Knowledge Graph Programming with a Human-in-the-Loop: Preliminary Results

Yuze Lou, Mahfus Uddin, Nathaniel Brown, Michael Cafarella
lyzlyz@umich.edu,mahfusu@umich.edu,nthnb@umich.edu,michjc@umich.edu
University of Michigan
Ann Arbor, Michigan

ABSTRACT

In this paper we introduce *knowledge graph programming*, a new method for writing extremely succinct programs. This method allows programmers to save work by writing programs that are brief but also underspecified and underconstrained; a human-in-the-loop "data compiler" then automatically fills in missing values without the programmer's explicit help. It uses modern data quality mechanisms such as information extraction, data integration, and crowdsourcing. The language encourages users to mention knowledge graph entities in their programs, thus enabling the data compiler to exploit the extensive factual and type structure present in modern KGs. We describe the knowledge graph programming user experience, explain its conceptual steps and data model, describe our prototype KGP system, and present some preliminary experimental results.

1 INTRODUCTION

Knowledge graph programming (KGP) aims to enable extremely succinct programs. Its core design idea is to allow users to save effort by underspecifying the program, and to then automatically fill in missing program details through modern data quality methods, such as data integration, information extraction, crowdsourcing, and so on. User programs make frequent references to entities drawn from a knowledge graph, such as Wikidata [5] or DBpedia [16]. By combining program logic with KG references, the succinct source code provides a clean logical definition of program execution, and gives the KGP system substantial implicit hints about how to best fill in missing values.

KGP shares some goals with program synthesis [7, 12, 13, 21] and ML-assisted programming methods [1, 9, 10] — namely, to greatly reduce the work done by a human programmer. However, we focus on efficiently producing program-like artifacts, rather

Figure 1: The program `Fly(Obama | Washington DC | Paris)` **can yield a professional-looking animation rendered on a world map, even though it is a single line.**

than producing source code per se. Moreover, our proposed design relies heavily on methods from the data management literature, such as relational schema design, data integration, and information extraction, as opposed to logical solvers.

Knowledge graph programming comprises three steps:

- The *knowledge programmer* writes a succinct but underspecified program, using frequent references to entities and predefined actions in a knowledge graph. A single user-written knowledge program can potentially be compiled into multiple target artifacts.
- The *target module(s)* translates the knowledge program's execution into a *data scaffold*: a set of data slots that must be filled before a useful software artifact can be generated. There is a separate target module for each type of target software artifact: one for phone apps, one for two-dimensional animations, one for web apps, and so on. Target modules are pluggable pieces of code that can expand KGP functionality over time.
- The *data compiler* fills in an empty scaffold's data slots as accurately as it can. Once populated, the data scaffold can be transformed into the desired software artifact.

The success of this final data compilation step — to populate a scaffold of missing values accurately, quickly, and at reasonable financial cost — is the key to a useful KGP system. The data compiler makes the difference between a usable software artifact and an unusable one.

Example — Figure 1 shows a frame from an animation that illustrates a plane flying from Washington, D.C. to Paris, France[1]. The animation lasts a few seconds and would be suitable for inclusion in a PowerPoint presentation, or as a segment in a longer video. It is the result of a one-step user program: Fly(Obama | Washington DC | Paris).

In this case, the knowledge programmer has used an *action* (Fly) from a predefined controlled list and has made casual textual references to three entities she assumes are in a backing knowledge graph: former president Barack Obama, and the cities Washington D.C. and Paris, France. She has further chosen to compile the program using the 2DMap target module.

The KGP system has: (1) mapped the programmer's textual references to unambiguous KG entities[2] to yield a simple knowledge program; (2) applied the 2DMap target module to this program to yield an empty data scaffold; (3) applied the data compiler to fill in the missing values, in this case mainly via transforming data values found in Wikidata; and (4) has created the 2DMap animation using the populated data scaffold.

Design Considerations — KGP saves programmers effort by not asking them to do much of the work needed to actually generate an artifact. In the above example, the programmer has not bothered to explicitly disambiguate the entity references (say, Paris, France vs Paris, Texas), has not bothered to look up whether those places actually exist in a dictionary of locations, has not looked up their real-world latitude and longitude, has not figured out how to map those latitude and longitude pairs to on-screen pixel locations, and has not found a bitmap of a plane to illustrate "Fly". The KGP system has filled in all of these values without bothering the programmer.

The KGP system has been able to do so because of the many *implicit* hints the knowledge programmer has provided, whether wittingly or not. For example, the two location parameters to Fly() exist in the back-end knowledge graph, and have been typed with city information, and have associated lat/long values. Although the expressivity of the program is limited by the actions that are available, a large general-purpose library of actions should enable a wide set of programs. However, it is certainly not the goal of KGP to be an entirely general-purpose programming language.

Vision — This is a very simple example, chosen because it is simple and easy to illustrate. However, our ambition for the project is to produce real pieces of increasingly useful software, which use more ambitious programming models and more complicated KG entities and actions. The long-term technical bet behind KGP's design is similar to that made by declarative query researchers in the mid-1970s [3, 19]: that in many cases, software systems can write better code than humans when provided with a compact spec (SQL / KGP programs) and substantial background information (database statistics / knowledge graphs and other sources of evidence). The core difference between these technical bets is KGP's focus on non-performance qualities of the generated program.

Organization — The remainder of this paper is organized as follows. We describe a detailed user interaction in Section 2. In Section 3 we describe the KGP intertwined execution and data models, and describe the data compiler in Section 4. In Section 5 we describe our prototype software system. We describe some preliminary experiments in Section 6. Finally, we cover related work in Section 7 and conclude with a discussion of future work in Section 8.

2 USER EXPERIENCE

In this section we expand upon the simple example from above. First, however, some concrete background information on knowledge graphs is useful.

2.1 Knowledge Graphs

A *knowledge graph* (KG) is a graph-structured database that aims at universal coverage within a target domain. Examples covering general-interest data include DBpedia [16], Wikidata [5], YAGO [18], KnowItAll [6], and industrial projects such as Google's Knowledge Graph and Microsoft's Satori. More narrow topics include MusicBrainz, GeoNames, and UniProtKB. Knowledge graphs are used today to power structured search results in web search engines (providing the "right-hand entity pane" for many search engine queries) and to answer questions in voice assistants.

KGs may be queried using a variety of tools, including graph database systems, but for most projects today the KG's most salient quality is its data quality, not the query characteristics it might imply. Knowledge graphs generally comprise:

- A set of entities, each with a unique identifier. For example, Q76 in Wikidata refers to the 44th President Barack Obama.
- For each entity, a set of property/value pairs. Properties also have unique identifiers. For example, the *date of birth* property in Wikidata is P569. Values can be either constants ("4 August 1961") or references to other entities in the KG.
- A set of types that organize the entities. In some knowledge graphs, type information is treated as a special case, and in others it takes the form of a particular property. For example, in Wikidata the property P31 ("instance of") communicates type-like information. In Wikidata, entity Q76 ("Barack Obama") has a P31 ("instance of") value of Q5 ("human").

The number of entities and properties varies across KGs. As of this writing, the Wikidata KG has 55,177,813 unique entities, and about 3,000 unique properties [23].

KGs have been used for question answering and search applications, and as inputs to machine learning pipelines. We are unaware of any previous use of KGs in programming settings.

2.2 Simple Example

Let us return to the simple program from Figure 1 and discuss the three steps in more detail: writing the *knowledge program*, running a *target module*, and executing the *data compiler*.

2.2.1 Writing the Knowledge Program. All of the primitives in this program — the action itself, and the three parameters — are drawn from knowledge resources that the KGP system has access to. The entities are drawn from one of the widely-available knowledge graphs; a good library of actions does not yet exist, but must be

---

[1]The full animation, plus an example of the user writing the tiny knowledge program, can be viewed at http://web.eecs.umich.edu/~michjc/fly_kgp.mp4

[2]In this case, the Wikidata KG: entities Q76 (Barack Obama), Q61 (Washington, D.C.), and Q90 (Paris, France)
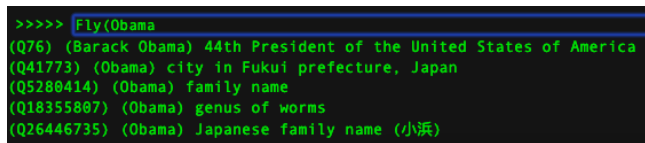
Figure 2: A user can write knowledge programs that initially contain ambiguous textual references to entities in a backing knowledge graph. The KGP programming environment uses autosuggest to force users to disambiguate the reference prior to execution.

Figure 3: A frame from a `2DIllustration` rendering of the same `Fly(Obama | Washington DC | Paris)` knowledge program, rendered without any human attention at the data compilation stage.

prevStep	curStep	FrameNum	Q76-lat	Q76-lon	...
INIT	1	0	38.90	-77.04	...
INIT	1	1	...	...	...
...	...	...	...	...	...
INIT	1	88	...	...	...
INIT	1	89	48.86	2.35	...

constructed and published for this system. The user can simply assume that in the vast majority of cases, the KGP system knows about the actions and entities she wants to include; she does not have to look them up prior to writing code.

The example here is chosen with simple and well-known entities, but KGP systems can easily include access to more specialized entity sources, such as organization-specific LDAP directories, or product inventories.

The KGP interface maps user-entered strings to candidate entities as the user types, and requires strings to be mapped to unambiguous KG entities before the program can be completed. Figure 2 shows the programmer console interface that forces this disambiguation. Of course, textual reference disambiguation in the general case is an extremely challenging problem, albeit one that has enjoyed a substantial amount of research attention. For the time being, we believe a combination of interactivity and user judgment working in a "human-in-the-loop" fashion will allow us to sidestep most of the hard technical issues that usually surround disambiguation.

This program consists of a single action, but in general knowledge programs can comprise multiple actions, executed in multiple steps. Executing a step in the knowledge program appends a row to a special table called `ProgramState`; this row includes the step number, as well as any facts that are yielded by executing the step. There is a namespace of possible facts shared by all KGP actions. Running the knowledge program from Figure 1 yields the simple `ProgramState` table seen below:

Step	Facts
INIT	At(Q76, Q61);
1	At(Q76, Q90); Fly(Q76)

2.2.2 Running the Target Module(s). After the knowledge program has run and has yielded a `ProgramState` table, the chosen target module(s) generates a *data scaffold*. The data scaffold is a table where each row describes a single-action transition in the `ProgramState table`. The scaffold contains as many rows as the target module decides to add, and has columns that are particular to the combination of `ProgramState` facts and the *target module(s)*. The target module may not be able to fully populate the table with useful values, but once it is populated, generating the user's desired software artifact is straightforward. An example for a 90-frame `2DMap` animation of our program is seen below:

Imagine the user instead chose to use the `2DIllustration` target module, which generates more figurative "educational cartoon" to illustrate Obama's trip. A frame from such an artifact is shown in Figure 3[3]. The data scaffold that arises from the `2DIllustration` target module is potentially much more challenging to populate than the one described above; for example, we need to choose images to represent each location, as well as how to place them relative to each other onscreen.
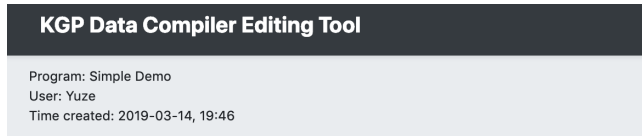
Note that while `2DMap` and `2DIllustration` are well-matched to our `Fly()` action, they are general-purpose modules. (For example, we are experimenting with rendering TCP/IP-like code using `2DIllustration`.) They are capable of rendering any knowledge program, though of course some programs would be poor choices for these targets. Like the templates that accompany PowerPoint decks or WordPress content, certain target modules will be better suited to certain kinds of knowledge program.

2.2.3 Executing the Data Compiler. The data compiler's role is to fill in the data scaffold generated in the previous step. This task is easy in the case of the `2DMap` scaffold: using default KG values for city locations — and simply interpolating Obama's position over multiple frames — will yield a good-looking result.

It is more difficult in the `2DIllustration` case. The compiler has identified fairly high-quality images to identify the relevant entities (Obama and the two cities), but the initial animation is crude: Obama simply slides from left to right. Figure 4 shows an example screenshot of a crowdsourcing data entry page that the data compiler can generate; the crowdsourcing worker can modify the city's relative position and size onscreen. A better data scaffold, with an improved data compiler, could provide animation details, such as ease-in and ease-out or an animated body for Obama.

The data quality burden placed on the data compiler may initially seem unreasonable, but consider the massive amount of implicit evidence that the programmer has provided. Thanks to the knowledge graph, we know that entity Q76 (Barack Obama) is a human being,

---

[3]The full animation can be seen at http://web.eecs.umich.edu/~michjc/obama_2d_illustration.mov

KGP Data Compiler Editing Tool

Program: Simple Demo
User: Yuze
Time created: 2019-03-14, 19:46

Entity: Q90 (Paris)

		Q90 (Paris)
Image:		Q61 (Washington, D.C.) Q76 (Barack Obama)
	Choose File No file chosen	
x-pos	100 from top	
y-pos	100 from right	
Width	200	
Height	200	
Submit		

Figure 4: A web app interface that is executed as part of the data compiler. It asks a crowdworker to supply useful data values that are then integrated into the target artifact.

and so the `Fly()` action in the 2DMap scaffold should be rendered using an airplane rather than with explicit wings. We also know that if any other human has been previously illustrated with the `Fly()` action (say, entity Q6279, Joe Biden), we can likely reuse those filled-in values for the data scaffold. Finally, we know from KG information that the two cities are distant from each other, and so their bitmaps should be drawn far apart on the screen in 2DIllustration. KGP's entity-centric programming model is designed to maximize this implicit evidence about desired data scaffold values.

Although this example is simple, it shows the exciting possibilities of knowledge graph programming: programs that are extremely simple and easy to write, nonetheless made into high-quality artifacts using modern advances in data quality methods. The last few years have shown that the "data quality improvement curve" is a good one for KGP to exploit. But in case those methods are insufficient, the programmer is at least left with a partially-complete artifact that can be improved with further direct effort.

2.2.4 Other Programs. Finally, although we only described a one-step, one-action, straight-line program in this example, the general KGP architecture applies equally well to multi-step, multi-action programs, and to non-straight-line programs. For example, imagine a program that describes the actions a new employee must take during the first day of work. She must:

(1) Fill in a retirement savings election form
(2) Submit it to a company official who checks it for mistakes
(3) If there are no mistakes, the form is electronically submitted to a financial company, which sends the employee a receipt
(4) If there are mistakes, the official returns the form to the employee to be corrected

The logic here is simple, and could use basic actions like `EnterInformation()`, `Transmit()`, `Check()`, and `Return()`. The
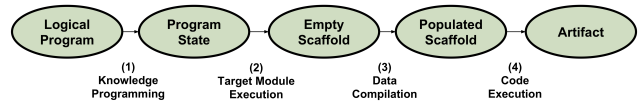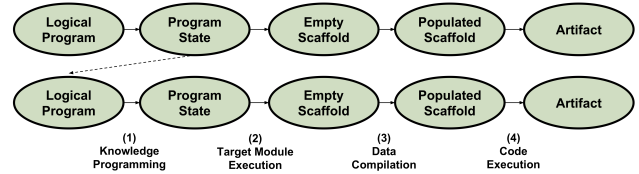

Figure 5: The KGP execution pipeline.

Figure 6: Incremental version of KGP execution.

entities — the employee, the retirement form, the company official, and the financial company — all likely exist in either a public knowledge graph or in the company's LDAP server. And once this short knowledge program is written, it can be compiled into a range of useful software artifacts: an informative video during employee on-boarding, a textual TODO list for the company official, and a series of web apps that enforce the document and receipt handling.

We describe the multi-action execution model below.

3 EXECUTION AND DATA MODEL

The above example illustrated a simple "batch" mode in which the entire logical KGP program can be run to completion — and the corresponding `ProgramState` table constructed — prior to any target module or data compiler execution. Figure 5 summarizes this overall execution flow of writing and running a KGP program. However, in cases where the knowledge program has conditional execution, or requires human input for execution, straight-line batch execution is not possible. Instead, the incremental execution scenario interleaves knowledge program execution, target modules' scaffold generation, and data compilation.

Figure 6 shows our incremental execution model. Because rows in the data scaffold always refer to a single pair of states in `ProgramState`, such as the transition from INIT to step 1, we do not have to wait until the end of knowledge program execution — that is, the full completion of the `ProgramState` table — in order to invoke the target module.

Instead, we repeatedly:

(1) Run a single step from the knowledge program, yielding a single new row in the `ProgramState` table
(2) Run the user's chosen target module(s), yielding zero or more rows in the empty data scaffold, all with the same `prevStep`, `curStep` pair.
(3) Apply the data compiler to these new rows
(4) Invoke the artifact using the newly-populated data

4 THE DATA COMPILER

The data compiler is the most technically challenging part of our system and its success determines whether the artifacts produced

Human Contributions
Direct programmer specification
Reusing previous values from same user
Log analysis (reusing values from other users)
Crowdsourcing
Implicit Relationships
Entities with same KG type likely share values
Entities in same action slot position likely share values
Entities in similar actions (e.g. Walk() and Run()) share values
Entities with similar target module roles likely share values
External Resources
Different KGs (e.g. Wikidata, DBpedia, Open Linked Data)
External structured DBs
Information extraction from external text
Data fusion and integration

Table 1: Sources of data compiler evidence

are good or bad. We aim to build a data compiler using two basic design principles:

First, the data compiler can use any data quality mechanism it chooses, including information extraction, data integration, crowdsourcing, and so on. For example, consider a case where a user writes a program that shows the mountains conquered by a famous climber: Mount Everest, Mount Kilimanjaro, and so on. It would be reasonable for the 2DIllustration target module's data scaffold to require height information, so the mountains can be properly rendered on-screen. If needed, the data compiler can obtain this information by building an information extractor and applying it to general web text. Alternatively, it could try to integrate information drawn from an external geographic database, such as MONDIAL [20].

Second, the system and programmer provides the data compiler with multiple overlapping sources of weak evidence about the desired data values. Those sources of evidence include usage logs, crowd annotations, data constraints, standardized datasets (such as the above MONDIAL database), and others. The evidence can be divided into 3 major parts: Human Contributions, Implicit Relationships and External Resources. Some possible examples of such evidence can be seen in Table 1.

Human Contributions represent the portion of usable data from the subjective effort of human beings. The most significant one in this category is crowdsourcing. To take a simple example, a company can ask crowd workers to fill in missing property values for entities in the knowledge program.

Implicit Relationships represent the source of evidence coming from abundant correlation among properties that can be used to impute missing values or improve existing values. Take the 2DIllustration rendering of the Fly(Obama | Washington DC | Paris) as an example, Washington DC and Paris are both instances of "Q515: city" in the Wikidata KG, while Obama is an instance of "Q5: human". The KGP system can infer that entities of the same type are likely to have the same value for some properties.

Here, the system infers that the two cities should be rendered in the same image size for the 2DIllustration target module. Therefore, Washington DC and Paris have the same size while Obama does not in Figure 3.

External Resources play an important role in the KGP system. KGs are the starting point of of the KGP system and usually contain lots of useful data. Other than KGs, the data compiler should also be able to grab information from LDAP servers, standard JSON data feeds, standard taxonomies and ontologies, and even unstructured text processed by information extraction tools.

5 SYSTEM PROTOTYPE

Our current KGP prototype is written with its core execution engine in Python, and user interface elements written using HTML and JavaScript. It uses Wikidata as its backing knowledge graph, plus a small library of actions, including Walk(), Fly(), Transmit(), and others. It currently implements two useful target modules (2DMap and 2DIllustration), as well as a couple of modules used for program debugging. We have implemented both straight-line and conditional event-driven execution models.

In addition to writing KGP programs in a file, the user can write them using an interactive console, as pictured in Figure 2. This console allows the user to write programs that are reflected immediately onscreen. The user also has access to a set of "metacommands" that control the current set of target modules, whether the KGP program should be restarted, etc.

The system's data compiler currently draws on values from three sources: the background KG, estimates from the target modules, and crowdsourced inputs. The crowdsourcing interface can be seen in Figure 4. A user (whether the programmer or not) can use this tool to provide new scaffold values, and thereby influence the target software artifact.

6 PRELIMINARY EXPERIMENTS

Our current prototype has a working interface and execution model, but we are still building out a rich set of actions, target modules, and data compilation techniques. As a result, our preliminary experiments focus on the knowledge programmer experience, in particular: can the console system easily disambiguate programmer entity references without too much programmer effort? If this disambiguation process is extremely burdensome for the programmer, then users will abandon the KGP framework before we even get to tackle more ambitious programs or the technically-challenging parts of the data compiler step.

We examined the interactive console's diambiguation system on two real-world data sets: (1) 100 city names, sampled from 385 of the largest cities in the United States; and (2) 100 person names, sampled from a list of 11,341 prominent people in world history [24].

We ran the KGP console entity-disambiguation tool on all the names from our test sets. We also ran the search for people with only their first names to test how the system reacts with ambiguous input. The console uses information retrieval-style methods to rank all entities in the backing knowledge graph, based on the user's input string; it shows the top-10 entity guesses.

We measured the percentage of console suggestions that have the correct Wikidata entity as the number-1 hit, the percentage of

	% top-1 hit	% top-10 hit	Avg rank
Cities	84.14	98.78	1.15
People (full name)	91.9	92.9	1.01
People (first name)	9.1	25.3	4.36

Table 2: Experimental results

console suggestions that have the correct Wikidata entity in the top-10, and the average rank of the correct entities of those that are in the top-10.

The results are shown in Table 2. We see that the system is very effective when the user provides a full text input, even in the challenging People category[4]. This is especially notable when considering that many individuals in the test set would be considered obscure for many users. Partial input is generally not effective (consider using the first name John) but providing more textual evidence is likely not a challenge for the programmer, and is akin to simply providing a longer web search string.

7 RELATED WORK

There are several broad areas of related work.

7.1 Program Synthesis

There has been a substantial amount of recent interest in *program synthesis*, the task of automatically writing source code that yields a program that matches a user-provided specification[4, 17]. Although most of this work has taken place in the programming languages community, there has also been some activities in data management. The most successful artifact to come from program synthesis work is likely the FlashFill [11] feature in Microsoft Excel, which synthesizes small data extraction programs in response to user-provided examples. Program synthesis shares with knowledge graph programming a goal of reducing the amount of human work necessary to accomplish a programming task.

Program synthesis efforts vary widely in the languages they target, the format of the user's specification, and the kinds of programs they generate. However, they generally share an emphasis on (1) generating short code snippets (2) using logical solver mechanisms (3) yielding entirely correct programs. In contrast, knowledge graph programming (1) does not generate much or any traditional source code, (2) employs data quality mechanisms from the data management, machine learning, and crowdsourcing communities [15], with an aim of (3) creating programs that can be "better" or "worse" but are always correct.

7.2 Domain-Specific Languages

Research in *domain-specific languages* (DSLs) attempts to build programming languages that are targeted exclusively for particular program types, enabling both succinct source code and deeper semantic information that can be used at optimization time [2, 22]. Each new DSL requires a new language and runtime infrastructure, though the burden can be reduced by using a DSL toolkit.

---

[4]The Wikidata KG contained 4,378,742 human-typed entities at the time of this experiment

Knowledge graph programs are general-purpose; there is no need for a new language with new programs for every new domain. However, there is some new infrastructure required for each output type, in the form of target modules. Further, the performance of the data compiler will potentially differ for each new module.

7.3 ML-Assisted Programming

There has been a very recent set of papers that use machine learning methods to assist in writing programs [1, 8–10]. One well-known effort is the DeepCoder project [1], which uses a trained artifact to guide exploration of a program synthesis space. Another is Kraska, *et al.*'s work to learn novel database index structures using deep neural network methods [14]. This area of work is very new and heterogeneous, making it difficult to summarize right now. Nonetheless, knowledge graph programming is distinctive compared to most of these efforts in (1) its use of a clear programming language and model that non-ML experts can write, and (2) its ability to use off-the-shelf data quality tools during data compilation (as opposed to more bespoke neural networks or other architectures).

8 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced *knowledge graph programming*, a framework that aims to dramatically increase programmer productivity by exploiting recent advances in data quality methods. This work is extremely preliminary, and there is a huge amount of future work. Two especially important areas include (a) the data compiler, and how effectively it can fill in missing values at a certain human effort cost, and (b) the expressiveness of future programs.

REFERENCES

[1] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989
[2] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011.* 89–100. https://doi.org/10.1109/PACT.2011.15
[3] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control.* ACM, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515
[4] Robert A. Cochran, Loris D'Antoni, Benjamin Livshits, David Molnar, and Margus Veanes. 2015. Program Boosting: Program Synthesis via Crowd-Sourcing. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, NY, USA, 677–688. https://doi.org/10.1145/2676726.2676973
[5] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandecic. 2014. Introducing Wikidata to the Linked Data Web. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I.* 50–65. https://doi.org/10.1007/978-3-319-11964-9_4
[6] Oren Etzioni, Michael J. Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. 2005. Unsupervised named-entity extraction from the Web: An experimental study. *Artif. Intell.* 165, 1 (2005), 91–134. https://doi.org/10.1016/j.artint.2005.03.001
[7] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 420–435. https://doi.org/10.1145/3192366.3192382
[8] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). arXiv:1608.04428 http://arxiv.org/abs/1608.04428
[9] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. (2014). arXiv:arXiv:1410.5401v2 [cs.NE]

[10] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476. https://doi.org/10.1038/nature20101

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 317–330. https://doi.org/10.1145/1926385.1926423

[12] Sumit Gulwani, Susmit Jha, and Ashish Tiwari. 2011. Synthesis of loop-free programs. In *PLDI 2011.*

[13] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* 215–224. https://doi.org/10.1145/1806799.1806833

[14] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* 489–504. https://doi.org/10.1145/3183713.3196909

[15] Sang Won Lee, Yujin Zhang, Isabelle Wong, Yiwei Yang, Stephanie D. OâĂŹKeefe, and Walter S. Lasecki. 2017. SketchExpress: Remixing Animations for More Effective Crowd-Powered Prototyping of Interactive Interfaces. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (2017), 817–828. https://doi.org/10.1145/3126594.3126595

[16] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195. https://doi.org/10.3233/SW-140134

[17] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke ZeÂŁlemoyer, and Michael D. Ernst. 2017. *Program Synthesis from Natural Language Using Recurrent Neural Networks.* Technical Report. Seattle, WA, USA.

[18] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. 2015. YAGO3: A Knowledge Base from Multilingual Wikipedias. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings.* http://cidrdb.org/cidr2015/Papers/CIDR15_Paper1.pdf

[19] Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14 (March 1971), 151–165. https://doi.org/10.1145/362566.362568

[20] Wolfgang May. 1999. *Information Extraction and Integration with Florid: The Mondial Case Study.* Technical Report 131. Universität Freiburg, Institut für Informatik. Available from http://dbis.informatik.uni-goettingen.de/Mondial.

[21] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* UC Berkeley.

[22] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices* 35 (01 2000), 26–36.

[23] Wikidata. 2018. Wikidata: Statistics. https://www.wikidata.org/wiki/Wikidata:Statistics

[24] Amy Zhao Yu, Shahar Ronen, Kevin Hu, Tiffany Lu, and Cesar Hidalgo. 2016. Pantheon 1.0, A Manually Verified Dataset of Globally Famous Biographies. https://doi.org/10.7910/DVN/28201