A Drag-and-Drop UI for MS Academic

Leonard Pabst
leonard.pabst@student.hpi.de
Hasso Plattner Institute
Potsdam, Brandenburg, Germany

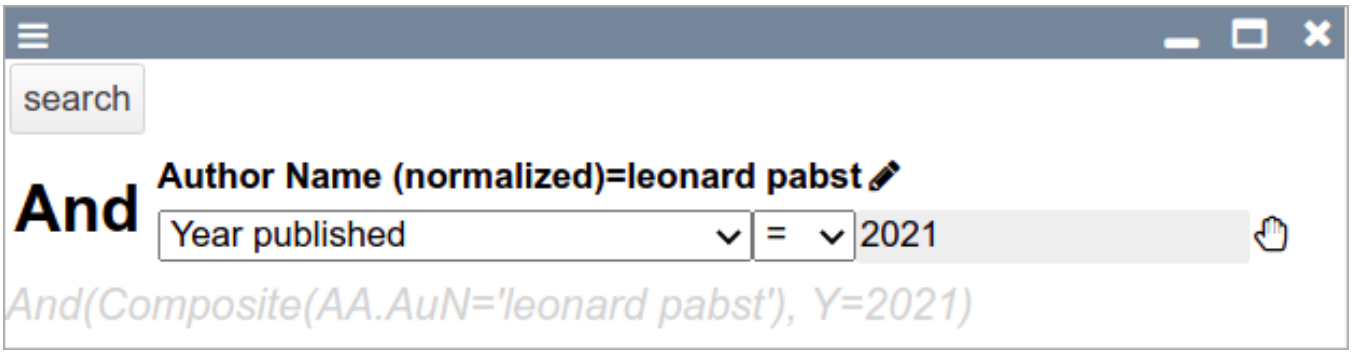

Figure 1: The User Interface as presented in this paper.

ABSTRACT

Microsoft (MS) Academic offers a sophisticated query language to access its extensive data base of academic publications. This query language provides users with the ability to write complex requests, allowing them to find exactly what they are looking for. However, the user experience in the construction of such complex requests is very poor; the language is prone to syntax errors and queries quickly become incomprehensible to the users. We present a user interface for the MS Academic query language that hides incomprehensible parts of queries and only allows for appropriate interactions. To evaluate our approach, we built examples in Lively4. We discovered that even our simple examples bring a benefit to the user experience by hiding complexity of the API.

KEYWORDS

design, graphical dsl, projectional editing, language, visual language, usability, abstract syntax tree

ACM Reference Format:
Leonard Pabst. 2021. A Drag-and-Drop UI for MS Academic. In *Proceedings of Programming Languages: Concepts, Trends, Environments (PLCTE'20).* ACM, New York, NY, USA, 8 pages.

PLCTE'20, WiSe20/21, Potsdam, Germany

1 INTRODUCTION

Microsoft Academic[1] holds over 250 million academic publications [1]. With its large data base it is one of the leading sources of academic papers [2]. To access these papers, Microsoft Academic offers its own query language.

However, browsing through the Microsoft Academic data base is difficult. Small queries are easy to write, but hard to refine or reuse. Instead of having a typical browsing experience, it feels like one has to program to get to the desired results.

A user has to follow a certain syntax and schema. Because of that, one can easily make syntax errors. In addition to that one has to use elements like IDs, seemingly random sequences of numbers, that are not readable for oneself if the results should be precise. That leads to the user being unable to understand his or her own query after writing it.

With a graphical UI, one could hide unreadable elements and only allow interactions that do not lead to syntax errors. An approach might be a graphical DSL like Blockly. In this paper, we present a prototype built in Lively4 that already improves the user experience.

2 MOTIVATION

Microsoft Academic is a search engine for academic papers. It uses a dedicated query language as further explained in Section 3. This paper challenges the user experience when browsing the MS Academic database using its query language. To understand the status quo, we take the perspective of a user. Imagine she wants to find a paper about which she knows the following:

(1) published in 2020
(2) authors: Tom Beckmann, Robert Hirschfeld.

This should be enough to find the specific paper. Now the user only has to write the corresponding query.

[1] https://academic.microsoft.com/home

The user follows the query expression syntax by Microsoft Academic[2]. She first writes the three parts of the query separately and starts by selecting the value 2020 for the attribute year (Y): $Y = 2020$.

This is not as straightforward for the author *AA*, since this is a so-called "composite" attribute. These composite attributes have components, which work as a kind of sub-attribute, indicated by a dot. Knowing that, the user tries to match the author name "tom beckmann" to the attribute *AA.AuN: AA.AuN='tom beckmann'*. However, this does not work yet, because composite attributes additionally have to be enclosed in the *Composite*() function: *Composite(AA.AuN='tom beckmann')*.
Last, but not least, she writes the last part of the query: *(Composite(AA.AuN='robert hirschfeld')*

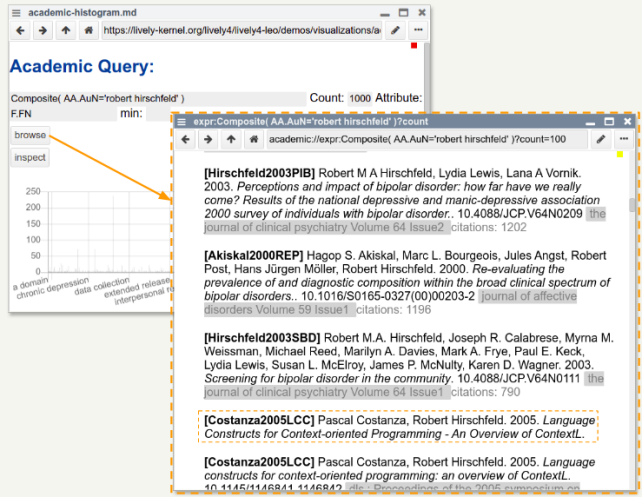

Figure 2: Results of an ambiguous query

Figure 2 shows how rather common names can make it difficult to find papers of the right author. For that reason, the user wants to use the ID of the correct Robert Hirschfeld. To do that, she finds one of his papers (in Figure 2, entry number 4) and identifies his ID through that: *Composite(AA.AuId=2154319088)*. Same for Tom Beckmann: *Composite(AA.AuId=3047607744)*

She combines these separate queries with *And*() functions: *And(And(Composite(AA.AuId = 2154319088), Y = 2020), Composite(AA.AuId = 3047607744))*.

This whole user experience feels like programming to her. Even though her search parameters are not that complicated, writing the right query is. This example did not include any of the syntax errors she received throughout this process. Also, if the user wanted to reuse that query to find a different paper from Robert Hirschfeld, she needs to find out again, which of the IDs belong to whom, making reutilization much more difficult. Microsoft Academic has a rich database, but gives users browsing through it a hard time. How could a dedicated user interface hide the unreadable parts of the query and help avoid syntax errors? How should possible interaction with the query look like?

[2]https://docs.microsoft.com/en-us/academic-services/project-academic-knowledge/reference-query-expression-syntax

Table 1: Overview over the elements of a simple query

Component	Description
Attribute	1-4 letters, can have components: AA.AuN
Comparator	equality or inequality signs
Value	following attribute type, ranges with brackets

3 BACKGROUND

Block-based programming approaches like Scratch [4, 5] or Blockly [3] show how graphical dsls can work. We can refer to them when thinking about interaction possibilities or design.
For the use case of this paper, it makes sense to look into Microsoft Academic's query language in detail. Section 2 already gives an impression how queries might look like, but to be able to parse textual queries into a graphical user interface, one has to look even further.
A simple query follows the same general structure: *[Attribute] [Comparator] [Value]* further explained in Table 1. A concatenated query consists of multiple simple queries combined through an *And*() or *Or*(): $Or(Y = 1985, Y = 2008)$.
An **Attribute** has the following properties:

- Mostly consists of 1-4 letters, e.g. *Y* for Year, *Ti* for Title.
- Can have components, similar to attributes of objects in object oriented programming and that are basically sub-attributes on their own, e.g. *AA* for Author/Affiliation has *AA.AuN* for the author's name and *AA.AuId* for the author's ID.
- In a query, components need the *Composite*() function around the query, e.g. *Composite(AA.AuN='leonard pabst')*

Only characters from a relatively small group can be used as **Comparators**:

- = means single or prefix value.
- == means exact single value
- >, <, >=, <=, = can be used for ranges

The **Value** must follow the type of the attribute:

- Single quotation marks are necessary around a value for a textual attribute.
- Brackets around tuples can be used to query ranges of number values. Squared brackets indicate inclusion, round brackets exclusion, e.g. $Y = [2010, 2012)$ queries papers from 2010-2011.

4 MAIN PART

We can formulate three main assumptions for room of improvement of the usability of the current query user interface:

(1) The user interface can hide or cover some information that is necessary for the query, but unnecessary or irritating for the user: the ID of the author or paper and the Composite() function, since components of attributes are already indicated by a dot. can reduce cognitive load [6, 7].
(2) The user could profit from limited interaction, e.g. instead of writing out attributes (and maybe adding a typo), choosing

the attribute from a list. Therefore, syntax errors could be avoided.

(3) Browsing and concatenation of queries could be supported through interactions like drag and drop inspired from block based user interfaces.

In this section, we will describe our approach for a prototype that improves the user experience following our previous assumptions. This prototype's development environment is Lively4[3]. Lively already offers tools for browsing and querying academic papers using the Microsoft Academic API, as Figure 3 shows.

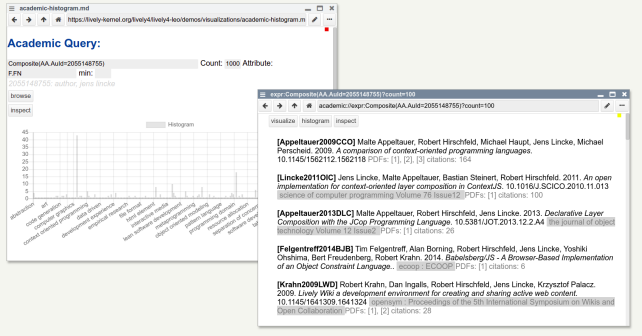

Figure 3: Existing tools for MS Academic in Lively4

4.1 Architecture

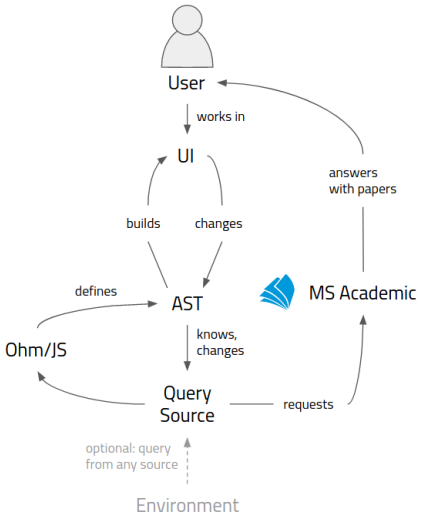

Figure 4: The architectural diagram of the prototype

Figure 4 shows the architecture behind our prototype. Given a query from any source (for example by dragging and dropping a query in the UI), we use an Ohm/JS grammar that parses the textual query into an abstract syntax tree (AST). Depending on the type of query, it has the type of a simple query (Figure 5) or a conjunction query (Figure 6, with simple queries as sub objects "left" and "right").

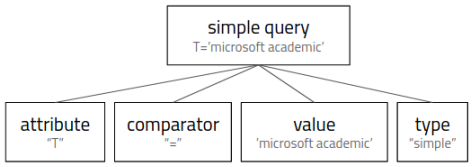

Figure 5: Visualization of a simple query's AST

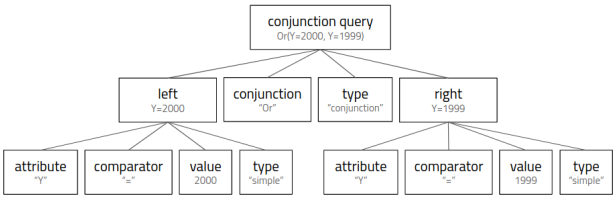

Figure 6: Visualization of a conjunction query's AST

The AST still stores the original query, being able to send said query to Microsoft Academic to receive the papers that fulfill the query in response. The AST also dynamically builds the user interface (UI). A query element can be in the read-mode, which aims at comprehensibility and allows users to drag and drop elements onto each other or in the edit-mode, in which users can alter the actual query. In this proof of concept, the UI assumes the shape of a rotated tree as Figure 7 shows. For demonstration purposes, we display the underlying query source as well. If any of the attributes in the AST is an ID like *AA.AuId*, the AST substitutes this attribute with the corresponding name, more on that in Section 4.2. That way, in the UI, the query itself and the ID are not directly visible anymore in the read-mode. When the user works on the UI, he automatically makes changes to the AST, which then parses the textual query anew from the UI to keep query and UI in sync.

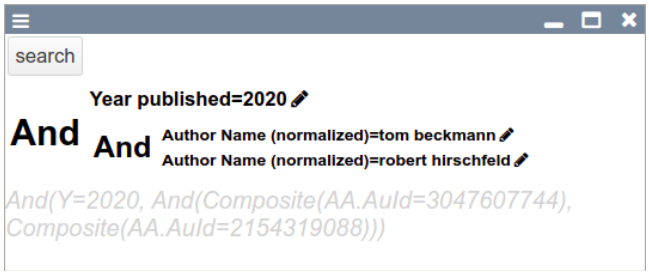

Figure 7: Rotated tree structure of the UI

[3]https://github.com/LivelyKernel/lively4-core

4.2 Implementation

Let's look a bit deeper at certain elements of the UI.

The QueryUI consists of different query elements that are dynamically built from the AST, which is parsed from the textual query as explained in Section 4.1. A conjunction query element combines two full further queries with a conjunction (*And* or *Or*). The associated query element consists of a text element for the *conjunction* and besides that two smaller sub query elements (in regards to the font size), arranged in a rotated tree-structure, see Figure 8.

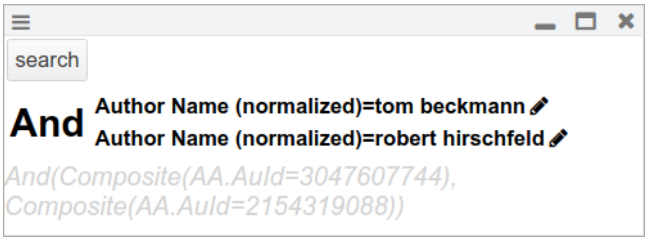

Figure 8: The UI rendered from a conjunction query

A simple query element can have two modes, the edit- and the read-mode, as Section 4.1 mentions.

In the edit-mode, it consists of a dropdown for the *attribute*, a dropdown for the *comparator*, an input field, following the type of the attribute (e.g. a date picker for the *Date* attribute), for the *value* and a toggle button to switch to read-mode as Figure 9 presents.

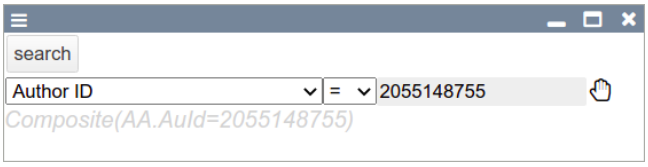

Figure 9: A simple query in edit-mode

In the read-mode, it simply consists of a span element for *attribute*, *comparator* and *value* respectively and also the toggle button to switch to edit-mode. In this mode, users can drag and drop queries onto each other. Additionally, when the user hovers over the simple query element in read-mode, two buttons (*And* and *Or*) appear as a halo (Figure 10). Clicking one of these buttons transforms the simple query element in a conjunction with the original query set as one of the sub queries. This is how the user can "grow" the AST, i.e. adding attributes to the query.

Attributes that represent IDs are shown as such in the edit-mode, but are substituted with the corresponding name attribute in the read-mode (see Figure 11). That way, the user can understand the query better and reuse it, while the query element itself still knows the query source and works with the more precise ID. In both modes, the abbrevations that are necessary for the query (like *AA.AuN*) are mapped and substituted with short descriptions (Author Name (normalized)) following the Microsoft Academic Paper Entity documentation[4] with slight adaptations for some attributes.

[4]https://docs.microsoft.com/en-us/academic-services/project-academic-knowledge/reference-paper-entity-attributes

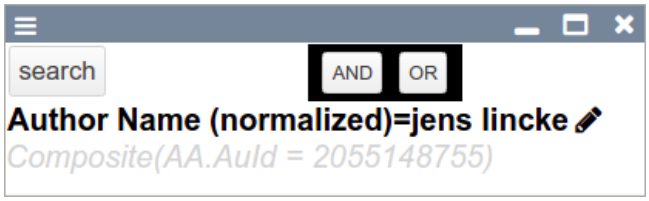

Figure 10: A halo to create conjunction queries

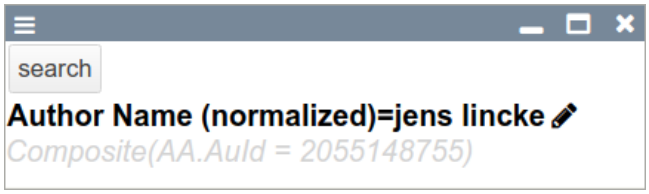

Figure 11: The ID in the query is substituted with the name in the UI

When dragging a query element and dropping it onto another element or another UI, the user actually transports the query source to the new element by setting it in the *'text'* mimetype of the drag and drop event, which updates itself according to this new query. That way, when attributes like *AA.AuId* (author ID) are subsituted with the author's name in the UI, the new element still understands the original query source as visualized in Figure 12. At the same time, the user can highlight a textual query from any source and drop it onto the query element, updating it if possible.

In Section 2, we already identified the *Composite*() function as a redundant part of the query, since composite attributes are already indicated by a dot. For this reason, we do not display the *Composite*() function in the UI, but to keep track of this information and to be able to distinguish between composite and other queries, we store the respective string ("simple" or "composite") in the query's *type* value (see Figure 5).

5 DISCUSSION AND EVALUATION

Looking back at the use case from Section 2, what changes with our QueryUI? The user still searches for a paper with the following properties:

(1) published in 2020
(2) authors: Tom Beckmann, Robert Hirschfeld.

The user starts by selecting the first attribute she wants to search for: she selects the *Year published* and sets it equal to 2020. Through hovering over this existing query element, she chooses an *And*()-conjunction to add the next query element. At this point, she still has to add the author's ID to the query manually to be sure to receive the correct results. We discuss later in this section, what can be possible additions that improve that situation. For now, she selects *Author.ID* as the attribute and sets its value to 3047607744. As soon as she is done editing this query element, it changes to *Author Name (normalized) = tom beckmann*. She then adds another *And*()-conjunction, drags the query she just wrote and drops it on the new query element, therefore copies it, and edits its "hidden" ID to

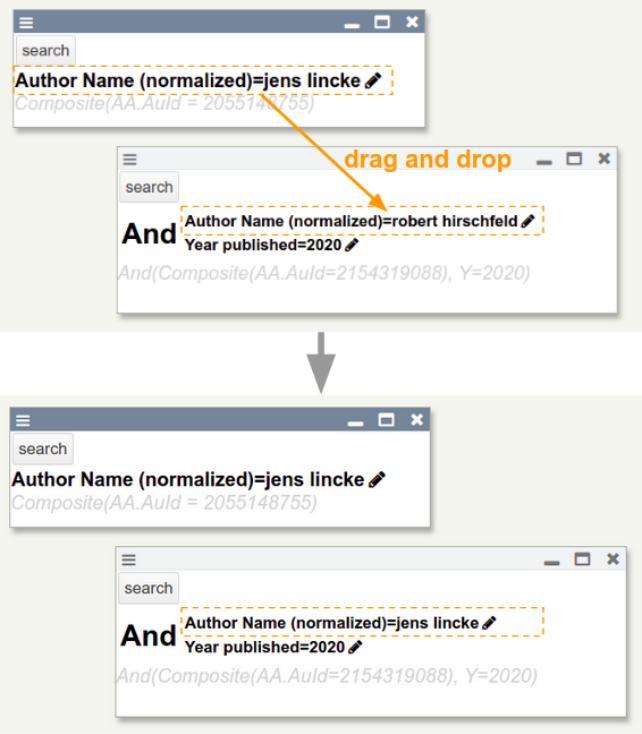

Figure 12: Drag and drop of a query element into another

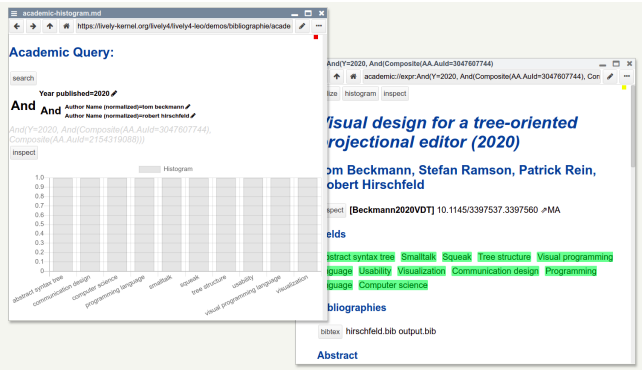

Figure 13: Finding an academic paper with the new QueryUI

2154319088, finishing the query. Figure 13 shows the corresponding UI in the academic-histogram environment

Table 2: Changes in browsing MS Academic with the new QueryUI

Task	Without QueryUI	With QueryUI
Read & understand query	–	✓
Able to reuse query	–	✓
Combine queries	–	✓
Avoid syntax errors	✗	✓
Edit query source directly	✓	✗
Long queries are handy	✗	✗
Interaction with BibTeX entries	✗	✗

✗ means: not achievable
– means: somewhat achievable
✓ means: easily achievable

After testing out the example from the beginning with the new UI and looking at the overview in Table 2, we can conclude that the overall usability of the browsing experience in Microsoft Academic improved, meaning that the user is less likely to produce syntax errors and needs less interactions with the UI to actually browse through Microsoft Academic's data base. On another note, there are some obvious features missing and the user experience is still not optimal as mentioned in the example. A completely different approach might be to only let the user select a list of attributes with respective values instead of building trees with multiple *And*()-conjunctions. This limits the possible complexity the the query, but could also just be enough to browse academic papers. However, we did not follow up on that approach.

As a follow up on this prototype, we consider the following features to be most beneficial to the user experience:

(1) drag-and-drop interaction with the results on the UI, e.g. being able to drag-and-drop the author from a BibTeX entry onto a query element, whereby the AST automatically receives the new author, adds it to the query and updates the query source
(2) upon entering an author's name, the user can open another tool displaying appropriate author IDs with some of their respective publications for the user to choose from
(3) recommendations on which attributes to add to the query to further narrow down the results
(4) add more logic to the queries with integrating scripts on the results in Lively, e.g. only show papers with less than 10 co-authors (probably only useful for selected users)
(5) add a button to switch between different representations of the query: seeing the easy-to-read query in the UI (default), seeing query source in the UI or maybe even seeing the textual query source without the UI (the current status quo)

6 RELATED WORK

While many of the findings following research on block-based languages like Scratch are quite domain-specific and often refer to programming, we are still able to derive some insights to our work.

Scratch shows how a well-designed user interface can simplify the user experience and lower the initial hurdle even for more complicated use cases like programming [4, 5]. Aspects that we

can derive from Scratch: eliminating syntax errors by modifying the interface to only allow valid operations helps users to focus on more interesting problems right away. Block shapes and visual feedback help users to interact inuitively with the UI right away[3], which is something, that can even be expanded on in our prototype. Scratch uses only a small number of commands. We already only show attributes in the UI that support at least one operation, but it might be interesting to see, which attributes users actually utilize when browsing MS Academic to see if some more of them could be ignored in the UI.

On another note, Blockly allows for a reduced cognitive load by hiding API complexity [6]. Our prototype does the same when hiding IDs or finding descriptions for attributes in the UI, hopefully helping users to focus on what they actually want when browsing Microsoft Academic: finding academic publications.

7 CONCLUSION

In this paper, we presented a prototype built in Lively that realizes an improved user experience when browsing the Microsoft Academic data base. The proposed approach is quicker and simpler to use and less prone to syntax errors, but it still is not perfect. The user can now better understand a given query and can easily reuse the query or only parts of it by using drag-and-drop to combine queries. Avoiding syntax errors comes at the cost of not being able to edit the query "source code" directly. On the downside, long queries are still not handy and the user is missing the possibility to reuse a query's results (BibTeX entries) in another query. Most of these issues can be addressed in future work.

REFERENCES

[1] Topic Browser | ETAP - Publications | Microsoft Academic, https://academic.microsoft.com/publications. Last accessed 18 Feb 2021

[2] Harzing, A.-W., & Alakangas, S. (2017). Microsoft Academic is one year old: the Phoenix is ready to leave the nest. Scientometrics, 112(3), 18871894.

[3] Fraser, N. (2015). Ten things weve learned from Blockly. In 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) (pp. 4950).

[4] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Silverman, B. (2009). Scratch: programming for all. Communications of The ACM, 52(11), 6067.

[5] Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. ACM Transactions on Computing Education, 10(4), 16.

[6] Olney, A. M., & Fleming, S. D. (2019). A Cognitive Load Perspective on the Design of Blocks Languages for Data Science. In 2019 IEEE Blocks and Beyond Workshop (B&B).

[7] Bart, A. C., Gusukuma, L., & Kafura, D. (2017). Really pushing my buttons: Affordances in block interfaces. In 2017 IEEE Blocks and Beyond Workshop (B&B) (pp. 1720).

8 APPENDIX

Table 3: Schema of an academic paper entity

Name	Description	Type	Operations
AA.AfId	Author affiliation ID	Int64	Equals
AA.AfN	Author affiliation name	String	Equals, StartsWith
AA.AuId	Author ID	Int64	Equals
AA.AuN	Normalized author name	String	Equals, StartsWith
AA.DAuN	Original author name	String	None
AA.DAfN	Original affiliation name	String	None
AA.S	Numeric position in author list	Int32	Equals
AW	Unique, normalized words in abstract[5]	String[]	Equals
BT	BibTex document type[6]	String	None
BV	BibTex venue name	String	None
C.CId	Conference series ID	Int64	Equals
C.CN	Conference series name	String	Equals, StartsWith
CC	Citation count	Int32	None
CitCon	Citation contexts[7]	Custom	None
D	Date published in YYYY-MM-DD format	Date	Equals, IsBetween
DN	Original paper title	String	None
DOI	Digital Object Identifier	String	Equals, StartsWith
E	Extended metadata	Extended	None
ECC	Estimated citation count	Int32	None
F.DFN	Original field of study name	String	None
F.FId	Field of study ID	Int64	Equals
F.FN	Normalized field of study name	String	Equals, StartsWith
FamId	Primary paper ID of the paper family[8]	Int64	Equals
FP	First page of paper in publication	String	Equals
I	Publication issue	String	Equals
IA	Inverted abstract	InvertedAbstract	None
Id	Paper ID	Int64	Equals
J.JId	Journal ID	Int64	Equals
J.JN	Journal name	String	Equals, StartsWith
LP	Last page of paper in publication	String	Equals
PB	Publisher	String	None
Pt	Publication type[9]	String	Equals
RId	List of referenced paper IDs	Int64[]	Equals
S	List of source URLs of the paper, sorted by relevance	Source[]	None
Ti	Normalized title	String	Equals, StartsWith
V	Publication volume	String	Equals
VFN	Full name of the Journal or Conference venue	String	None
VSN	Short name of the Journal or Conference venue	String	None
W	Unique, normalized words in title	String[]	Equals
Y	Year published	Int32	Equals, IsBetween

Documentation: https://docs.microsoft.com/en-us/academic-services/project-academic-knowledge/reference-paper-entity-attributes

[4] excluding common/stopwords

[5] 'a': Journal article, 'b': Book, 'c': Book chapter, 'p': Conference paper

[6] List of referenced paper ID's and the corresponding context in the paper (e.g. [123:["brown foxes are known for jumping as referenced in paper 123", "the lazy dogs are a historical misnomer as shown in paper 123"])

[7] If paper is published in multiple venues (e.g. pre-print and conference) with different paper IDs, this ID represents the main/primary paper ID of the family. The field can be used to find all papers in the family group, i.e. FamId=<paper_id>

[8] 0: Unknown, 1: Journal article, 2: Patent, 3: Conference paper, 4: Book chapter, 5: Book, 6: Book reference entry, 7: Dataset, 8: Repository

```
Academic {
 Exp =
   AcademicQuery

 AcademicQuery = Attribute Comparator Value -- simple
   | ("And" | "Or") "(" AcademicQuery "," AcademicQuery ")" -- complex
   | "Composite(" CompositeQuery ")" -- composite
 CompositeQuery = Attribute "." Attribute Comparator Value -- simple
   | ("And" | "Or") "(" CompositeQuery "," CompositeQuery ")" -- complex

 Comparator =
   PartialComparator "="?
 PartialComparator =
   "=" | "<" | ">"

 Attribute (an attribute) =
   letter letter? letter? letter?

 Value (a value) =
   "\'" alnum* "\'" -- string
   | Number
   | Date
   | ( "[" | "(" ) Number "," Number ( "]" | ")" ) -- numberRange
   | ( "[" | "(" ) Date "," Date ( "]" | ")" ) -- dateRange

 Number =
   digit+
   Date =
   "\'" Number "-" Number "-" Number "\'"
}
```

Figure 14: Ohm grammar for a Microsoft Academic query