dbpedia.org/resource/Mount_Everest>	8848.0
<http://dbpedia.org/resource/K2>	8611.0
<http://dbpedia.org/resource/Kangchenjunga>	8586.0

SPARQL syntax is reminiscent of SQL syntax and clauses like SELECT, ORDER BY, and LIMIT have in SPARQL the same meaning as in SQL. Basic graph patterns are represented reusing Turtle syntax for RDF graphs with the addition of variables, which are represented by labels prefixed by quotation mark and have as scope the whole query[4]. Both basic graph patterns and operations like filter or union are specified in the WHERE clause of the query. Listing 2 shows a query to get the three highest mountains. Table 1 shows the results of the query when executed on the RDF graph in Figure 1.

The *SPARQL Protocol* [14] is a protocol over HTTP used to provide a Web API to query an RDF dataset. The client sends queries to a service offering this API and gets back the results. The URI at which a SPARQL Protocol service listens for requests is called a *SPARQL endpoint*. Several RDF datasets offer publicly accessible SPARQL endpoints.

2.3. A Reference Dataset: DBpedia

Both in the examples and in the user evaluation we used a well known Linked Data source: DBpedia [15], an RDF dataset generated from Wikipedia. It is obtained through a set of scripts that extract structured data from the Wikipedia pages, especially leveraging the content of infoboxes, fixed-format tables that can be found in the right-hand corner of some articles. In the Linked Data community, DBpedia is widely considered a reference dataset because of its wide coverage and the long-term continuity of the project.

2.4. Block Programming

Programming environments for block languages typically offer a visual, drag-and-drop interface. Blocks are graphical elements associated with code snippets, which can be connected to one another like jigsaw puzzle pieces, at the same time composing in a syntactically correct form the associated code. Different kinds of blocks have different shapes, so that

[4]When using subqueries, which are neither described here nor available in SparqlBlocks, the scope of a variable is instead restricted to the current subquery, unless it is projected out in the SELECT clause

the way they can be combined is visually apparent, exposing the affordances of the language and preventing syntax errors. In Section 4.3 we will present in detail the features of block programming environments that we have adopted in SparqlBlocks.

From now on, we adopt for block programming concepts the notation used in the documentation and code of Blockly [16], a widely used block programming library on which our tool is based.

3. Related Work

We report on visual query tools for Linked Data as well as on block programming environments.

3.1. Structured Queries on Linked Data

Several interactive tools have been proposed to support structured querying of RDF data sources, at various levels of abstraction and using different paradigms. A basic distinction can be made between tools: (1) those requiring usage of SPARQL syntax and (2) those based on metaphors aimed at lowering the learning curve and providing more intuitive interaction. The first kind of UIs includes advanced editors, e.g. YASGUI [17], or integrated environments, e.g. Twinkle[5], but users still have to know SPARQL and the vocabularies used in order to design a query.

UIs of the second kind provide interaction with another — textual or visual — representation of the query, then transformed to SPARQL to be executed. Text-based UIs use forms, e.g. SPARQLViz [18], or controlled construction of natural language statements, e.g. SPARKLIS [5]. These systems do not scale well when the query complexity increases and do not easily permit code reuse. As for visual tools, most of them use a graph-based paradigm (e.g. NITELIGHT [6], QueryVOWL [19]), others a dataflow-based paradigm (e.g. SparqlFilterFlow [20]), and at least one, VQS [21], a combination of both. Graph-based interfaces fit the RDF graph pattern matching model well and they can be see as one possible RDF embodiment of the successful *query-by-example* paradigm, in which the user defines queries by perusing the same structure of the data (tables in the original definition for relational databases [22], graphs in the case of RDF). Dataflow-based interfaces, on the other side, are effective in representing SPARQL functional operators (e.g., `UNION`). However, both types of interfaces are inefficient in terms of screen real estate and may present problems with interaction.

There is an existing tool that adopts the block programming paradigm for building SPARQL queries: the SPARQL/CQELS Visual Editor designed for the Super Stream Collider framework [23]. In that tool, blocks strictly follow the language structure and syntax and the UI requires at least basic knowledge of SPARQL. Conversely, the SparqlBlocks UI is intended to to provide blocks that should be mostly self describing and usable also without previous knowledge of the SPARQL syntax.

[5]http://www.ldodds.com/projects/twinkle/

Finally, for most of the existing tools, the visualization of the result set is passive and presented in an independent container (e.g., in many Web-based interfaces, the result page replaces the query page). In our system, a query and its results share the same workspace, supporting interactive database exploration.

3.2. Block Programming Environments

The MIT Media Lab pionereed research in educational tools for teaching the use of technology and especially programming skills. This research provided the foundation for the development of StarLogo [24], then Scratch [7] — the first broadly successful visual programming software for kids — and their successor for mobile apps, MIT App Inventor [8], which was originally developed at Google and is now based at MIT. The block programming paradigm has since been applied to several scenarios.

Apart from the cited SPARQL/CQELS Visual Editor, other block programming environments have been created to design structured queries. Bags [25] and DBSnap [26] are two visual tools to design relational algebra expressions. The former has the typical block programming look and is similar to Scratch, while the latter has a peculiar tree appearance to suite the specific context. Results so far show the effectiveness of these approaches to teach relational algebra concepts [26]. In fact, SparqlBlocks' relational operators are modelled in a similar way. But many of our operators are unique to our case because: (1) we operate on a graph data model, instead of on a relational database; (2) we allow the user to perform queries on any online SPARQL endpoint, not just on a predefined set of relations.

It is also interesting also to see how block programming environments following the imperative paradigm have been extended to allow querying and manipulating data from databases. App Inventor has interfaces to online database services like Google Fusion Tables[6] and Firebase Realtime Database[7]. Punya[8], a fork from App Inventor to build mobile applications for crisis data [27], has blocks to send SPARQL queries and receive the results. In the Learning with Data project, Scratch has been extended with blocks that query a specific dataset [28]. The approach followed in all these cases is to offer a fairly thin layer of indirection among the environment and the service offered by the database. In these environments, blocks are used to execute the queries, but not to build them: either only basic queries can be executed (most of the environments) or complex queries are passed as text (e.g., SPARQL queries in Punya) that must be built by other means. In a way, these environments stop where SparqlBlocks kicks in, at the level of the logic of the query. They are thus not comparable with SparqlBlocks, being instead potentially complementary to it.

[6]https://fusiontables.google.com/
[7]https://firebase.google.com/docs/database/
[8]http://air.csail.mit.edu/punya/

4. Features of the Programming Environment

We describe here the SparqlBlocks UI, along with motivations for the main design choices. We first present the requirements we identified and the main strategies chosen to approach them. We then proceed to describe the solution, from its top-level features to the details of the visual language, describing the different types of blocks provided.

4.1. Requirements

Based on an analysis of the existing tools, and the shortcomings discussed above, we identified the following basic requirements for the design of the SparqlBlocks UI:

1. users need not know SPARQL syntax — hence visual clues and constraints should prevent syntax errors;
2. users' need for inputting text should be minimized;
3. users should have direct access to commonly used structures;
4. users should be able to use SparqlBlocks as a step to learn the SPARQL (textual) syntax;
5. users should be able to work even without prior knowledge of the dataset — hence exploratory queries should be explicitly supported.

4.2. Main Strategies

In order to meet the previous requirements, we opted for the following general strategies.

- **Explicit language affordances.** Users should be able to visually explore the set of available language elements and how they can be combined.
- **Prevention of syntax errors.** All admissible sentences in the visual language should correspond to actual queries.
- **Multiple ways of achieving a solution.** It should be possible to build a query in many different ways, as in the underlying SPARQL language, to cope with different approaches.
- **Block Programming.** The user interface should use the block programming metaphor, with language elements represented by visual, composable blocks.
- **Results view integrated with query view.** User interfaces for building queries — both textual and visual ones — typically consist of two separate areas, one for the query and one for the results. They may be shown simultaneously, one along the other, or at different times, one after the other. To facilitate the exploratory design of queries, we should instead experiment with mixing results and views in the same workspace. It should be possible to design more than one query at once and to show the corresponding results beside each query. It should be possible to use the visual elements corresponding to the results of a query to create new queries or to modify the current one.

4.3. General Block Programming Features

Based on these general strategies, we designed a concrete solution, considering also the constraints imposed by the technologies and the required effort. Figure 2 shows a screenshot of the SparqlBlocks UI. We start our presentation of the SparqlBlocks language and environment by listing the top-level features of the solution.

The following features are the ones that are common to most block programming environments.

Language elements as blocks. Language elements, represented as keywords and grammar structures in textual languages, are represented as *blocks* (see labels *a* and *b* in Figure 2). Different language elements are represented by blocks with different visual properties (shape, color, textual labels, icons, etc.). Language items that require flexibility in their definition (like literal values and variable names) are represented in the blocks through *fields* (*c*) that provide an input of some kind (free text, dropdown).

Program structure through block composition. Blocks connect to each other via jigsaw-like *connectors* (*d*) The syntax of the language is then defined by the available blocks and how they may be connected. Possible connections are hinted to the user by the visual properties of the connector (male/female, shape) and implemented by visually dragging and attaching one block to another.

Workspace as canvas. The programmer's *workspace* is shown as an unlimited canvas in which to organize and connect the blocks (*e*). The connections define the program, thus having a directly functional meaning, while the placement of blocks in the canvas space has no effect on the program: it serves the purpose of code organization and potentially of communication in case of a collaborative setting. The canvas has some of the usual controls of the window metaphor, scrollbars and zoom, along with a control of the desktop metaphor, the trash bin to delete blocks.

Visual toolbox as inventory of components. The workspace usually starts empty and blocks are dragged from the area for the *toolbox* component (*f*), where the available blocks are organized into *categories* (*g*).

Shadow blocks as defaults and examples. This is a new feature introduced in Blockly to represent defaults and usage examples. Shadow blocks (*i*) can be blocks of any of the defined block types, with a specific behaviour: they are shown in a lighter shade of the same colour and they disappear if another block is connected in their place. They are used inside other blocks to offer sensible defaults or example values, while remaining less intrusive than regular blocks used for the same purpose: regular blocks would remain on the workspace after replaced and may hence need to be deleted/moved with a further user action.

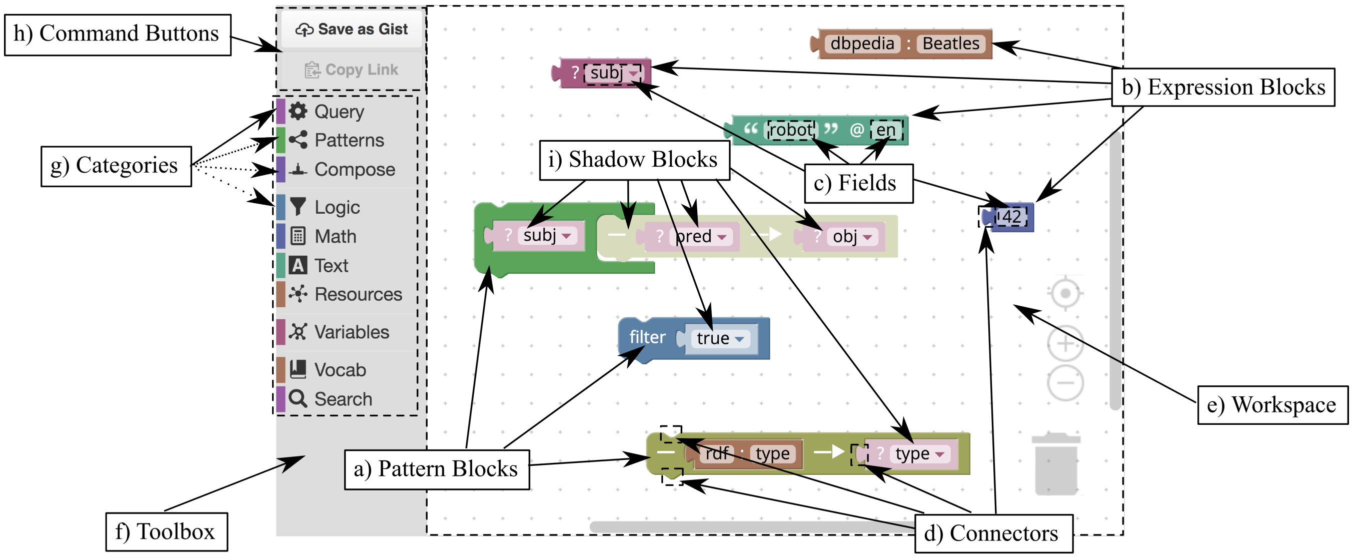


Figure 2. The user interface of SparqlBlocks.

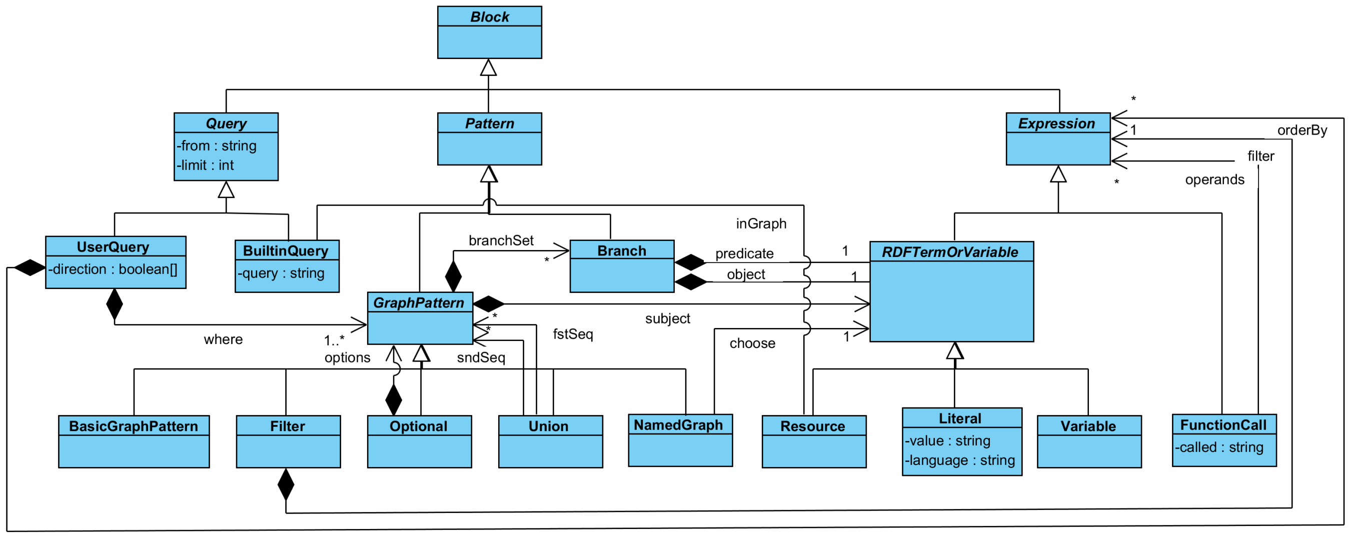


Figure 3. A class diagram synthesising information on the block types in SparqlBlocks.

4.4. Specific Features of SparqBlocks

The following features are the main ones specific to the programming environment of SparqlBlocks.

Query blocks, pattern blocks, and expression blocks. Multiple types of blocks are connected to build queries. Figure 3 shows the taxonomy of block types used in SparqlBlocks, in the form of a class diagram, also presenting information on fields and relations between block types. All the classes that are further specialised are represented as abstracts. Each block of type *Query* represents the execution of a query against a specific SPARQL endpoint (see Section 4.5). It also maintains two pieces of information:

- the *from* text field, in which the URL of the remote SPARQL endpoint has to be specified[9];
- the *limit* numeric text field, which sets the maximum number of result rows that are returned.

A specific query is defined by a structure of *Pattern* blocks that the user composes inside the query block. Each pattern block represents in turn either part of a basic graph pattern on the dataset or some operator to be applied on other patterns (see Section 4.7). Finally, *Expression* blocks are used for language structures representing scalar values, i.e. single RDF terms, variables, and calls to SPARQL functions and operators (see Section 4.6). Each of these three main types of blocks has different external connections, so that users can immediately distinguish the role of these categories: query blocks have

[9]In the examples in this paper the value of the field is always the DBPedia endpoint [15].

no external connections, pattern blocks have top and bottom connections, expression blocks have one left connection. These visual cues are important, but not sufficient, as specific subtypes have different roles in the language. So, when a block of a specific type is expected on a connection end, the systems checks if the block being connected is compatible. If the check fails, the block is "repelled" by the connection, thus signalling to the user that the connection is not permitted.

Results of execution as blocks. The results of a query are shown as blocks that can be used in the workspace, either to modify the query itself, or to build new queries (see Figure 4d). This feature supports an exploratory approach to the design of the queries.

Queries and results integrated in the workspace. To support the reuse of query results as blocks, results are part of the workspace itself, attached to the corresponding query. This enables to have multiple queries and their results directly visible and actionable together in the workspace, avoiding the need to switch between different query outputs.

Live query execution. Queries in the workspace are immediately executed when created and every time they are changed. This means the user has immediate "real time" feedback[10] and may change the query accordingly.

Export of queries and results. Users can export queries to see them represented in the SPARQL syntax. Proficient users may use this feature to sketch a new query visually and then continue working on it in SPARQL, while others may profit from this feature to learn SPARQL syntax. Result sets may also be exported for reuse.

Built-in queries. The blocks defined so far offer a bottom-up approach in which queries are built from components that represent basic functional elements. Along with those basic components, the tool provides a set of higher level blocks to perform specific queries that are often needed. These are the queries used to look for classes and properties in the used vocabulary and for specific resources in the data (see Section 4.8).

4.5. User Query Blocks

The language mimics the structure of SPARQL. As in SPARQL, the simplest query is defined through a basic graph pattern, whose simplest form is in turn a triple pattern. The blocks in Figure 4a form a triple pattern that matches any triple having `dbr:The_Beatles` as subject and `dbo:formerBandMember` as predicate. For each such triple, the variable `member` is bound to the corresponding object (see Section 4.7 for details on pattern blocks).

[10]The time to get the results from a remote SPARQL endpoint obviously depends on bandwidth and server response times for the specific query. Nevertheless, as the queries are executed asynchronously, possible delays do not affect the responsiveness of the user interface.

In order to execute a query defined according to this pattern, the *user query block*[11] (see Figure 4b) is defined. This block, representing a whole query on its own, has no external connections, as it is not meant to be stacked or put inside other blocks. Instead, it has the following fields and connections, besides *limit* and *from*, inherited from the abstract class *Query*:

- the *where* connection, hosting a sequence of graph pattern blocks (see Section 4.7) describing the query;
- a variable number of *orderBy* connections, in each of which an expression block (see Section 4.6), typically a variable, represents an ordering criterion;
- a drop down *direction* field for each *orderBy* connection, to select the direction (ascending/descending) of that specific criterion;

The number of *orderBy* connections is variable because a desired ordering may be obtained by a sequence of criteria (e.g., from a dataset of some people and their telephone numbers, to generate a telephone directory we may order first by surname and then by name when the surname is the same). To avoid to overly complicating the ordering mechanism, this variability is managed in a simple way: even if the ordering is not required, an empty connection is already available when the query is dragged to the workspace; as soon as a block is connected to this connection, a new empty connection is created to the right and so on, so that an empty connection is always available. This is the only case in which we allow an empty connection[12].

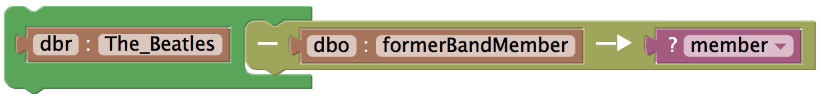
As soon as a pattern is attached to a query block, the query is run on the remote dataset. Figure 4c shows the query block connected to the pattern and "waiting for the results". When results are ready (as in Figure 4d), they appear in tabular format attached to the lower connector of the query block. Each row of the results represents a different matching of the pattern, with the corresponding variable bindings. The single data items are represented as expression blocks (specifically, resource and literal blocks) that can be dragged from the result set to create other queries (or even to modify the one that generated them).

If the content of a query block is modified, the new query is executed immediately and results shown as soon as they are available[13]. Query execution is in any case non-blocking, i.e. the UI is reactive and operational even if one or more queries are being executed.

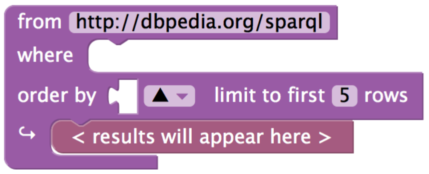
[11]The complete name "use query block" is here used to distinguish this block from the built-in query blocks described in Section 4.8. In the rest of the paper, this block is also called simply "query block" when the meaning is not ambiguous.

[12]This choice has admittedly the potential to confound the user in believing that a block is strictly required in that empty connection. Nevertheless, the other alternatives that have been evaluated (like using the Blockly mutator mechanism, that requires opening a sort of mini-workspace used to configure a block through blocks representing parts of it) pose other forms of cognitive overhead. In the evaluation described later in this paper, this choice did not cause confusion in the users.

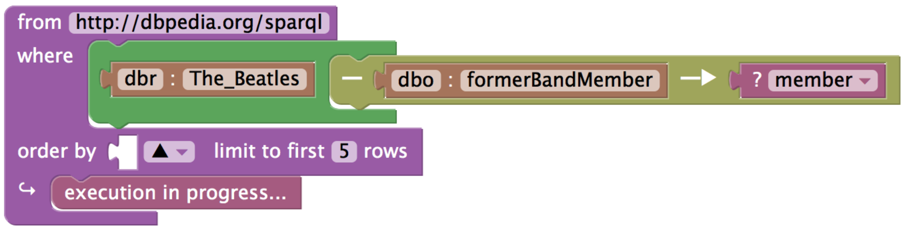
[13]When the query is modified, the table with the previous result set is also immediately wiped (replaced with the "execution in progress..." block in Figure 4c) to avoid any confusion.



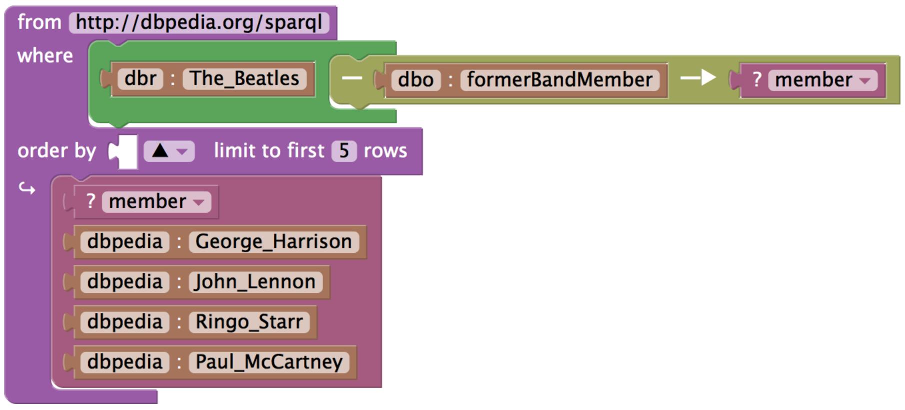
(a) Blocks for pattern



(b) Block for query



(c) Blocks joined and query in execution



(d) Query executed and results shown

Figure 4. Execution of a query in SparqlBlocks.

4.6. Expression Blocks

Expression blocks (see *b* in Figure 2) are visually identified by having horizontal output (male) connectors and represent the elements of the SPARQL language that correspond to scalar values. The expression blocks are further classified in *resource blocks*, *literal blocks*, *variable blocks*, and *function call blocks*.

Resource blocks may be specified either with a full IRI or in prefix notation (e.g. `dbpedia:Beatles` in Figure 2). Sparql-Blocks offers ways to look for resources in a dataset and to cast them directly as usable blocks (see Sections 4.5,4.8, and categories *Resources* and *Vocab* in Section 4.9). As a further option, users can look for the IRI through external means (e.g., a Linked Data browser, like the one used as front-end for the resources by DBpedia) and then copy/paste it on a blank resource block; if a known namespace is recognized, the IRI is converted on the fly to the prefixed version.

Literal blocks hold strings with an optional language tag (e.g. `"robot"@en` in Figure 2), booleans, or numbers (e.g. `42` in Figure 2).

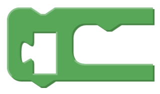
Variable blocks are used as wildcards in patterns. They are represented by specific blocks with a drop-down menu, that can be used to create a new variable or use one of the existing ones (`subj` in Figure 2).

Function call blocks represent calls to SPARQL functions and operators (`lengthOf`, `absolute`, `<`, `+`, etc.). Each function call block has a number of internal connections equal to the number of parameters (operands) that the corresponding function (operator) has. Each internal connection accepts an expression block.

Even if the output connectors for all the value blocks have the same appearance, they cannot all be connected in the same ways, because of the different roles in the language. RDF terms and variables can be used in basic graph patterns (with some limitations, see Section 4.7), while functions/operators cannot. The latter are permitted whenever, in SPARQL, expressions can be used. Currently, in SparqlBlocks, this corresponds to the case of filters (see Section 4.7) and ordering.

4.7. Pattern Blocks

Pattern blocks (see *a* in Figure 2) are visually identified by having vertical connectors (female on top and male on bottom) that allow them to be stacked one under the other and used in specific contexts, like the *where* connector of the query block. As for the value blocks, even if the output connectors of all the pattern blocks have the same appearance, they cannot be connected all in the same way. In the case of pattern blocks there are two strictly separated types: the *pattern blocks*, and the *branch blocks*.


(a) Graph Pattern

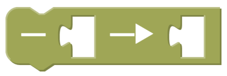

(b) Branch

Figure 5. Blocks for basic graph patterns.

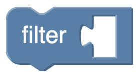
Basic graph patterns are built using the blocks shown in Figure 5, starting from the **graph pattern block** (Figure 5a), used to group triple patterns that have a resource or variable in common. It has two internal connectors:

- a *subject* connector that accepts a resource block or variable block;
- a *branch set* connector for adding triple patterns related to the common resource/variable, accepting a sequence of branch blocks.

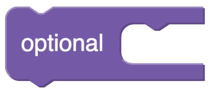
A **branch block** adds a triple pattern for which the common resource/variable of the basic graph pattern block is the subject. It has two internal connectors:

- a *predicate* connector that accepts a resource block or a variable block;
- an *object* connector that accepts a resource block, a literal block, or a variable block.

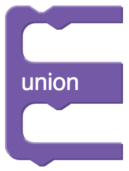
Hence, following SPARQL syntax (see Section 2.2), branch blocks can be stacked into basic graph pattern blocks to join multiple triple patterns that have a resource/variable in common. Basic graph pattern blocks can be in turn stacked one upon another to build more complex basic graph patterns.



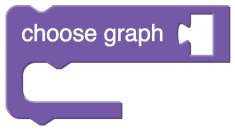
(a) Filter



(b) Optional pattern



(c) Union of two patterns



(d) Named graph pattern

Figure 6. Pattern blocks beyond basic graph patterns.

Apart from the basic graph patterns, other pattern blocks can be used to build more complex queries:

- the **filter block** (Figure 6a), filters the matchings of the graph pattern sequence[14] according to a given condition (which must evaluate to true to pass the filter); the *filter* connector accepts an expression block that will be evaluated as boolean;
- the **optional block** (Figure 6b) adds a sequence of graph pattern blocks as optional in the sense that its matching is not required but if matched the variables will be bound accordingly;
- the **union block** (Figure 6c) adds two sequence of pattern blocks as alternatives in the sense that the union of the matching of the first and the second pattern will be considered;
- the **named graph block** (Figure 6d) selects a specific named graph of the source RDF dataset (via the *choose* connector, accepting a resource block or a variable one), for the sequence of contained graph patterns.

4.8. Built-in Query Blocks

There are some queries that are useful for every RDF dataset, regardless of its specific domain, for example queries used to explore the vocabulary used in a dataset, i.e. the IRIs used for classes, properties, and other important resources. Hence, we present the user with a library of **built-in query blocks** that offer pre-built queries for common tasks. The *BuiltinQuery* type of blocks inherits from *Query* the *from* and *limit* fields, to specify the SPARQL endpoint and the maximum number of result rows, respectively. These blocks also have an *inGraph* connector that optionally accepts a resource block to select a named graph from the dataset (if not used, the default graph is selected). As with user query blocks, the built-in query blocks are executed as soon as the required fields and connections are filled and are executed again each time some of the fields or connections are changed. The results, also in this case, are shown as a table connected to the lower part of the query.

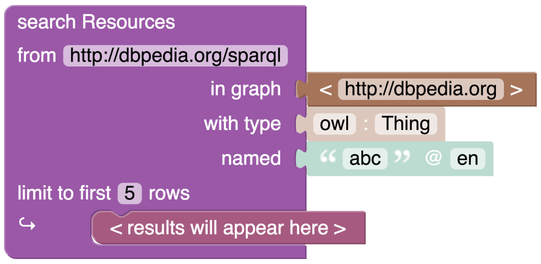
Currently, the library comprises the following blocks[15]:

Search resources block, having a *withType* connector, which accepts a resource block and a *named* connector, accepting a literal block with text (Figure 7a). It looks for resources of the specified type and containing the specified text in their label. The columns of the table of results are *?resource* and *?label*.

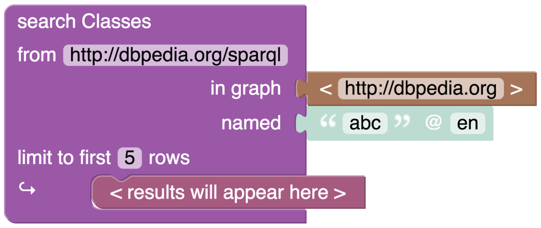
Search classes block, having a *named* connector, which accepts a literal block with text (Figure 7b). It looks for classes whose label contains the specified text. The typical use of a class in a query is to look for instances of that class. So, to give a sensible default and to reduce the potential errors, for every result there is a column with a pre-built pattern looking

[14]The filter block operates on the graph pattern sequence this block is part of, i.e. the sequence comprising blocks stacked both above and below the filter block.

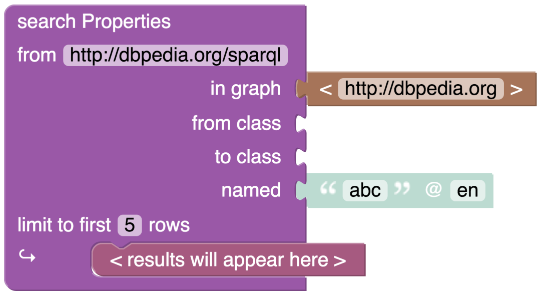
[15]To keep the diagram of Figure 3 simple, we do not detail these specialised classes.


search Resources
from http://dbpedia.org/sparql
in graph < http://dbpedia.org >
with type owl : Thing
named " abc " @ en
limit to first 5 rows
< results will appear here >

(a) Search Resources


search Classes
from http://dbpedia.org/sparql
in graph < http://dbpedia.org >
named " abc " @ en
limit to first 5 rows
< results will appear here >

(b) Search Classes


search Properties
from http://dbpedia.org/sparql
in graph < http://dbpedia.org >
from class
to class
named " abc " @ en
limit to first 5 rows
< results will appear here >

(c) Search Properties

Figure 7. Built-in query blocks.

for instances of that class. The columns of the table of results are *?class*, *?label*, and *?pattern*.

Search properties block, having the optional connectors *fromClass* and *toClass*, both accepting resource blocks, and a *named* connector that accepts a literal block with text (Figure 7c). It looks for properties such that the specified text is contained in their label, optionally going from a given class to another given class. The typical use of a property in a query is to look for pairs of resources connected by that property. So, in analogy with the search classes block, there is a column with a pre-built branch that uses the property. The columns of the table of results are *?property*, *?label*, *?domain*, *?range*, and *?branch*.

4.9. Organization in Categories

Blocks are visualized and organized (in the toolbox) according to their role. The *query* block is provided under the category *Query*. Blocks used to build graph patterns are grouped under the category *Patterns*. The *optional*, *union*, and *named graph* blocks are under the category *Compose*. The categories *Logic* (also containig the *filter* block), *Math*, and *Text* contain *literal*, *operators* and *functions* blocks, organized according to their types. Resource and variable blocks are available in the respective categories *Resources* and *Variables*. The category *Vocab* contains some pattern and branch blocks for standard vocabularies, organized by categories named after each of the vocabularies. Finally, all the built-in query blocks are grouped under the category *Search*, as they all deal with retrieving resources.

5. Implementation

Blockly [16] is a JavaScript library for block programming maintained by Google, on which several tools — including MIT App Inventor 2 — are based. It provides a set of basic blocks covering the structure of typical imperative programs. Most importantly, it is also extensible programmatically to define new blocks. SparqlBlocks is based on an extension of the Blockly JavaScript library, working entirely on the client side. We extended the library to supply the specific blocks needed for SPARQL queries and execution. We also added the necessary code to generate SPARQL fragments from the blocks. The SPARQL execution block listens for changes in its query connection; each time the query changes, the corresponding SPARQL query is generated and sent to a SPARQL endpoint. The SPARQL endpoint to be used is set as a field of the execution block. The results are used to dynamically generate the result block and its sub-blocks. The standard prefix definitions from *prefix.cc*[16] are used to add prefix declarations in the query sent to the endpoint and to convert the IRIs in the result to the prefixed notation.

SparqlBlocks is available online[17] and can be used directly to query any public SPARQL endpoint. Furthermore, the code is free and publicly available[18].

6. Design Evaluation

Before organizing an evaluation with the users, SparqlBlocks was formally evaluated "in house" using the relevant heuristics. We will first present its evaluation in terms of cognitive dimensions and then analyze its affordances in the specific context of query design.

6.1. Cognitive Dimensions

Cognitive dimensions [29] is a framework developed by Green for the analysis of the properties of programming languages. Green and Petre later derived from that general framework a more specific one targeting visual programming languages [30]. An initial evaluation of SparqlBlocks was performed based on the cognitive dimensions deemed as relevant to our context.

Consistency. The block syntax favours internal consistency; external consistency is satisfied with respect to SPARQL textual syntax because the structure is maintained and, partially, with respect to other Blockly-based languages because the appearance and behaviour of basic expressions is preserved.

[16]http://prefix.cc/
[17]http://sparqlblocks.org/
[18]https://github.com/miguel76/SparqlBlocks

Diffuseness. Representation through blocks requires more space than textual representation. However, SparqlBlocks is designed to be efficient in terms of graphic entities, with the visual appearance of each block tuned to minimize the space required for it.

Error-proneness. The chance of making syntax errors is extremely reduced, compared to textual syntax: the affordances are first coarsely constrained by the differences between shapes, then by other visual cues, and finally by the behaviour of visual elements. The use of results for refining existing queries or creating new ones reduces the chance of mismatches between queries and data structure.

Hidden dependencies. Dependency and scope of variables are strictly related to the SPARQL query structure that is closely followed by the block structure.

Premature commitment. Queries can be as easily modified, decomposed, and recomposed at any time, the user is thus not forced in any way to stick to earlier decisions.

Progressive evaluation. A trial and error approach is favoured by the automatic execution of queries in query blocks. While the user builds a query, she can immediately see its results. Furthermore, the user can try, as separate queries, any graph pattern that will eventually become part of the complete query. This method is favoured by the possibility of having multiple queries in the workspace.

6.2. Query Design Affordances

To further argue in favor of the proposed paradigm, we show some of the affordances of SparqlBlocks related to the query construction process. We analyse the basic operations that a developer — whether an expert or a beginner — needs to perform. The underlying idea is that designing a query is an interactive process in which the query emerges from a sequence of basic operational steps, possibly retracting or modifying the effect of previous ones, without having to be completely detailed in advance.

Generalization/Specialization. The design of a query may proceed from a generic version, with a minimal number of constraints, designed to start getting some data. Then, by adding constraints, the query will gradually become more selective. This process, which we call *query specialization*, is supported by reducing the free variables (through replacement with constant values or with already used variables), by adding filter blocks, by adding more complex basic graph patterns to be fulfilled, and by moving patterns out from an *optional* or *union* block. The usage of blocks also for representing query results allows the user to easily identify and manipulate the specific blocks corresponding to resources or literals needed to replace variables or to create filter expressions. An example of *query specialization* is the transformation from the query in Figure 8a (which asks for some mountains and their locations) to the query in Figure 8b (where only Brazilian mountains are selected) by dragging the resource `dbpedia:Brazil` from the results in the place of the variable `area`.

The design of a query may also start from a specific query on known data and then proceed by relaxing some constraints to include a greater set of results. This process of *query generalization* is supported by replacing constant values with variables, by removing filter blocks, by removing graph patterns (or part of), and by moving graph patterns under an *optional* or *union* block. These actions correspond directly and naturally in SparqlBlocks to the removal of blocks or the creation of new ones for the variables. An example of generalization can be the reversal of the specialization example, i.e. from the query in Figure 8b to the one in Figure 8a, by dragging on the workspace a new variable block to replace the resource `dbpedia:Brazil`.

Composition/Decomposition. A complex query may be created by composing different queries together, so that, for example, the output of separate components of the query can be checked before composing them. As the proposed UI permits multiple queries to be built and executed in the same workspace, composition is directly executed by joining blocks from different queries[19]. For example, to build a query for European mountains, the user may first design a query to get mountains (Figure 8a) and another query to get the European countries (Figure 9a). The complete query can be composed by dragging the graph pattern of the latter query and adding it inside the former one and changing the `area` variable to `country` (Figure 9b).

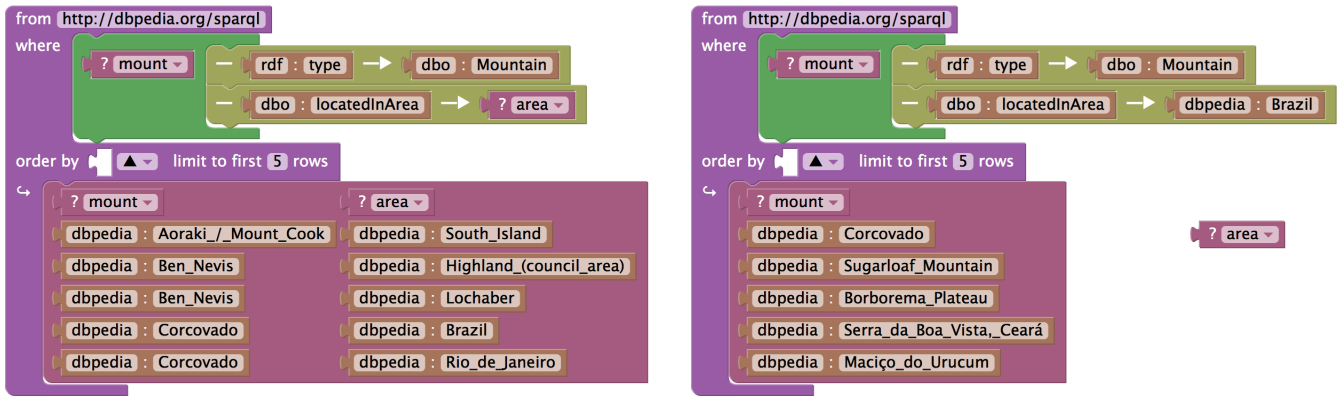
The reverse operation is to decompose a query in smaller ones. From the point of view of the UI the actions are similar to the ones needed for composition: dragging blocks and creating some new ones. Reversing the example of composition gives an example of decomposition.

Stepwise Querying. Sometimes a query design comes after some exploratory steps that identify relevant resources, classes, or properties. This approach is supported by permitting values dragged out of the result set of a query to be used by other queries. The old query may then be removed or simply kept aside for further use. For example, Figure 10 shows the use of a built-in query block to get some candidate classes to represent the set of mountains, by searching among available classes using their *label* attribute. As soon as `dbo:Mountain` is identified as the relevant class, the corresponding resource block may be dragged away and used for follow-up queries, as in Figure 8a (in the object connection of the `rdf:type` branch of the basic graph pattern).

7. Iterative Design

SparqlBlocks has been tested with users to evaluate and identify possible improvements. An informal evaluation has been carried on with expert users, whose observations have led to several improvements in the tool.

[19]Blocks may also be duplicated to preserve the original queries.



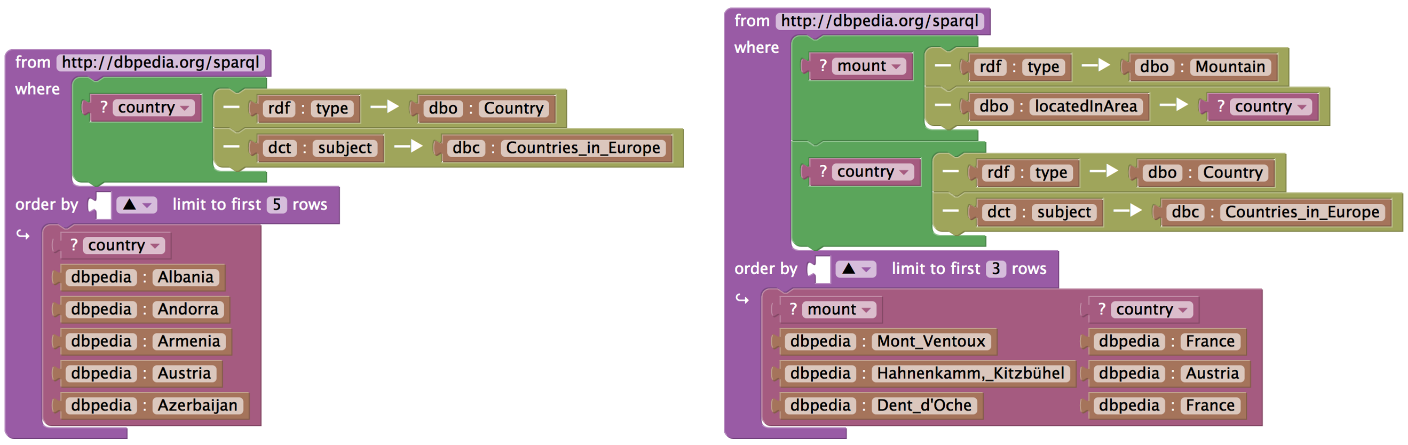
```
from http://dbpedia.org/sparql
where
  ? mount  — rdf : type → dbo : Mountain
           — dbo : locatedInArea → ? area
order by [ ▲ ]  limit to first 5 rows
↪
  ? mount                          ? area
  dbpedia : Aoraki_/_Mount_Cook    dbpedia : South_Island
  dbpedia : Ben_Nevis              dbpedia : Highland_(council_area)
  dbpedia : Ben_Nevis              dbpedia : Lochaber
  dbpedia : Corcovado              dbpedia : Brazil
  dbpedia : Corcovado              dbpedia : Rio_de_Janeiro
```

```
from http://dbpedia.org/sparql
where
  ? mount  — rdf : type → dbo : Mountain
           — dbo : locatedInArea → dbpedia : Brazil
order by [ ▲ ]  limit to first 5 rows
↪
  ? mount                                ? area
  dbpedia : Corcovado
  dbpedia : Sugarloaf_Mountain
  dbpedia : Borborema_Plateau
  dbpedia : Serra_da_Boa_Vista,_Ceará
  dbpedia : Maciço_do_Urucum
```

(a) Query to get mountains, generalization of query (b).

(b) Query to get Brazilian mountains, specialization of query (a).

Figure 8. Example of specialization/generalization.



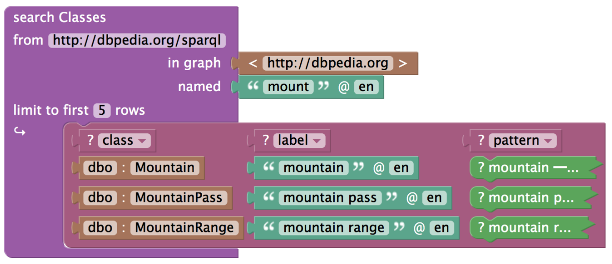
```
from http://dbpedia.org/sparql
where
  ? country  — rdf : type → dbo : Country
             — dct : subject → dbc : Countries_in_Europe
order by [ ▲ ]  limit to first 5 rows
↪
  ? country
  dbpedia : Albania
  dbpedia : Andorra
  dbpedia : Armenia
  dbpedia : Austria
  dbpedia : Azerbaijan
```

```
from http://dbpedia.org/sparql
where
  ? mount    — rdf : type → dbo : Mountain
             — dbo : locatedInArea → ? country
  ? country  — rdf : type → dbo : Country
             — dct : subject → dbc : Countries_in_Europe
order by [ ▲ ]  limit to first 3 rows
↪
  ? mount                               ? country
  dbpedia : Mont_Ventoux                dbpedia : France
  dbpedia : Hahnenkamm,_Kitzbühel       dbpedia : Austria
  dbpedia : Dent_d'Oche                 dbpedia : France
```

(a) Query to get European countries.

(b) Query for European mountains, a composition of (a) and (c).

Figure 9. Example of query composition in SparqlBlocks.



```
search Classes
from http://dbpedia.org/sparql
              in graph  < http://dbpedia.org >
                 named  “ mount ” @ en
limit to first 5 rows
↪
  ? class              ? label                        ? pattern
  dbo : Mountain       “ mountain ” @ en              ? mountain —...
  dbo : MountainPass   “ mountain pass ” @ en         ? mountain p...
  dbo : MountainRange  “ mountain range ” @ en        ? mountain r...
```

Figure 10. Getting classes for “mount”.

7.1. Feedback from Users

After the first version of the tool was finished, we presented it at the International Semantic Web Conference (ISWC) 2015 and at the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 2015. These are top conferences for, respectively, the Semantic Web and Visual Languages research communities.

Demonstrating the tool in these two venues allowed us to gather feedback from experts of two complementary aspects of the user interface: its efficacy as a query design tool for the Semantic Web and its expressivity as a visual language (and specifically as a block programming language). The tool was presented to interested users in the informal way typical of software demonstrations in conferences. About a dozen of users were invited to use the application and to share their feelings about it in a colloquial fashion.

The following are the main reactions that we collected about the system as a whole:

- the UI was seen as **appealing** by all the users;
- users had **mixed feelings about ease of use**: although they felt the system was easy to use, they were not immediately confident in using it, due to the perceived complexity of the system in terms of available blocks and possible combinations;
- most users considered the tool as **novel**, especially for the use of the blocks in the results.

Detailed findings on specific UI elements are presented in the next subsection, along with the performed changes.

7.2. Changes to the User Interface

The overall positive reactions concerning novelty and appeal of the SparqlBlocks environment elicited subsequent work to improve it, especially in the areas that were perceived by the users as lacking. The following changes were introduced with the aim to achieve a better user experience.

Join query block with execution block to reduce the number of different available blocks and types of connections. The blocks were distinct because of their different conceptual role, one representing a complete SPARQL query (containing a sequence of patterns and fields for ordering and limiting the results) *vs.* the other representing a SPARQL endpoint (containing a query, the field for the endpoint URL, and a generated output field for the results). In practice, the execution block made sense only connected to the query block (and vice versa) and the chance of moving an entire query from an endpoint to another was not exploited enough to justify the increased complexity. So the two blocks were joined together in a new query block that includes also the text field for the URL of the endpoint and the generated field for the results.

Simplify table of results to avoid confusion, reduce the space occupied, and the cognitive load. The appearance was changed from the previous complex block structure to a single *results* block containing directly the draggable blocks, still arranged in a table. While before there were a large number of blocks used without functional reason, after the change the only non-draggable block is the top one[20].

Replace pre-built queries with built-in query blocks to reduce the cognitive load for both beginners and experienced users. Initially, instead of built-in query blocks, we provided pre-built queries exposing all the details. The idea was to give the chance to not only use them but also to learn from their structure and change them according to specific needs. The pre-built queries occupied a lot of screen space and represented a heavy cognitive load for all the users; beginners were especially not confident of where the parameters for the query had to be placed. After the change, the predefined queries are represented by single blocks (the built-in query blocks) in which the only exposed connections and fields correspond to the parameters that must be set to use the queries.

Reduced use of predefined values to reduce confusion in perceived affordances and avoid the perceived extra work needed to replace values. Blockly allows the toolbox to contain not just basic blocks, but pre-connected groups of blocks. In the first version we used this feature to provide default and example values for basic blocks. That was found to be to confusing, so we reduced the use of these pre-connected sub-blocks, in several cases replacing them with shadow sub-blocks.

[20]Due to the way the layout of blocks is managed in Blockly, it is not currently possible to avoid the use of this last non-functional block without extensive implementation work.

Support for building queries from classes and properties. The typical use of a class in a query is to look for instances of that class. So, to give a sensible default and to reduce the potential errors, for every result of a black-box query searching for classes there is a column with a pre-built pattern looking for instances of that class. Similarly, the typical use of a property in a query is to look for pairs of resources connected by that property. So, for every result of a black-box query searching for properties, there is a column with a pre-built branch that uses the property. In both cases, those added blocks may be reused directly in a query.

Directly offer highly used classes and properties to avoid unnecessary effort to discover or manually write down them. Classes like `foaf:Agent` and properties like `rdfs:seeAlso` are widely used in RDF datasets so it makes sense to have them at hand, rather then discovering them via black-box queries. The new toolbox category *Vocab* contains a set of subcategories corresponding to different vocabularies. For each vocabulary a set of common classes and properties is available, already available in the form of patterns and branches.

Keep in memory resources that have being used during the session, to recover specific resources, classes, properties that may have been found and then lost. The category *Resources* contains now both the blank resource blocks (one in prefix notation and one in full URI notation) and all the previously created resources, the most recent ones on top.

8. User Evaluation

After the updates described in Section 7, the SparqlBlocks UI was evaluated by a group of Master students, PhD students, and young researchers of the Computer Science Department of Sapienza, University of Rome. They all had a solid background in Computer Science but they were not conversant with specific Semantic Web technologies like RDF and SPARQL. This evaluation was designed as a formal assessment involving the solution of three tasks and a questionnaire.

8.1. Setup

Users went through an online questionnaire with questions and activities organized in a sequential manner:

1. rate their own background expertise on Semantic Web concepts (Semantic Web, Linked Data) and technologies (RDF, SPARQL), databases (relational databases, graph databases, SQL), and block programming environments (by indicating known environments);
2. rate the expected complexity of each of the three tasks beforehand, without having seen the tool yet, and figuring out how difficult it would be to look for the reply with access to the Web;
3. follow an interactive tutorial designed to teach basic use of the SparqlBlocks environment;
4. execute three tasks with the tool;

5. rate the actual complexity of executing each task;
6. rate the confidence in the result obtained for each task;
7. rate the tool for perceived easiness of use, appeal, and novelty;
8. write open-ended comments on SparqlBlocks.

All the ratings were on a scale from 0 to 6. For the block programming environments, participants were asked to select the environments that they knew about, among a list including the most common ones (Scratch, MIT App Inventor, etc.) and the option *other* to include other environments.

11 people participated in the evaluation. Even if the whole evaluation could have been executed online, in most of the cases there was a facilitator so that further qualitative data and feedback could be gathered. Only 3 participants executed the tasks on their own. When present, the facilitator assumed also the role of giving real-time help on the elements of the tool, as currently there is not a complete help system embedded within the tool. In any case, no explicit suggestions were given on how to solve the tasks.

There were no time limits, but participants were informed that the estimated duration of the test (including questionnaire, tutorial, and tasks) was 50 minutes. They were told to try to solve as many tasks as they could. They had thus the implicit option to stop before achieving the solution of all the tasks and record what they completed.

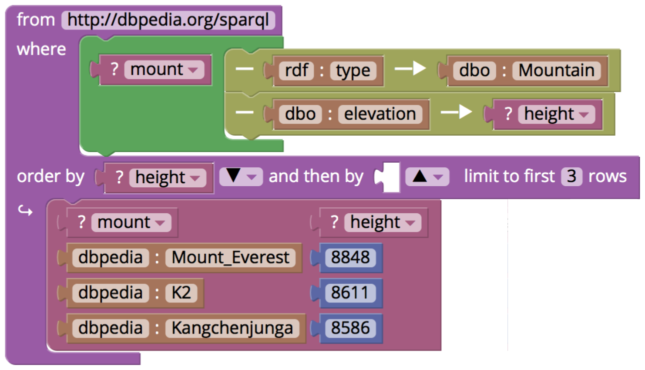
8.2. Tasks

The tasks were chosen so as to require the design of structured queries of increasing complexity. Queries were run on the DBpedia SPARQL endpoint. For each task, users were asked to find the resource corresponding to the result by building appropriate queries with the system:

1. third highest mountain in the world;
2. lowest mountain above 8,000 m in the world;
3. third highest mountain between China and Nepal.

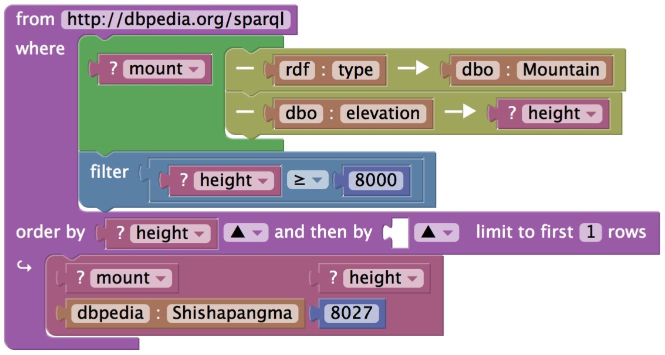
These three main tasks were intertwined with some helper tasks that required the user to find the elements that were useful to design the main queries: the class used for mountains and the one for countries, the property used for the elevation of mountains and the one used for their location, and the resources used for China and Nepal. These helper tasks required the appropriate use of black-box query blocks that search for classes, properties, and resources. These classes, properties, and resources could have been given directly to the participants as basic blocks with which to build the queries, but we preferred to test the more realistic situation in which the user has no prior logic of the vocabulary used in the dataset.

Task 1 required building a query with a graph pattern to get all the mountains and their elevations, then to set the ordering to decreasing in respect to the elevation (see Figure 11). Task 2 required adding a filter to that query, so that only mountains with elevation more than 8,000 m. were selected, and to invert the ordering to increasing (see Figure 12). Task 3



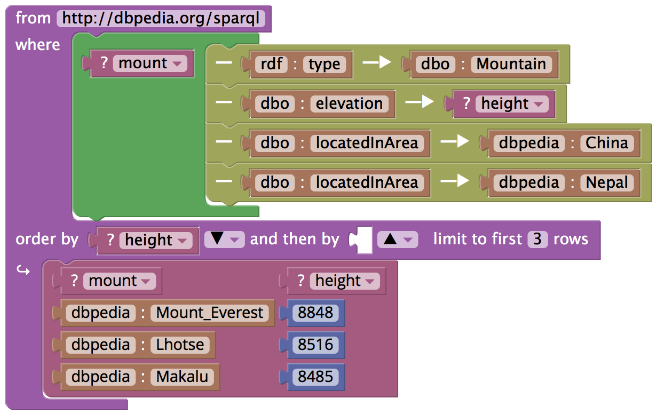
```
from http://dbpedia.org/sparql
where
  ? mount — rdf : type → dbo : Mountain
          — dbo : elevation → ? height
order by ? height ▼ and then by ▲ limit to first 3 rows
↪
? mount                      ? height
dbpedia : Mount_Everest      8848
dbpedia : K2                 8611
dbpedia : Kangchenjunga      8586
```

Figure 11. Query to solve Task 1.



```
from http://dbpedia.org/sparql
where
  ? mount — rdf : type → dbo : Mountain
          — dbo : elevation → ? height
  filter ? height ≥ 8000
order by ? height ▲ and then by ▲ limit to first 1 rows
↪
? mount                      ? height
dbpedia : Shishapangma       8027
```

Figure 12. Query to solve Task 2.



```
from http://dbpedia.org/sparql
where
  ? mount — rdf : type → dbo : Mountain
          — dbo : elevation → ? height
          — dbo : locatedInArea → dbpedia : China
          — dbo : locatedInArea → dbpedia : Nepal
order by ? height ▼ and then by ▲ limit to first 3 rows
↪
? mount                      ? height
dbpedia : Mount_Everest      8848
dbpedia : Lhotse             8516
dbpedia : Makalu             8485
```

Figure 13. Query to solve Task 3.

required extending the graph pattern such that the mountains selected are required to be located both in China and Nepal (see Figure 13). Queries similar to the ones required in Tasks 1 and 2 were shown in the tutorial (using a text variable instead of a numeric one), while no query in the tutorial had a characteristic required in Task 3: two branches with the same predicate but different objects. Listings 2 (earlier on page 3), 3, and 4 show the SPARQL queries to solve the three tasks. They are the textual counterpart of the block configurations shown in Figures 11, 12, and 13.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height .
  FILTER(?height >= 8000) .
}
ORDER BY (?height)
LIMIT 1
```

Listing 3. SPARQL corresponding to blocks in Figure 12.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbpedia: <http://dbpedia.org/resource/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height ;
    dbo:locatedInArea dbpedia:China ;
    dbo:locatedInArea dbpedia:Nepal .
}
ORDER BY DESC(?height)
LIMIT 3
```

Listing 4. SPARQL corresponding to blocks in Figure 13.

9. Quantitative Results

All the participants filled out the questionnaire, followed the tutorial, and were able to solve at least the first task. The average total completion time was slightly higher than the estimated 50 minutes, around 1 hour. 9 participants out of 11 were able to solve all three tasks, while the other two solved only the first task.

We proceed to show in more detail the quantitative data gathered through the questionnaire and measuring completion times for the different parts of the test. The group of participants is small and thus these data do not have statistical significance. Nevertheless, the following quantitative analysis, along with the qualitative analysis presented in Section 10, is useful to discuss the effectiveness and challenges of Sparql-Blocks and to gain insights on aspects to consider for such tools.

9.1. Questionnaire

The plots in figures 14 through 16 are histograms that show the distribution of the ratings given by the participants. The horizontal axis is rating on a scale of 0 to 6, while the vertical axis is the number of participants who gave that rating[21]. Figure 14 shows the distributions for participants' self-assessed relevant background knowledge, while Figure 15 shows, for each task, the distributions for *expected* (assessed before using the tool) and *actual* complexity (using the tool) and confidence of the results obtained (with the tool). Figure 16 shows the distributions for the rating given by the participants to the various aspects of the proposed user interface: ease of use, use

[21]The total height of all the bars in each plot is thus 11 (for the 11 users).

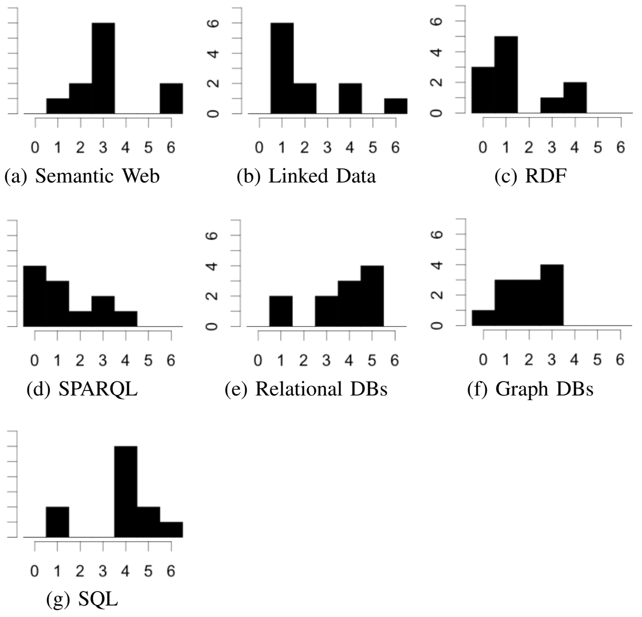


Figure 14. Distributions of knowledge of technologies

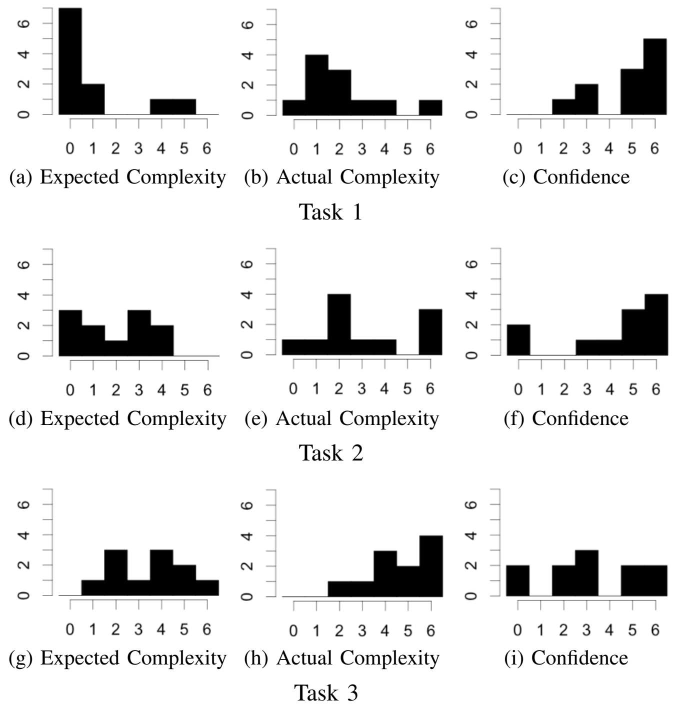


Figure 15. Distributions of Complexity by Task

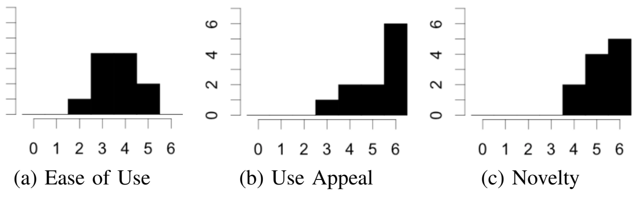


Figure 16. Distributions of Ratings for Tool Aspect

appeal, and perceived novelty. Only two participants reported to have used at least one block programming environment: in both cases they had used Scratch, in one also Blockly Games.

9.2. Completion Times

Due to a bug in the logging of the tool we were not able to record the detailed timing for the test, so for most participants we have just the whole time of completion (including questionnaire and tutorial) that ranged between 46 and 76 minutes, with an average of approximately 62 minutes. For 5 of the participants, anyway, we have the breakdown of times among questionnaire, test, and tutorial, which should give a quite accurate estimate of what happened in the other cases. The time to reply to the questionnaire was consistently around 4-5 minutes; the tutorial took between 19 and 33 minutes to be completed (with an average of approximately 25 minutes); the three tasks were solved between 23 and 48 minutes (with an average of approximately 35 minutes). Among the tasks, the first two required roughly the same amount of time (from 5 to 10 minutes each) while the third required roughly triple effort (from 15 to 30 minutes).

9.3. Discussion

Regarding background knowledge and recalling that all the participants are enrolled in the department of computer science as at least Master students, it is not surprising that relational databases (Figure 14e) and SQL (Figure 14g) are quite well known subjects. Most of the participants declared to have a midway expertise relating the Semantic Web (Figure 14a), but did not know about specific Semantic Web technologies like RDF (Figure 14c) and SPARQL (Figure 14d), with around two thirds of the participants selecting 0 or 1. Also the Linked Data expertise (Figure 14b) is fairly low (again, around two thirds selecting 1), even if it is a term that shares a good part of its meaning and technologies with the Semantic Web concept. Maybe this could be explained by the fact that the term Linked Data is less used than Semantic Web in an academic context. Finally, it is also interesting to see that a technology like graph databases (Figure 14f), which seems to be quite trending among developers, is not well known in an academic group of computer scientists.

We assume that the knowledge of relational databases and SQL has helped participants in understanding the aspects that SPARQL (and our closely related visual language) has in common with relational algebra. At the same time, the basic data model (RDF) is quite different from the relational model and in part closer to the model of graph databases. So the fact that RDF, SPARQL, and graph databases were poorly known implies a potential challenge in understanding the graph data model and the basics of SPARQL, which is built on graph pattern matchings. The qualitative analysis described in Section 10 confirms this issue.

Regarding the predicted complexity of tasks (Figures 15a, 15d, 15g), participants describe a progressively growing complexity from Task 1 to Task 3, in agreement with the increased complexity of the query required to solve tasks. The complexity perceived after solving the tasks (Figures 15b, 15e, 15h) is distributed instead more like the actual solution times: Task 2 is considered only slightly more difficult than Task 1, while Task 3 is perceived as much more complex. The perceived confidence with the found result (Figures 15c, 15f, 15i) follows a similar trend: participants are quite confident in the solutions found for Tasks 1 and 2, while confidence in the solution for Task 3 is mixed. Some hints for the higher complexity found for Task 3 were already given in Section 8.2: the novelty of an aspect of the query with respect to the queries presented in the tutorial. This difference is further discussed in the qualitative analysis described in Section 10.

The comparison between the complexity expected for a task and the actual complexity of solving it with SparqlBlocks leads to a quite unappealing conclusion: the complexity met by participants using the tool is typically higher than the expected one. This is, in afterthought, a reasonable assessment of complexity by the users: even if the tasks cannot be solved by, say, a single Google query (and Tasks 2 and 3 cannot), a user can quite easily find the results in a relatively short time (compared to measured completion times) by exploring the Web. Tasks 2 and 3 may probably be solved quicker by an expert user of SparqlBlocks (Task 1 is such an easily found answer that it would be hard to beat Google time), but for a new user it is not the case. Needing to learn the vocabulary is also a partial hindrance to the quick design of queries.

Perhaps in contrast with the complexity encountered when solving the tasks, the analysis of the global ratings given by the users to the tool is quite encouraging. The ease of use of the tool is considered from average to good, perhaps recognizing that the usage is not trivial, but after a bit of learning SparqlBlocks permits building complex queries without having to learn a specialized language like SPARQL. Better still, the participants rated the tool as being highly appealing and novel. While the assessment of novelty by users not familiar with Semantic Web standards and tools may and should be downplayed, the fact that the tool was appealing to non experts of the field is for us a very promising result. We also recognize that a probable contribution to this result is the appeal given by the block programming paradigm to people that had not experimented with it before (9 participants out of 11).

10. Qualitative Analysis

A facilitator attended most of the user tests (8 out of 11), taking notes on relevant user behaviour, receiving explicit suggestions and feedback, and following the chain of action-related reasoning of the users. Furthermore, the online questionnaire contained an open-ended field for comments that was used to describe the experience by the three participants that worked on their own. The qualitative data gathered by these means is summarized in this section, focusing on perceived issues, strengths, and suggestions.

10.1. Issues Found

The main issues recognized are described in the following paragraphs.

Difficulties in understanding how blocks may be connected. These appear to be due to two main reasons:

- Failure to understand the behaviour of the block-based UI, especially for the cases of stacking pattern/branches (the availability of the top and bottom connections to stack similar blocks was not evident for many participants) and of replacing shadow blocks (the connector was often not recognized as available when a shadow block was in place);
- Failure to understand the basic structure of patterns-/branches, or the role of variables versus resources (many participants were trying to find analogies with relational algebra, which were often misleading with respect to graph pattern matching and the way variables are bound in SPARQL). A recurring error was using properties in expressions or order by clause, where the intended behaviour would have been realized by putting in the same place the object variable connected to the subject by that property (see for example the query in Figure 17, where the intended query would have been the one in Figure 12).

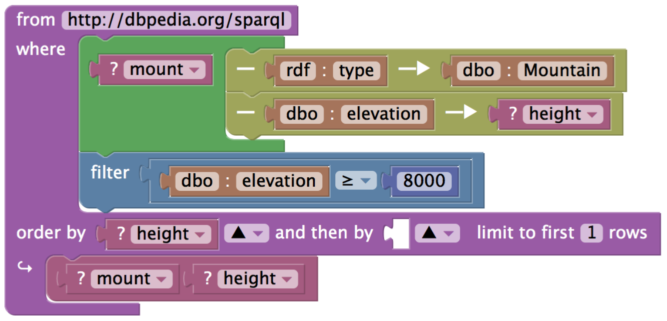


Figure 17. Syntactically correct query that has two constants compared in the filter clause, leading to an empty result set as the filter expression is always false.

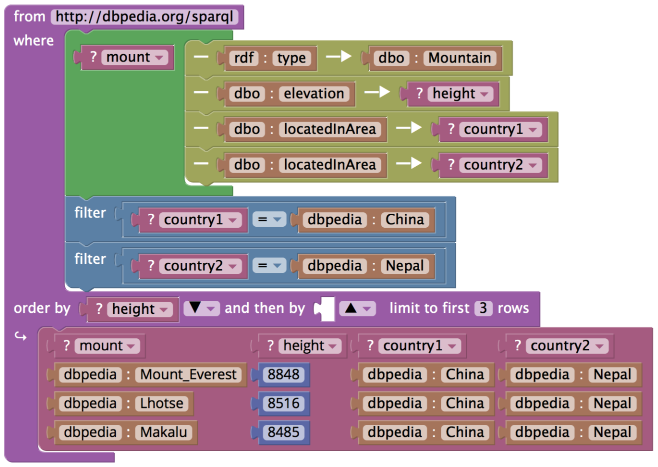
Query execution and table results not self-evident. At the beginning of the tutorial, the fact that the query is automatically executed and that the table of results is attached to the same block is not apparent. After some steps of the tutorial, this fact was understood by everyone because it is central to every action that is performed, but missing this fact initially may confound the user and eventually slow down progress in the tutorial.

Operators hidden in drop-down menus were hard to find. Some operator/function blocks contain a drop-down menu for selecting one such item; for example a block is used for all comparison operators ($<, \leq, =, \neq, \geq, >$) and one for logic operators (*and*,*or*). But in the corresponding category of the toolbox, each block is shown with the default operator/function selected, thus hiding the availability of other operators/functions. That confused many participants, until the logic of the groupings was understood (after which it was not a problem anymore).

Difficulties in using variable blocks. The combination of the nontrivial way in which variables are used in SPARQL (different from both how variables are used in typical programming languages and field names are used in SQL) and some idiosyncrasies in the behaviour of variable blocks (they have a drop-down menu with which the variable used in that place may be changed to be one of the other variables in use or *rename it*, but that changes all the occurrences of the variable) lead to many difficulties in usage of variables. A typical problem was that the participant, trying something or just exploring, randomly changed a default variable to point to another one, leading to the disappearance of the old variable name (because being a variable name generated by default it was not stored), then tried to return to the previous state by using the *rename* command that changed instead the name of that variable in both occurrences. Many times participants expected that dragging a variable from a pattern to a *filter* block or an *order by* field had a copy behaviour rather than a move behaviour.

Complexity of task 3. While the issues in designing queries for Tasks 1 and 2 were mostly related to understanding the user interface and the basic blocks of the language, designing the query for Task 3 proved to be a challenge in terms of actually "thinking about it" for many participants. Several of them initially tried a graph pattern in which the branch with property `dbo:locatedInArea` was used just once and then tried to solve the problem by using a filter that required the corresponding variable to be both `dbpedia:China` and `dbpedia:Nepal`. As the filter is applied to a matching at a time, such query does not give any results (a variable cannot have simultaneously two different values). To solve this task, the query needs to have two branches with property `dbo:locatedInArea` that may connect to two different variables that can then be constrained to be equal respectively to `dbpedia:China` and `dbpedia:Nepal` in a filter (see Figure 18). Even better, these two `dbo:locatedInArea` branches may directly connect to `dbpedia:China` and `dbpedia:Nepal`, respectively (as shown in Figure 13). All the participants who solved Task 3 basically achieved it by using one of these two queries (with some variations), but guessing a working query was challenging for most of them. This not an issue related to environment, but rather a recognition of the added step of reasoning (and comprehension of the system) needed to solve this task.

Unhelpful endpoint errors. The SPARQL protocol, used to communicate with SPARQL endpoints, does not give much support to the sensible communication of server errors, so very different errors (like a syntax error or a time-out) can be distinguished only through a non-standardized textual description returned by the server. As an example, in Table 2 we show the responses of different SPARQL endpoints for the same


from http://dbpedia.org/sparql
where
? mount
— rdf : type → dbo : Mountain
— dbo : elevation → ? height
— dbo : locatedInArea → ? country1
— dbo : locatedInArea → ? country2
filter ? country1 = dbpedia : China
filter ? country2 = dbpedia : Nepal
order by ? height ▼ and then by ▲ limit to first 3 rows

? mount	? height	? country1	? country2
dbpedia : Mount_Everest	8848	dbpedia : China	dbpedia : Nepal
dbpedia : Lhotse	8516	dbpedia : China	dbpedia : Nepal
dbpedia : Makalu	8485	dbpedia : China	dbpedia : Nepal

Figure 18. Alternative query to solve task 3.

type of error, a timeout during the execution of the query. SparqlBlocks shows an error in place of the table of results labelled with the truncated (not to break the user interface) error message from the server. This is often not very helpful to the user, like in the example shown in Figure 19 of a query on the WikiData endpoint.

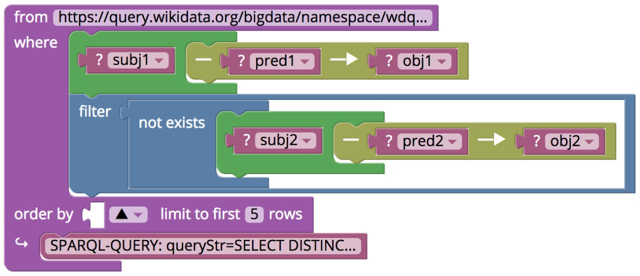
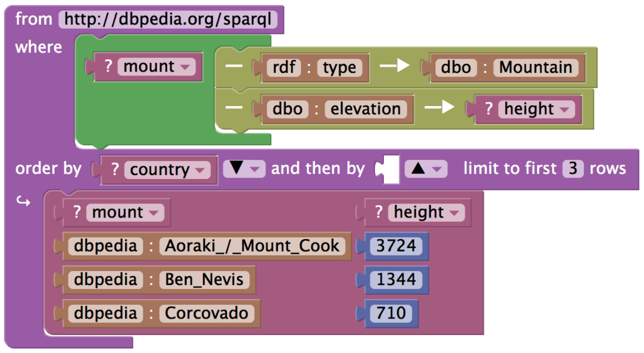

from https://query.wikidata.org/bigdata/namespace/wdq...
where
? subj1 — ? pred1 → ? obj1
filter not exists
? subj2 — ? pred2 → ? obj2
order by ▲ limit to first 5 rows
SPARQL-QUERY: queryStr=SELECT DISTINC...

Figure 19. Query block with an error shown.

Acceptance of nonmeaningful queries. One of the main advantages of the block-based interface is that syntax errors are avoided. Nevertheless, the participants sometimes built queries that either contained redundant parts or were bound to return an empty result set, for one of two reasons:

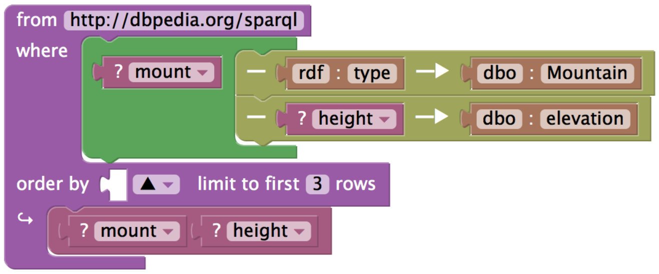
- the syntax was accepted but it actually made no sense; e.g. a variable not used in the query graph patterns was used in a filter or in a block connected through an *order by* connector (see for example the query in Figure 20);
- a query made sense in general terms but it did not respect the dataset semantics; e.g. a property was used as subject or object in a graph pattern[22] (see for example the query in Figure 21).

[22]That could make sense in special cases, like to query ontology meta-data, but, in practice, it is often just an error.


from http://dbpedia.org/sparql
where
? mount
— rdf : type → dbo : Mountain
— dbo : elevation → ? height
order by ? country ▼ and then by ▲ limit to first 3 rows

? mount	? height
dbpedia : Aoraki_/_Mount_Cook	3724
dbpedia : Ben_Nevis	1344
dbpedia : Corcovado	710

Figure 20. Syntactically correct query having a variable appearing only in the order by clause, which has no effect on the ordering.


from http://dbpedia.org/sparql
where
? mount
— rdf : type → dbo : Mountain
— ? height → dbo : elevation
order by ▲ limit to first 3 rows

? mount	? height

Figure 21. Syntactically correct query in which a property appears in the role of the object, leading to an empty result set.

10.2. Perceived Strengths

Many users saw the following as strengths of the tool. (Phrases shown surrounded by quotation marks are actual quotes from users[23].)

"Once you get used to it, it is very intuitive." After the tutorial and the execution of tasks that were quite intense with a lot of concepts to learn, the participants generally felt quite empowered and they felt that, at that point, they were able to solve similar problems more easily.

"Once you see the connector highlighted, you see where you can put the block." As previously described, participants sometimes felt frustrated for not understanding immediately where a block could be connected. Perhaps for that reason, the visual feedback given when a block is close to a compatible connector (the connector is highlighted) was much appreciated.

"I enjoyed it very much!"/"I loved using this tool!" Many participants expressed enjoyment using the tool, in accordance with the high rate given to Use Appeal in the questionnaire.

"It may be useful for education." Some participants highlighted SparqlBlocks potential as an educational tool.

"Search blocks are really useful!" A participant with previous experience with Semantic Web technologies indicated the black-box query blocks as especially useful, possibly for

[23]Quotes are used to exemplify concepts expressed by multiple users (of whom at least one of them used the given wording).

Table 2. Timeout error response messages from different SPARQL endpoints.

Endpoint	Server	Status Code	Text Message
DBpedia	OL Virtuoso	500 SPARQL Request Failed	Virtuoso S1T00 Error SR171: Transaction timed out SPARQL query: *(SPARQL)*
WikiData	Sesame	500	SPARQL-QUERY: queryStr=*(SPARQL)* *(full Java stack trace of a* `QueryTimeoutException`*)*
Linked Open Aalto Data Service	Apache Fuseki	503 Query timed out	Error 503: Query timed out Fuseki - version 2.5.0 (Build date: ...)

being familiar with the problem of exploring a dataset without prior knowledge of the used vocabularies.

10.3. Suggestions

We received the following suggestions.

To add contextual help and examples associated with the block types and accessible from the workspace.

To add retry button (or auto-retry) for when the query fails for (possibly temporary) connection problems. Now the only (admittedly suboptimal) solution is to disconnect and reconnect the query pattern stack.

Joins among queries. While it is possible to combine any number of patterns in a query to a single endpoint, it is not currently possible to design queries that access multiple endpoints. It would not be difficult to introduce this extension in the user interface, considering that SPARQL supports that feature through an optional extension called SPARQL Federation, but we should connect to a server supporting it. Public SPARQL endpoints like DBpedia do not usually support SPARQL Federation.

11. Conclusions and Future Work

We have proposed SparqlBlocks as a visual language and an interactive environment embodying a new paradigm for querying Linked Data. SparqlBlocks is based on a novel take on block programming: using blocks not only to program but also to show results, which can be incorporated in incremental design of queries. A group of users with strong computer science background but small to no experience in querying Linked Data were able to successfully design non trivial queries with the tool.

At the same time, the evaluation and analysis of the use of SparqlBlocks opens up new questions and stimulates further experimentation in the field of Linked Data access and block programming environments.

11.1. SparqlBlocks

The users had some issues related to some aspects of the user interface, especially concerning the **representation of graph patterns** and the **usage of variables**. While participants managed anyway to effectively use the tool in a relatively short amount of time, tackling these issues is probably critical to lower the bar for expertise and effort required to start using SparqlBlocks. A central point is that maintaining the full expressiveness of SPARQL has a cost in terms of having a complex visual language (the user can easily mix things in legal but meaningless ways). In many cases limiting the expressiveness may help the novice user.

11.2. SPARQL

Some issues or requests push the limits of current SPARQL infrastructure.

Better feedback on server errors would require going beyond the current SPARQL protocol, while in the medium term it could be tackled by designing a layer that may interpret the output of the most used kinds of SPARQL endpoints and give a semantically well-defined answer.

Proposing more complex queries, for example joining patterns from multiple endpoints, requires not just adding the missing pieces of the full SPARQL language, but also having on the server side a system capable of executing those complex distributed queries in an efficient way —which is not a fully solved problem, neither in practice nor in theory.

11.3. Block Programming

Some issues should be analysed in the context of block programming environments.

How to best represent optional component with default values. This concerns the trade-off between (1) offering blocks with defaults (for example through shadow blocks) versus (2) requiring filled connections versus (3) allowing empty connections (with implicit defaults).

Management and visual representation of variables. In several block programming environments, variables are all global, thus sacrificing the principle of information hiding in order to gain the possibility of visually interacting with variable blocks without having to manage scopes. In some cases, like MIT App Inventor 2, the management of variables has been designed to permit lexically scoped variables [31]. Locally scoped variables are introduced as parameters of functions or through a specific block that initializes a set of local variables and encloses the instructions in which they may be used. In SPARQL, variables are locally scoped to queries, but in the language there is no explicit declaration of them. Furthermore, while variables in basic graph patterns may be novel or refer to existing variables, variables in expressions (for *filter* and *order by* fields) should have already been introduced in some basic graph pattern of the same query. We chose to keep the system simple by managing the variables as if they were all global and leaving it to the user to deal with them in the correct way. There is possibly room for improvements by devising a visual representation of the scope system of SPARQL.

Shapes and allowed connections. The recognition of available connections is paramount to the effective usage of a block programming environment, but a complex language may have many types (hence potentially many connection types) and connectors that accept multiple types. Which is the "right" trade-off in the number of different shapes? How can the UI be augmented to further support fine-grained distinctions among types in a way that the user would still be able to see the potential connections at a glance? Blockly is rather conservative in having just two types of connectors. The OpenBlocks system [32], upon which the original MIT App Inventor and StarLogo TNG were based, had support for more connector shapes and supported polymorphism. Some prototype systems went to the length of supporting arbitrary complex derivative types (like tuples or functions) by graphically composing basic connector shapes through a set of rules [33, 34]. We chose not to follow this line, as we were interested in first analysing the potential of the approach for grasping the essential aspects of SPARQL. Further experimentation will be needed to establish the trade-off between the greater expressivity given by an increase in the available possibilities for manipulation and the consequent increase of cognitive load. It could also be interesting to explore how a hierarchical system of types may be represented through the use of different connection shapes.

Richness and organization of the toolbox. The organization of the toolbox is paramount to user's comprehension of language affordances. It is not trivial to find a compromise on the number of offered blocks. For example, blocks representing multiple operators (e.g., the logic operator block, used for *and* and *or*) are shown just with default one (in this case *and*) to avoid overloading the toolbox. To use another operator, the user must use a drop-down menu and change it. This was found to be non evident to several users. So, at least for some cases, it may be worth to show already all or most of the options as separate blocks in the toolbox, as, for example, MIT App Inventor does.

Extension of the SparqlBlocks' paradigm to similar applications. The results of the evaluation of the tool are so far promising, so it could be interesting to extend the paradigm to similar languages. The proposed incremental approach and UI could potentially be applied to multiple query languages and data models.

12. Acknowledgments

We thank all the reviewers for the valuable contributions they made to the development of this paper. We are specially grateful to Franklyn Turbak for his help and tireless commitment to improve this paper through countless suggestions and amazing editing work. We also thank all the volunteers who participated in the user study for their time.

References

[1] T. Berners-Lee, "Linked data," 2006. [Online]. http://www.w3.org/DesignIssues/LinkedData.html

[2] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data – The Story So Far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[3] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C REC 25, February 2014. [Online]. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225

[4] S. Harris *et al.*, "SPARQL 1.1 Query Language," W3C REC 21, March 2013. [Online]. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[5] S. Ferré, "Sparklis: a SPARQL Endpoint Explorer for Expressive Question Answering," in *Proceedings of the 2014 International Conference on Posters & Demonstrations Track (ISWC-PD'14)*, vol. 1272. CEUR-WS, 2014.

[6] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt, "NITELIGHT: A Graphical Tool for Semantic Query Construction," in *Proceedings of the 5th International Workshop on Semantic Web User Interaction (SWUI '08)*, vol. 543. CEUR-WS, 2008.

[7] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[8] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media, Inc., 2014.

[9] P. Bottoni and M. Ceriani, "Linked data queries as jigsaw puzzles: a visual interface for SPARQL based on Blockly library," in *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter (CHItaly 2015)*. ACM, 2015, pp. 86–89.

[10] P. Bottoni and M. Ceriani, "SPARQL Playground: a block programming tool to experiment with SPARQL," in *Proceedings of the ISWC 2015 workshop on Visualizations and User Interfaces for Ontologies and Linked Data (VOILA 2015)*, 2015, p. 103.

[11] D. Peterson, S. S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes," W3C REC 5, April 2012. [Online]. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405

[12] M. Duerst and M. Suignard, "Internationalized Resource Identifiers (IRIs)," RFC 3987 (Proposed Standard), Internet Engineering Task Force, Jan. 2005. [Online]. http://www.ietf.org/rfc/rfc3987.txt

[13] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986 (INTERNET STANDARD), Internet Engineering Task Force, Jan. 2005, updated by RFC 6874. [Online]. http://www.ietf.org/rfc/rfc3986.txt

[14] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "SPARQL 1.1 Protocol," W3C REC 21, March 2013. [Online]. http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[15] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia," *Semantic Web Journal*, vol. 5, pp. 1–29, 2014.

[16] N. Fraser, "Blockly — Google Developers," accessed: 2017-05-10. [Online]. https://developers.google.com/blockly/

[17] L. Rietveld and R. Hoekstra, "YASGUI: Not just another SPARQL client," in *Proceedings of the Extended Semantic Web Conference (ESWC 2013) Satellite Events*. Springer, 2013, pp. 78–86.

[18] J. Borsje and H. Embregts, "Graphical query composition and natural language processing in an RDF visualization interface," B.S. Thesis, Erasmus School of Economics and Business Economics, Erasmus University, Rotterdam, 2006.

[19] F. Haag, S. Lohmann, S. Siek, and T. Ertl, “QueryVOWL: Visual composition of SPARQL queries,” in *Proceedings of the Extended Semantic Web Conference (ESWC 2015) Satellite Events*. Springer, 2015.

[20] F. Haag, S. Lohmann, S. Bold, and T. Ertl, “Visual SPARQL querying based on extended filter/flow graphs,” in *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI ’14)*. ACM, 2014, pp. 305–312.

[21] J. Groppe, S. Groppe, and A. Schleifer, “Visual query system for analyzing social semantic web,” in *Proceedings of the 20th International Conference Companion on World Wide Web (WWW ’11)*. ACM, 2011, pp. 217–220.

[22] M. M. Zloof, “Query-by-Example: a data base language,” *IBM Systems Journal*, vol. 16, no. 4, pp. 324–343, 1977.

[23] H. N. M. Quoc, M. Serrano, D. Le-Phuoc, and M. Hauswirth, “Super stream collider-linked stream mashups for everyone,” in *Proceedings of the Semantic Web Challenge co-located with ISWC2012*, 2012.

[24] M. Resnick, “StarLogo: An environment for decentralized modeling and decentralized thinking,” in *Conference Companion on Human Factors in Computing Systems (CHI ’96)*. ACM, 1996, pp. 11–12.

[25] J. Gorman, S. Gsell, and C. Mayfield, “Learning relational algebra by snapping blocks,” in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE ’14)*. ACM, 2014, pp. 73–78.

[26] Y. N. Silva and J. Chon, “DBSnap: Learning database queries by snapping blocks,” in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE ’15)*. ACM, 2015, pp. 179–184.

[27] A. Jain, J. Adebayo, E. de Leon, W. Li, L. Kagal, P. Meier, and C. Castillo, “Mobile Application Development for Crisis Data,” *Procedia Engineering*, vol. 107, pp. 255–262, 2015.

[28] S. Dasgupta and B. M. Hill, “Scratch Community Blocks: Supporting children as data scientists,” in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI ’17)*. ACM, 2017, pp. 3620–3631.

[29] T. R. G. Green, “Cognitive dimensions of notations,” *People and Computers V*, pp. 443–460, 1989.

[30] T. R. G. Green and M. Petre, “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework,” *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[31] F. Turbak, D. Wolber, and P. Medlock-Walton, “The design of naming features in App Inventor 2,” in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2014, pp. 129–132.

[32] R. V. Roque, “OpenBlocks: an extendable framework for graphical block programming systems,” Master’s thesis, Massachusetts Institute of Technology, 2007.

[33] M. Vasek, “Representing expressive types in blocks programming languages,” Undergraduate thesis, Wellesley College, 2012.

[34] S. Lerner, S. R. Foster, and W. G. Griswold, “Polymorphic blocks: Formalism-inspired UI for structured connectors,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI ’15)*. ACM, 2015, pp. 3063–3072.