Genie: A Generator of Natural Language Semantic Parsers for Virtual Assistant Commands

Giovanni Campagna*
Computer Science Department
Stanford University
Stanford, CA, USA
gcampagn@cs.stanford.edu

Silei Xu*
Computer Science Department
Stanford University
Stanford, CA, USA
silei@cs.stanford.edu

Mehrad Moradshahi
Computer Science Department
Stanford University
Stanford, CA, USA
mehrad@cs.stanford.edu

Richard Socher
Salesforce, Inc.
Palo Alto, CA, USA
rsocher@salesforce.com

Monica S. Lam
Computer Science Department
Stanford University
Stanford, CA, USA
lam@cs.stanford.edu

Abstract

To understand diverse natural language commands, virtual assistants today are trained with numerous labor-intensive, manually annotated sentences. This paper presents a methodology and the Genie toolkit that can handle new compound commands with significantly less manual effort.

We advocate formalizing the capability of virtual assistants with a *Virtual Assistant Programming Language (VAPL)* and using a neural semantic parser to translate natural language into VAPL code. Genie needs only a small realistic set of input sentences for validating the neural model. Developers write templates to synthesize data; Genie uses crowdsourced paraphrases and data augmentation, along with the synthesized data, to train a semantic parser.

We also propose design principles that make VAPL languages amenable to natural language translation. We apply these principles to revise ThingTalk, the language used by the Almond virtual assistant. We use Genie to build the first semantic parser that can support compound virtual assistants commands with unquoted free-form parameters. Genie achieves a 62% accuracy on realistic user inputs. We demonstrate Genie's generality by showing a 19% and 31% improvement over the previous state of the art on a music skill, aggregate functions, and access control.

*Equal contribution

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314594

CCS Concepts • **Human-centered computing** → **Personal digital assistants**; • **Computing methodologies** → *Natural language processing*; • **Software and its engineering** → *Context specific languages.*

Keywords virtual assistants, semantic parsing, training data generation, data augmentation, data engineering

ACM Reference Format:
Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: A Generator of Natural Language Semantic Parsers for Virtual Assistant Commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314594

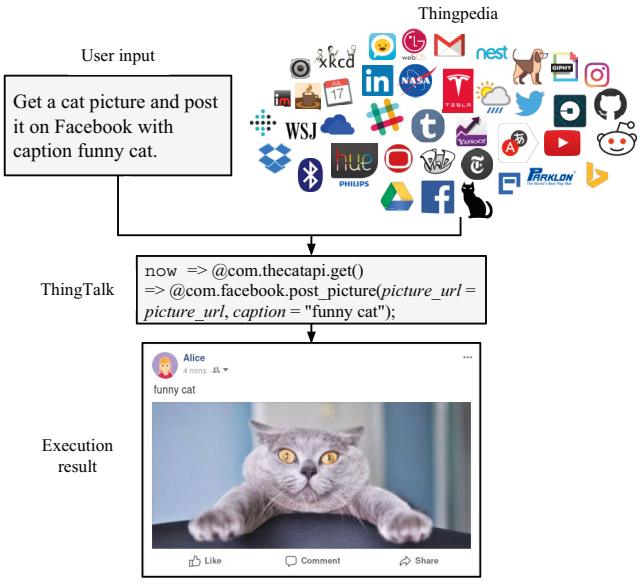

Figure 1. An example of translating and executing compound virtual assistant commands.

1 Introduction

Personal virtual assistants provide users with a natural language interface to a wide variety of web services and IoT devices. Not only must they understand primitive commands across many domains, but they must also understand the composition of these commands to perform entire tasks. State-of-the-art virtual assistants are based on *semantic parsing*, a machine learning algorithm that converts natural language to a semantic representation in a formal language. The breadth of the virtual assistant interface makes it particularly challenging to design the semantic representation. Furthermore, there is no existing corpus of natural language commands to train the neural model for new capabilities. This paper advocates using a *Virtual Assistant Programming Language* (VAPL) to capture the formal semantics of the virtual assistant capability. We also present Genie, a toolkit for creating a semantic parser for new virtual assistant capabilities that can be used to bootstrap real data acquisition.

1.1 Virtual Assistant Programming Languages

Previous semantic parsing work, including commercial assistants, typically translates natural language into an intermediate representation that matches the semantics of the sentences closely [4, 30, 33, 44, 51]. For example, the Alexa Meaning Representation Language [30, 44] is associated with a closed ontology of 20 domains, each manually tuned for accuracy. Semantically equivalent sentences have different representations, requiring complex and expensive manual annotation by experts, who must know the details of the formalism and associated ontology. The ontology also limits the scope of the available commands, as every parameter must be an entity in the ontology (a person, a location, etc.) and cannot be free-form text.

Our approach is to represent the capability of the virtual assistant fully and formally as a VAPL; we use a deep-learning semantic parser to translate natural language into VAPL code, which can directly be executed by the assistant. Thus, the assistant's full capability is exposed to the neural network, eliminating the need and inefficiency of an intermediate representation. The VAPL code can also be converted back into a canonical natural language sentence to confirm the program before execution. Furthermore, new capabilities can be supported by extending the VAPL.

The ThingTalk language designed for the open-source Almond virtual assistant is an example of a VAPL [8]. ThingTalk has one construct which has three clauses: when some event happens, get some data, and perform some action, each of which can be predicated. This construct combines primitives from the extensible runtime skill library, Thingpedia, currently consisting of over 250 APIs to Internet services and IoT devices. Despite its lean syntax, ThingTalk is expressive. It is a superset of what can be expressed with IFTTT, which has crowdsourced more than 250,000 unique compound commands [56]. Fig. 1 shows how a natural-language sentence can be translated into a ThingTalk program, using the services in Thingpedia.

However, the original ThingTalk was not amenable to natural language translation, and no usable semantic parser has been developed. In attempting to create an effective semantic parser for ThingTalk, we discovered important design principles for VAPL, such as matching the non-developers' mental model and keeping the semantics of components orthogonal. Also, VAPL programs must have a (unique) canonical form so the result of the neural network can be checked for correctness easily. We applied these principles to overhaul and extend the design of ThingTalk. Unless noted otherwise, we use ThingTalk to refer to the new design in the rest of the paper.

1.2 Training Data Acquisition

Virtual assistant development is labor-intensive, with Alexa boasting a workforce of 10,000 employees [36]. Obtaining training data for the semantic parser is one of the challenging tasks. How do we get training data before deployment? How can we reduce the cost of annotating usage data? Wang et al. [57] propose a solution to acquire training data for the task of question answering over simple domains. They use a syntax-driven approach to create a canonical sentence for each formal program, ask crowdsourced workers to paraphrase canonical sentences to make them more natural, then use the paraphrases to train a machine learning model that can match input sentences against possible canonical sentences. Wang et al.'s approach designs each domain ontology individually, and each domain is small enough that all possible logical forms can be enumerated up to a certain depth.

This approach was used in the original ThingTalk semantic parser and has been shown to be inadequate [8]. It is infeasible to collect paraphrases for all the sentences supported by a VAPL language. Virtual assistants have powerful constructs to connect many diverse domains, and their capability scales superlinearly with the addition of APIs. Even with our small Thingpedia, ThingTalk supports hundreds of thousands of distinct programs. Also, it is not possible to generate just one canonical natural language that can be understood across different domains. Crowdworkers often paraphrase sentences incorrectly or just make minor modifications to original sentences.

Our approach is to design a *NL-template language* to help developers data-engineer a good training set. This language lets developers capture common ways in which VAPL programs are expressed in natural language. The NL-templates are used to *synthesize* pairs of natural language sentences and their corresponding VAPL code. A sample of such sentences is paraphrased by crowdsource workers to make them more natural. The paraphrases further inform more useful templates, which in turn derives more diverse sentences

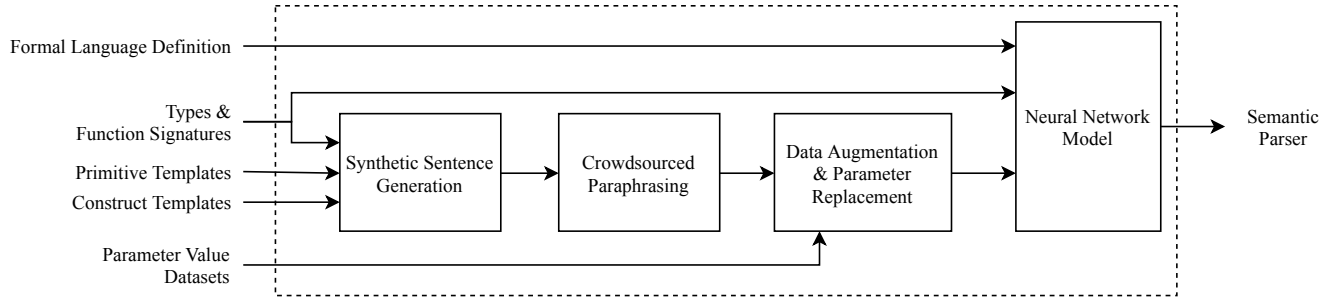

Figure 2. Overview of the Genie Semantic Parser Generator

for paraphrasing. This iterative process increases the cost-effectiveness of paraphrasing.

Whereas the traditional approach is only to train with paraphrase data, we are the first to add *synthesized* sentences to the training set. It is infeasible to exhaustively paraphrase all possible VAPL programs. The large set of synthesized, albeit clunky, natural language commands are useful to teach the neural model compositionality.

1.3 Contributions

The results of this paper include the following contributions:

1. We present the design principles for VAPLs that improve the success of semantic parsing. We have applied those principles to ThingTalk, and created a semantic parser that achieves a 62% accuracy on data that reflects realistic Almond usage, and 63% on manually annotated IFTTT data. Our work is the first semantic parser for a VAPL that is extensible and supports free-form text parameters.
2. A novel NL-template language that lets developers direct the synthesis of training data for semantic parsers of VAPL languages.
3. The first toolkit, Genie, that can generate semantic parsers for VAPL languages with the help of crowdsourced workers. As shown in Fig. 2, Genie accepts a VAPL language and a set of NL-templates. Genie adds synthesized data to paraphrased data in training, expands parameters, and pretrains a language model. Our neural model achieves an improvement in accuracy of 17% on IFTTT and 15% on realistic inputs, compared to training with paraphrase data alone.
4. Demonstration of extensibility by using Genie to generate effective semantic parsers for a Spotify skill, access control, and aggregate functions.

1.4 Paper Organization

The organization of the paper is as follows. Section 2 describes the VAPL principles, using ThingTalk as an example. Section 3 introduces the Genie data acquisition pipeline, and Section 4 describe the semantic parsing model used by Genie. We present experimental results on understanding ThingTalk commands in Section 5, and additional languages and case studies in Section 6. We present related work in Section 7 and conclusion in Section 8.

2 Principles of VAPL Design

Here we discuss the design principles of Virtual Assistant Programming Languages (VAPL) to make it amenable to natural language translation, using ThinkTalk as an example. These design principles can be summarized as (1) strong fine-grained typing to guide parsing and dataset generation, (2) a skill library with semantically orthogonal components designed to reduce ambiguity, (3) matching the user's mental model to simplify the translation, and (4) canonicalization of VAPL programs.

2.1 Strong, Static Typing

To improve the compositionality of the semantic parser, VAPLs should be statically typed. They should include fine-grain domain-specific types and support definitions of custom types. The ThinkTalk type system includes the standard types: strings, numbers, booleans, and enumerated types. It also has native support for common object types used in IoT devices and web services. Developers can also provide custom *entity types*, which are internally represented as opaque identifiers but can be recalled by name in natural language. Arrays are the only compound type supported.

To allow translation from natural language without contextual information, the VAPL also needs a rich language for constants. For example, in ThingTalk, measures can be represented with any legal unit, and can be composed additively (as in "6 feet 3 inches", which is translated to 6ft + 3in); this is necessary because a neural semantic parser cannot perform arithmetic to normalize the unit during the translation.

Arguments such as numbers, dates and times, in the input sentence are identified and normalized using a rule-based algorithm [15]; they are replaced as named constants of the form "NUMBER_0", "DATE_1", etc. String and named entity parameters instead are represented using multiple tokens, one for each word in the string or entity name; this allows the words to be copied from the input sentence individually.

Class *c*:	`class` @*cn* [`extends` @*cn*]* { [*qd*]* [*ad*]* }
Class name *cn*:	identifier
Query declaration *qd*:	`monitorable? list? query` *fn*([*pd*]*);
Action declaration *ad*:	`action` *fn*([*pd*]*);
Function name *fn*:	identifier
Parameter declaration *pd*:	[`in req` | `in opt` | `out`] *pn* : *t*
Parameter name *pn*:	identifier
Parameter type *t*:	`String` | `Number` | `Boolean` | `Enum`([*v*]+) | `Measure`(*u*) | `Date` | `PathName` | `Entity`(*et*) | `Array`(*t*) | ...
Value *v*:	literal
Unit *u*:	bytes | KB | ms | s | m | km | ...
Entity type *et*:	literal

Figure 3. The formal grammar of classes in the skill library.

Named entities are normalized with a knowledge base lookup after parsing.

2.2 Skill Library Design

The skill library defines the virtual assistant's knowledge of the Internet services and IoTs. As such, the design of the representation as well as how information is organized are very important. In the following, we describe a high-level change we made to the original ThingTalk, then present the new syntax of classes and the design rationale.

Orthogonality in Function Types. In the original Thingpedia library, there are three kinds of functions: *triggers*, *retrievals*, and *actions* [8]. Triggers are callbacks or polling functions that return results upon the arrival of some event, retrievals return results, actions create side effects. Unfortunately, the semantics of callbacks and queries are not easily discernible by consumers. It is hard for users to understand that "when Trump tweets" and "get a Trump tweet" refer to two different functions. Furthermore, when a device supports only a trigger and not a retrieval, or vice versa, it is not apparent to the user which is allowed. The inconsistency in functionality makes getting correct training and evaluation data problematic.

In the new ThingTalk, we collapse the distinction between triggers and retrievals into one class: queries, which can be monitored as an event or used to get data. The runtime ensures that any supported retrieval can be monitored as events, and vice versa. Not only does this provide more functionality, it also makes the language more regular and hence simpler for the users, crowdsourced paraphrase providers, and the neural model.

Skill Definition Syntax. For modularity, every VAPL should include a skill library, a set of classes representing the domains and operations supported. In ThingTalk, skills are IoT devices or web services. The formal grammar of ThingTalk classes is shown in Fig. 3.

Classes have functions, belonging to one of two kinds: *query functions* retrieve data and have no side-effects; *action functions* have side-effects but do not return data.

```
class @com.dropbox {
monitorable query get_space_usage(out used_space : Measure(byte),
                                  out total_space : Measure(byte));
monitorable list query list_folder(in req folder_name : PathName,
                                   in opt order_by : Enum,
                                   out file_name : PathName,
                                   out is_folder : Boolean,
                                   out modified_time : Date,
                                   out file_size : Measure(byte),
                                   out full_path : PathName);
query open(in req file_name : PathName,
           out download_url : URL);
action move(in req old_name : PathName,
            in req new_name : PathName);
... }
```

Figure 4. The Dropbox class in the ThingTalk skill library.

A query function can be *monitorable*, which means that the result returned can be monitored for changes. The result can be polled, or the query function supports push notifications. Queries that cannot be monitored include those that change constantly, such as the retrieval of a random cat picture used in Fig. 1. A query can return a single result or return a *list* of results.

The function signature includes the class name, the function name, the type of the function, all the parameters and their types. Data are passed in and out of the functions through named parameters, which can be required or optional. Action methods have only input parameters, while query methods have both input and output parameters.

An example of a class, for the Dropbox service, is shown in Fig. 4. It defines three queries: "get_space_usage", "list_folder", and "open", and an action "move". The first two queries are monitorable; the third returns a randomized download link for each invocation so it is not monitorable.

2.3 Constructs Matching the User's Mental Model

VAPLs should be designed to reflect how users typically specify commands. It is more natural for users to think about data with certain characteristics, rather than execution paths. For this reason, ThingTalk is data focused and not control-flow focused. ThingTalk has a single construct:

$$s \Rightarrow q \Rightarrow a$$

The *stream* clause, *s*, specifies the evaluation of the program as a continuous stream of events. The optional *query* clause, *q*, specifies what data should be retrieved when the events occur. The *action* clause, *a*, specifies what the program should do. The formal grammar is shown in Fig. 5.

An example illustrating parameter passing is shown in Fig. 1. The use of filters is demonstrated with the following example that automatically retweets the tweets from PLDI:

```
monitor(@com.twitter.timeline() filter author = @PLDI)
⇒ @com.twitter.retweet(tweet_id = tweet_id)
```

The program uses the *author* output parameter of the function @com.twitter.timeline() to filter the set of tweets, and

Program π:	s [⇒ q]? ⇒ a;
Stream s:	`now` | `attimer time` = v | `timer base` = v `interval` = v | `monitor` q | `edge` s `on` p
Query q:	f [ip = v]* [ip = op]* | q `filter` p | q `join` q [`on` [ip = op]+]?
Action a:	f [ip = v]* [ip = op]* | `notify`
Function f:	@cn.fn
Class name cn:	identifier
Function name fn:	identifier
Input parameter ip:	pn : t
Output parameter op:	pn : t
Parameter name pn:	identifier
Parameter type t:	the parameter type in Fig. 3
Predicate p:	`true` | `false` | `!`p | p `&&` p | p `||` p | op *operator* v | f [ip = v]* { p }
Value v:	literal | `enum` : identifier
Operator *operator*:	`==` | `>` | `<` | `contains` | `substr` | `starts_with` | `ends_with` | ...

Figure 5. The formal grammar of ThingTalk.

passes the *tweet_id* output parameter to the input parameter with the same name of @com.twitter.retweet().

Queries and Actions. To match the user's mental model, ThingTalk uses implicit, rather than explicit, looping constructs. The user is given the abstraction that they are describing operations on scalars. In reality, queries always return a list of results; functions returning a single result are converted to return singleton lists. These lists are implicitly traversed; each result can be used as an input parameter in a subsequent function invocation.

The result of queries can be optionally filtered with a boolean predicate using equality, comparions, string and array containment operators, as well as predicated query functions. For example, the following retrieves only "emails from Alice":

```
now ⇒ (@com.gmail.inbox()) filter sender = "Alice" ⇒ notify
```

We introduce the *join* operator for queries in ThingTalk to support multiple retrievals in a program. When joining two queries, parameters can be passed between the queries, and the results are the cross product of the respective queries. A program's action can be either the builtin `notify`, which presents the result to the user, or an action function defined in the library.

For example, the following "translates the title of New York Times articles":

```
now ⇒ @com.nytimes.get_front_page() join
      @com.yandex.translate() on text = title ⇒ notify
```

Streams. Streams are a new concept we introduce to ThingTalk. They generalize the trigger concept in the original language to enable reacting to arbitrary changes in the data accessible by the virtual assistant. A stream can be (1) the degenerate stream "now", which triggers the program once immediately, (2) a timer, or (3) a `monitor` of a query, which triggers whenever the query result changes. Any query that uses monitorable functions can be monitored, including queries that use joins or filters.

We introduce a stream operator, *edge filter*, to watch for changes in a stream. It triggers whenever a boolean predicate on the values monitored transition from false to true; the predicate is assumed to be previously false for the first value in a stream. The program below notifies the users each time the temperature drops below the 60 degrees Fahrenheit theshold.

```
edge (monitor @weather.current()) on temperature < 60F
⇒ notify;
```

The edge filter operator allows us to convert all previous trigger functions to the new language without losing functionality.

Input and Output Parameters. To aid translation from natural language, ThingTalk uses keyword parameters rather than the more conventional positional parameters. With keyword parameters, the semantic parser needs to learn just the partial signature of the functions, and not even the length of the signature. We annotate each parameter with its type, with the goal to help the model distinguish between parameters with the same name and to unify parameters by type. For readability, type annotations are omitted from the examples in this paper. To increase compositionality, we encourage developers to use the same naming conventions so the same parameter names are used for similar purposes.

When an output of a function is passed into another as input, the former parameter name is simply assigned to the latter. For example, in Fig. 1, the output parameter *picture_url* of @com.thecatapi.get() is assigned to the input parameter of @com.facebook.post_picture() with the same name. This design avoids introducing new variables in the program so the semantic parser only needs to learn each function's parameter names. If output parameters in two functions in a program have the same name, we assume that the name refers to the rightmost instance. Here we consciously trade-off completeness for accuracy in the common case.

2.4 Canonicalization of Programs

As described in Section 1, canonicalization is key to training a neural semantic parser. We use semantic-preserving transformation rules to give ThingTalk programs a canonical form. For example, query joins without parameter passing are a commutative operation, and are canonicalized by ordering the operands lexically. Nested applications of the `filter` operator are canonicalized to a single filter with the `&&` connective. Boolean predicates are simplified to eliminate redundant expressions, converted to *conjunctive normal form* and then canonicalized by sorting the parameters and operators. Each clause is also automatically moved to the left-most function that includes all the output parameters. Input parameters are listed in alphabetical order, which helps

the neural model learn a global order that is the same across all functions.

3 Genie Data Acquisition System

The success of machine learning depends on a high-quality training set; it must represent real, correctly labeled, inputs. To address the difficulty in getting a training set for a new language, Genie gives developers (1) a novel language-based tool to synthesize data, (2) a crowdsourcing framework to collect paraphrases, (3) a large corpus of values for parameters in programs, and (4) a training strategy that combines synthesized and paraphrase data.

3.1 Data Synthesis

As a programming language, ThingTalk may seem to have a small number of components: queries, streams, actions, filters, and parameters. However, it has a library of skills belonging to many domains, each using different terminology. Consider the function "list_folder" in Dropbox, which returns a modified time. If we want to ask for "my Dropbox files that changed this week", we can add the filter *modified_time* > `start_of_week`. An automatically generated sentence would read: "my Dropbox files having modified time after a week ago". Such a sentence is hard for crowdsource workers to understand. If they do not understand it, they cannot paraphrase it.

Similarly, even though ThingTalk has only one construct, there are various ways of expressing it. Here are two common ways to describe event-driven operations: "when it rains, remind me to bring an umbrella", or "remind me to bring an umbrella when it rains". Here are two ways to compose functions: "set my profile at Twitter with my profile at Facebook" or "get my profile from Facebook and use that as a profile at Twitter".

We have created a *NL-template language* so developers can generate variety when synthesizing data. We hypothesize that we can exploit compositionality in natural language to factor data synthesis into primitive templates for skills and construct templates for the language. From a few templates per skill and a few templates per construct component, we hope to generate a representative set of synthesized sentences that are understandable.

Primitive Templates. Genie allows the skill developer to provide a list of *primitive templates*, each of which consists of code using that skill, a natural language utterance describing it, and the natural language grammar category of the utterance. The templates define how the function should be invoked, and how parameters are passed and used. The syntax is as follows:

$$cat \ := \ u \rightarrow \lambda([pn : t]^*) \rightarrow [s \mid q \mid a]$$

This syntax declares that the utterance u, belonging to the grammar category *cat* (*verb phrase*, *noun phrase*, or *when phrase*), maps to stream s, query q or action a. The utterance may include placeholders, prefixed with $, which are used as parameters, *pn* of type t, in the stream, query or action.

Table 1. Examples of developer-supplied primitive templates for the @com.dropbox.list_folder and @com.dropbox.open functions in the Thingpedia library, with their grammar category. NP, WP, and VP refer to *noun phrase*, *when phrase* and *verb phrase*, respectively.

Natural language	Cat.	ThingTalk Code
my Dropbox files	NP	@com.dropbox.list_folder()
my Dropbox files that changed most recently	NP	@com.dropbox.list_folder(*order_by* = modified_time_decreasing)
my Dropbox files that changed this week	NP	@com.dropbox.list_folder(*order_by* = modified_time_decreasing) `filter` *modified_time* > `start_of_week`
files in my Dropbox folder $x	NP	$\lambda(x$: PathName) → @com.dropbox.list_folder(*folder_name* = x)
when I modify a file in Dropbox	WP	`monitor` @com.dropbox.list_folder()
when I create a file in Dropbox	WP	`monitor` @com.dropbox.list_folder() `on new` *file_name*
the download URL of $x	NP	$\lambda(x$: PathName) → @com.dropbox.open(*file_name* = x)
a temporary link to $x	NP	
open $x	VP	
download $x	VP	

Examples of primitive templates for functions @com.dropbox.list_folder and @com.dropbox.open are shown in Table 1. Multiple templates can be provided for the same function, as different combinations of input and output parameters can provide different semantics, and functions can be filtered and monitored as well.

Note that the function @com.dropbox.open is a query, not an action, because it returns a result (the download link); the example shows that utterances for queries can be both noun phrases ("the download URL") and verb phrases ("open"). The ability to map the same program fragment to different grammar categories is new in Genie, and differs from the knowledge base representation previously used by Wang et al. [57]. Other examples of verb phrases for queries are "translate $x" (same as "the translation of $x") and "describe $x" (same as "the description of $x").

While designing primitive templates, it is important to choose grammar categories that compose naturally. The original design of ThingTalk exclusively used verb phrases for queries; we switched to mostly noun phrases as they can be substituted as input parameters (e.g., "a cat picture" can be substituted for parameter $\$x$ in "post $\$x$ on Twitter" and "post $\$x$ on Facebook").

Construct Templates. To combine the primitives into full programs, the language designer also provides a set of *construct templates*, mapping natural language compositional

constructs to formal language operators. A construct template has the form:

$$lhs \ := \ [\text{literal} \mid vn : rhs]^{+} \rightarrow sf$$

which says that a derivation of non-terminal category *lhs* can be constructed by combining the literals and variables *vn* of non-terminal category *rhs*, and then applying the semantic function *sf* to compute the formal language representation. For example, the following two construct templates define the two common ways to express "when - do" commands:

COMMAND := s : WP ',' a : VP → **return** $s \Rightarrow a$;
COMMAND := a : VP s : WP → **return** $s \Rightarrow a$;

Together with the following two primitive templates:

WP := 'when I modify a file in Dropbox' →
    `monitor` @com.dropbox.list_folder()
VP := 'send a Slack message' → @com.slack.send()

Genie would generate the commands "when I modify a file in Dropbox, send a Slack message" and "send a Slack message when I modify a file in Dropbox".

Semantic functions allow developers to write arbitrary code that computes the formal representation of the generated natural language. The following performs typechecking, ensuring that only monitorable queries are monitored.

```
WP := 'when' q : NP 'change' → {
    if q.is_monitorable
        return monitor q
    else
        return ⊥
}
```

For another example, the following checks that the argument is a list, so as to be compatible with the semantics of the verb "enumerate":

```
COMMAND := 'enumerate' q : NP → {
    if q.is_list
        return now ⇒ q ⇒ notify
    else
        return ⊥
}
```

Each template can also optionally be annotated with a boolean flag, which allows the developer to define different subsets of rules for different purposes (such as training or paraphrasing).

Synthesis by Sampling Previous work by Wang et al. [57] recursively enumerates *all* possible derivations, up to a certain depth of the derivation tree. Such an approach is unsuitable for ThingTalk, because the number of derivations grows exponentially with increasing depth and library size. Instead, Genie uses a randomized synthesis algorithm, which considers only a subset of derivations produced by each construct template. The desired size is configurable, and the number of derivations decreases exponentially with increasing depth. The large number of low-depth programs provide breadth, and the relatively smaller number of high-depth programs add variance to the synthesized set and expand the set of recognized programs. Developers using Genie can control the sampling strategy by splitting or combining construct templates, using intermediate derivations. For example, the following template:

NP := q : NP 'having' p : PRED → **return** q `filter` p
COMMAND := 'get' q : NP 'and then' a : VP
    → **return** `now` ⇒ q ⇒ a

can be combined in a single template that omits the intermediate noun phrase derivation:

COMMAND := 'get' q : NP 'having' p : PRED 'and then' a : VP
    → **return** `now` ⇒ q `filter` p ⇒ a

(PRED is the grammar category of boolean predicates.) The combined template has lower depth, so more sentences would be sampled from it. Conversely, if a single template is split into two or more templates, the number of sentences using the construct becomes lower.

3.2 Paraphrase Data

Sentences synthesized from templates are not representative of real human input, thus we ask crowdsource workers to rephrase them in more natural sentences. However, manual paraphrasing is not only expensive, but it is also error-prone, especially when the synthesized commands are complex and do not make sense to humans. Thus, Genie lets the developer decide which synthesized sentences to paraphrase. Genie also has a crowdsourcing framework designed to improve the quality of paraphrasing.

Choosing Sentences to Paraphrase. Developers can control the subset of templates to paraphrase as well as their sampling rates. It is advisable to obtain some paraphrases for every primitive, but combinations of functions need not be sampled evenly, since coverage is provided by including the synthesized data in training. Our priority is to choose sentences that workers can understand and can provide a high-quality paraphrase.

Developers can provide lists of easy-to-understand and hard-to-understand functions. We can maximize the success of paraphrasing, while providing some coverage, by creating compound sentences that combine the easy functions with difficult ones. We avoid combining unrelated functions because they confuse workers.

Developers can also specify input parameter values to make the synthesized sentences easier to understand. String parameters are quoted, Twitter usernames have @-signs, etc, so workers can identify them as such and copy them in the paraphrased sentences properly. (Note that quotes are removed before they are used for training).

Crowdsourcing. Genie also automates the process of crowdsourcing paraphrases. Based on the selected set of synthesized sentences to paraphrase, Genie produces a file that can

be used to create a batch of crowdsource tasks on the Amazon Mechanical Turk platform. To increase variety, Genie prepares the crowdsource tasks so that multiple workers see the same synthesized sentence, and each worker is asked to provide two paraphrases for each sentence. We found that people will only make the most obvious change if asked to provide one paraphrase, and they have a hard time writing three different paraphrases.

Due to ambiguity in natural language and workers not reading instructions and performing minimal work etc., the answers provided can be wrong. Genie uses heuristics to discard obvious mistakes, and asks workers to check the correctness of remaining answers.

3.3 Parameter Replacement & Data Augmentation

During training, it is important that the model sees many different combinations of parameter values, so as not to overfit on specific values present in the training set. Genie has a built-in database containing 49 different parameter lists and gazettes of named entities, including corpora of YouTube video titles and channel names, Twitter and Instagram hashtags, song titles, people names, country names, currencies, etc. These corpora were collected from various resources on the Web and from previous academic efforts [1, 11, 14, 17, 19, 22, 31, 32, 41, 49, 62]. Genie also includes corpora of English text, both completely free-form, and specific to messages, social media captions and news articles. This allows Genie to understand parameter values outside a closed knowledge base and provides a fallback for generic parameters. Overall, Genie's database includes over 7.8 million distinct parameter values, of which 3 million are for free-form text parameters. Genie expands the synthesized and paraphrase dataset by substituting parameters from user-supplied lists or its parameter databases. Finally, Genie also applies standard data augmentation techniques based on PPDB [18] to the paraphrases.

3.4 Combining Synthesized and Paraphrase Data

Synthesized data are not only used to obtain paraphrases, but are also used as training data. Synthesized data provides variance in the space of programs, and enables the model to learn type-based compositionality, while paraphrase data provides linguistic variety.

Developers can generate different sets of synthesized data according to the understandability of the functions or the presence of certain programming language features, such as compound commands, filters, timers, etc. They can also control the size of each group by controlling the number of instantiations of each sentence with different parameters.

4 Neural Semantic Parsing Model

Genie's semantic parser is based on Multi-Task Question Answering Network (MQAN), a previously-proposed model

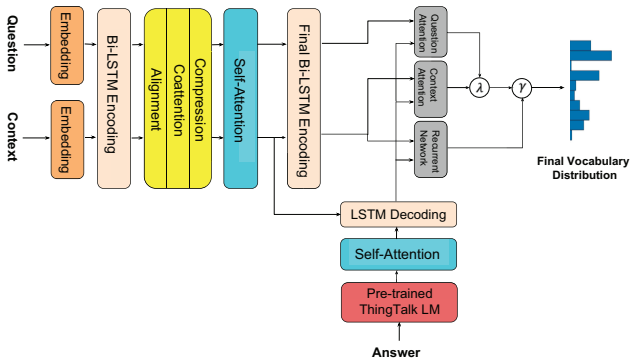

Figure 6. Model architecture of Genie's semantic parser.

architecture that was found effective on a variety of NLP tasks [39] such as Machine Translation, Summarization, Semantic Parsing, etc. MQAN frames all NLP tasks as contextual question-answering. A single model can be trained on multiple tasks (*multi-task training* [10]), and MQAN uses the question to switch between tasks. In our semantic parsing task, the context is the natural language input from the user, and the answer is the corresponding program. The question is fixed because Genie does not use multi-task training, and has a single input sentence.

4.1 Model Description

In MQAN both the encoder and decoder use a deep stack of recurrent, attentive and feed-forward layers to construct their input representations. In the encoder, each context and question are first embedded by concatenating word and character embeddings, and then fed to the encoder to construct the context and questions representations. The decoder uses a mixed pointer-generator architecture to predict the target program one token at a time; at each step, the decoder predicts a probability of either copying a token from the context or question, or generating one from a vocabulary, and then computes a distribution over input words and a distribution over vocabulary words. Two learnable scalar switches are then used to weight each distribution; the output is the token with the highest probability, which is then fed to the next step of the decoder, auto-regressively. The model is trained using token-level cross-entropy loss. We refer the readers to the original paper [39] for more details.

4.2 ThingTalk Language Model

Genie applies a pre-trained recurrent language model (LM) [40, 48] to encode the answer (ThingTalk program) before feeding it to MQAN. The high-level model architecture is shown in Fig. 6. Previous work has shown that using supervised [38, 48] and unsupervised [45] pre-trained language models as word embedding can be effective, because it captures meanings and relations in context, and exposes the model to words and concepts outside the current task.

The ThingTalk LM is trained on a large set of synthesized programs. This exposes the model to a much larger space of programs than the paraphrase set, without disrupting the balance of paraphrases and synthesized data in training.

4.3 Hyperparameters and Training Details

Genie uses the implementation of MQAN provided by decaNLP [39], an open-source library. Preprocessing for tokenization and argument identification was performed using the CoreNLP library [37], and input words are embedded using pre-trained 300-dimensional GloVe [43] and 100-dimensional character n-gram embeddings [21]. The decoder embedding uses 1-layer LSTM language model, provided by the floyhub open source library [16], and is pretrained on a synthesized set containing 20,168,672 programs. Dropout [50] is applied between all layers. Hyperparameters were tuned on the validation set, which is also used for early stopping. All our models are trained to perform the same number of parameter updates (100,000), using the Adam optimizer [28]. Training takes about 10 hours on a single GPU machine (Nvidia V100).

5 Experimentation

This section evaluates the performance of Genie on ThingTalk. Previous neural models trained on paraphrase data have only been evaluated with paraphrase data [57]. However, getting good accuracy on paraphrase data does not necessarily mean good accuracy on real data. Because acquiring real data is prohibitively expensive, our strategy is to create a high-quality training set based on synthesis and paraphrasing, and to validate and test our model with a small set of realistic data that mimics the real user input.

In the following, we first describe our datasets. We evaluate the model on paraphrases whose programs are not represented in training. This measures the model's compositionality, which is important since real user input is unlikely to match any of the synthesized programs, due to the large program space. Next, we evaluate Genie on the realistic data. We compare Genie's training strategy against models trained with either just synthesized or paraphrase data, and perform an ablation study on language and model features. Finally, we analyze the errors and discuss the limitations of the methodology.

Our experiments are performed on the set of Thingpedia skills available at the beginning of the study, which consists of 131 functions, 178 distinct parameters, and 44 skills. Our evaluation metric is *program accuracy*, which considers the result to be correct only if the output has the correct functions, parameters, joins, and filters. This is equivalent to having the output match the canonicalized generated program exactly. To account for ambiguity, we manually annotate each sentence in the test sets with all programs that provide a valid interpretation. The Genie toolkit and our datasets are available for download from our GitHub page[1].

5.1 Evaluation Data

We used three methods to gather data that mimics real usage: from the developers, from users shown a cheatsheat containing functions in Thingpedia, and IFTTT users. Because of the cost in acquiring such data, we managed to gather only 1820 sentences, partitioned into a 1480-sentence validation set and a 340-sentence test set.

Developer Data Almond has an online training interface that lets developers and contributors of Thingpedia write and annotate sentences. 1024 sentences are collected from this interface from various developers, including the authors. In addition, we manually write and annotate 157 sentences that we find useful for our own use. In total, we collect 1174 sentences, corresponding to 642 programs, which we refer to as the *developer data*.

Cheatsheet Data Users are expected to discover Almond's functionality by scanning a cheatsheet containing phrases referring to all supported functions [8]. We collect the next set of data from crowdsource workers by first showing them phrases of 15 randomly sampled skills. We then remove the cheatsheet and ask them to write compound commands that they would find useful, using the services they just saw. We annotate those sentences that can be implemented with the functions in Thingpedia. By removing the cheatsheet before writing commands, we prevent workers from copying the sentences verbatim and thus obtain more diverse and realistic sentences. Through this technique, we collect 435 sentences, corresponding to 342 distinct programs.

IFTTT Data We use IFTTT as a totally independent source of sentences for test data. IFTTT allows users to construct applets, a subset of the ThingTalk grammar, using a graphical user interface. Each applet comes with a natural language description. We collect descriptions of the most popular applets supported by Thingpedia and manually annotate them.

Because descriptions are not commands, they are often incomplete; e.g. virtual assistants are not expected to interpret a description like "IG to FB" to mean posting an Instagram photo on Facebook. We adapt the complete descriptions using the rules shown in Table 2. In total, we collect 211 IFTTT sentences, corresponding to 154 programs.

We create a 1480-sentence *validation set*, consisting of the entire developer data, 208 sentences from cheatsheet and 98 from IFTTT. The remaining 227 cheatsheet sentences and 113 IFTTT sentences are used as our *test set*; this set provides the final metric to evaluate the quality of the ThingTalk parser that Genie produces.

[1] https://github.com/stanford-oval/genie-toolkit

Table 2. IFTTT dataset cleanup rules.

Modification	Example (before)	Example (after)
Replace second-person pronouns to first-person	Blink your light	Blink my light
Replace placeholders with specific values	... set the temperature to ___°	... set the temperature to 25 C
Append the device name if ambiguous otherwise	Let the team know when it rains	Let the team know when it rains on Slack
Remove UI-related explanation	Make your Hue Lights color loop with this button	Make your Hue Lights color loop
Replace under-specified parameters with real values	Message my partner when I leave work	Message John when I leave work

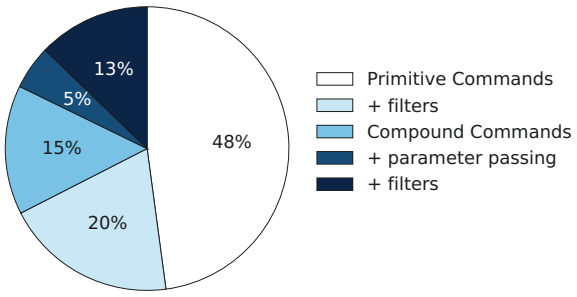

Figure 7. Characteristics of the ThingTalk training set (combining paraphrases and synthesized data).

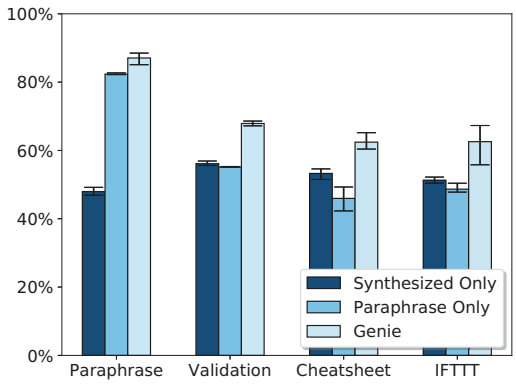

Figure 8. Accuracy of the Genie model trained on just synthesized data, just paraphrase data, or with the Genie training strategy. Error bars indicate the range of results over 3 independently trained models.

5.2 Evaluation

Limitation of Paraphrase Tests Previous paraphrase-based semantic parsers are evaluated only with paraphrases similar to the training data [57]. Furthermore, previous work synthesizes programs with just one construct template and one primitive template per function. Using this methodology on the *original* ThingTalk, a dataset of 8,195 paraphrase sentences was collected. After applying Genie's data augmentation on this dataset, the model achieves 95% accuracy. However, if we test the model on paraphrases of compound programs whose function combinations differ from those in training, the accuracy drops to 48%. This suggests that the dataset is too small to teach the model compositionality, even for paraphrases of synthesized data. The accuracy drops to around 40% with the realistic validation set. This result shows that a high accuracy on a paraphrase test that matches programs in training is a poor indication of the quality of the model. More importantly, we could not improve the result by collecting more paraphrases using this methodology, as the new paraphrases do not introduce much variation.

Training Data Acquisition Templates help in improving the training data by allowing more varied sentences to be collected. Our ThingTalk modification which combines triggers and retrievals as queries is also important; queries can be expressed as noun phrases, which can be inserted easily in richer construct templates. We wrote 35 construct templates for primitive commands, 42 for compound commands, and 68 for filters and parameters. In addition, we also wrote 1119 primitive templates, which is 8.5 templates per function on average.

Using the new templates, we synthesize 1,724,553 sentences, corresponding to 77,716 distinct programs, using a target size of 100,000 samples per grammar rule and a maximum depth of 5. With these settings, the synthesis takes about 25 minutes and uses around 23GBs of memory. The synthesized set is then sampled and paraphrased into 24,451 paraphrased sentences. After PPDB augmentation and parameter expansion, the training set contains 3,649,222 sentences. Paraphrases with string parameters are expanded 30 times, other paraphrases 10 times, synthesized primitive commands 4 times, and other synthesized sentences only once. Overall, paraphrases comprise 19% of the training set. The characteristic of this combined dataset is shown in Fig. 7; primitive commands are commands that use one function, while compound commands use two. The training set contains 680,408 distinct programs, which include 4,710 unique combinations of Thingpedia functions.

Sentences in the synthesized set use 770 distinct words; this number increases to 2,104 after paraphrasing and to 208,429 after PPDB augmentation and parameter expansion. On average, each paraphrase introduces 38% new words and 65% new bigrams to a synthesized sentence.

Evaluation on Paraphrases with Untrained Programs Our paraphrase test set contains 1,274 sentences, corresponding to 600 programs and 149 pairs of functions. All test sentences are compound commands that use function combinations *not* appearing in training. We show in Fig. 8 the average,

minimum, and maximum of program accuracy obtained with 3 different training runs. On the paraphrase test set, Genie obtains an average program accuracy of 87%, showing that Genie can generalize to compound paraphrase commands for programs not seen in training. This improvement in generalization is due to both the increased size and variance in the training set.

Evaluation on Test Data For the validation data described in Section 5.1, Genie achieves an average of 68% program accuracy. Of the 1480 sentences in the validation set, 272 sentences map to programs not in training; the rest uses different string parameters to programs that appear in the training set. Genie's accuracy is 30% for the former and 77% for the latter. This suggests that we need to improve on Genie's compositionality for sentences in the wild.

When applied to the *test* data, Genie achieves full program accuracy of 62% on cheatsheet data and 63% IFTTT commands, respectively. The difference in accuracy between the paraphrase test data and the realistic test sets underscores the limitation of paraphrase testing. Previous work on IFTTT to parse the high-level natural language descriptions of the rules can only achieve a 3% program accuracy [46].

5.3 Synthesized and Paraphrase Training Strategy

The traditional methodology is to train with just paraphrased sentences. As shown in Fig. 8, training on paraphrase data alone delivers a program accuracy of 82% on the paraphrase test, 55% on the validation set, 46% on cheatsheet test data, and 49% on IFTTT test data. This suggests that the smaller size of the paraphrase set, even after data augmentation, causes the model to overfit. Adding synthesized data to training improves the accuracy across the board, and especially for the validation and real data test sets.

On the other hand, training with synthesized data alone delivers a program accuracy of 48% on the paraphrase test, 56% on the validation set, 53% on cheatsheet test data, and 51% on IFTTT test data. When compared to paraphrase data training, it performs poorly on the paraphrase test, but performs better on the cheatsheet and IFTTT test data.

The combinination of using synthesized and paraphrase data works best. The synthesized data teaches the neural model many combinations not seen in the paraphrases, and the paraphrases teach the model natural language usage. Thus, synthesized data os not just useful as inputs to paraphrasing, but can expand the training dataset effectively and inexpensively.

5.4 Evaluation of VAPL and Model Features

Here we perform an ablation study, where we remove a feature at a time from the design of Genie or ThingTalk to evaluate its impact. We report, in Table 3, results on three datasets: the paraphrase test set, the validation set, and those programs in the validation set that have new combinations of functions, filters, and parameters not seen in training. We report the average across three training runs, along with the error representing the half range of results obtained.

Table 3. Accuracy results for the ablation study. Each "–" row removes one feature independently.

Model	Paraphrase	Validation	New Program
Genie	87.1 ± 1.8	**67.9 ± 0.7**	29.9 ± 3.2
– canonicalization	80.0 ± 1.3	63.2 ± 0.9	21.9 ± 0.9
– keyword param.	84.0 ± 0.6	66.6 ± 0.3	25.0 ± 2.0
– type annotations	86.9 ± 3.6	67.5 ± 0.6	**31.0 ± 1.1**
– param. expansion	78.3 ± 4.8	66.3 ± 0.4	30.5 ± 1.3
– decoder LM	**88.7 ± 1.0**	66.8 ± 0.8	27.3 ± 1.7

Canonicalization We evaluate canonicalization by training a model where keyword parameters are shuffled independently on each training example. (Note that programs are canonicalized during evaluation). Canonicalization is the most important VAPL feature, improving the accuracy by 5 to 8% across the three datasets.

Keyword Parameters Replacing keyword parameter with positional parameters decreases performance by 3% on paraphrases and 5% on new programs in the validation set. This suggests that keyword parameters improve generalization to new programs.

Type Annotations Type annotations are found to have no measurable effect on the accuracy, as the differences are within the margin of error. We postulate that keyword parameters are an effective substitute for type annotations, as they convey not just the type information but the semantics of the data as well.

Parameter Expansion Removing parameter expansion means that every sentence now appears only once in the training set. This decreases performance by 9% on paraphrases due to overfitting. It has little effect on the validation set because only a small set of parameters are used.

Pretrained Decoder Language Model We compare our full model against one that uses a randomly initialized and jointly trained embedding matrix for the program tokens. Our proposal to augment the MQAN model with a pretrained language model improves the performance on new programs by 3%, because the model is exposed to more programs during pretraining. It lowers the performance for the paraphrase set slightly, however, because the basic model fits the paraphrase distribution well.

5.5 Discussion

Error analysis on the validation set reveals that Genie produces a syntactically correct and type-correct program for 96% of the inputs, indicating that the model can learn syntax

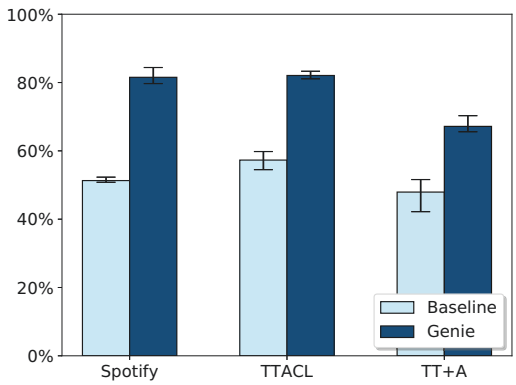

Figure 9. Accuracy of the three case studies on cheatsheet test data. Baseline refers to a model trained with no synthesized data, no PPDB augmentation, and no parameter expansion. Error bars indicate the range of results over 3 independently trained models.

and type information well. Genie identifies correctly whether the input is a primitive or a compound with 91% accuracy, and can identify the correct skills for 87% of the inputs. 82% of the generated programs use the correct functions; the latter metric corresponds to the *function accuracy* introduced in previous work on IFTTT [46]. Finally, less than 1% of the inputs have the correct functions, parameter names, and filters but copy the wrong parameter value from the input.

As observed on the validation set, the main source of errors is due to the difficulty of generalizing to programs not seen in training. Because the model can generalize well on paraphrases, we believe this is not just a property of the neural model. Real natural language can involve vocabulary that is inherently not compositional, such as "autoreply" or "forward"; some of that vocabulary can also be specific to a particular domain, like "retweet". It is necessary to learn the vocabulary from real data. The semantic parser generated by Genie may be used as beta software to get real user input, from which more templates can be derived.

6 Case Studies of Genie

We now apply Genie to three use cases: a sophisticated music playing skill, an access control language modeled after ThingTalk, and an extension to ThingTalk for aggregates. We compare the Genie results with a *Baseline* modeled after the Wang et al. methodology [57]: training only with paraphrase data, no data augmentation, and no parameter expansion.

6.1 A Comprehensive Spotify Skill

The Spotify skill in Almond [29] allows users to combine 15 queries and 17 actions in Spotify in creative ways. For example, users can "add all songs faster than 500 bpm to the playlist dance dance revolution", or "wake me up at 8 am by playing wake me up inside by evanescence".

Policy $\hat{\pi}$:	$\hat{p_\sigma}$: [now ⇒ $\hat{q}$ ⇒ notify | now ⇒ $\hat{a}$]
Query $\hat{q}$:	f filter p
Action $\hat{a}$:	f filter p
Source predicate $\hat{p_\sigma}$:	true | false | !p | p && p | p || p | σ *operator* v

Figure 10. The formal grammar of the primitive subset of TACL. Rules that are identical to ThingTalk are omitted.

This skill illustrates the importance of Genie's ability to handle quote-free sentences. In the original ThingTalk, a preprocessor replaces all the arguments, which must be quoted, with a PARAM token. This would have replaced "play 'shake it off' " and "play 'Taylor Swift' " with the same input "play PARAM'". However, these two sentences correspond to different API calls in Spotify because the former plays a given song and the latter plays songs by a given artist. Genie accepts the sentences without quotes and uses machine learning to distinguish between songs and artists. Unlike in previous experiments, since the parameter value is meaningful in identifying the function, we use multiple instances of the same sentence with different parameters in the test sets.

The skill developers wrote 187 templates (5.8 per function on average) and collected 1,553 paraphrases. After parameter expansion, we obtain a dataset with 165,778 synthesized sentences and 217,258 paraphrases.

We evaluate on two test sets: a paraphrase set of 684 paraphrase sentences, and a cheatsheet test set of 128 commands; the latter set contains 95 primitive commands and 33 compound commands. We also keep 675 paraphrases in the validation set, and train the model with the rest of the data.

On paraphrases, Genie achieves a 98% program accuracy, while the Baseline model achieves 86%. On cheatsheet data, Genie achieves 82% accuracy, an improvement of 31% over the Baseline model (Fig. 9). Parameter expansion is mainly responsible for the improvement because it is critical to identify the song or artist in the sentences.

6.2 ThingTalk Access Control Language

We next use Genie for TACL, an access control policy language that lets users describe who, what, when, where, and how their data can be shared [9]. A policy consists of the person requesting access and a ThingTalk command. For example, the policy "my secretary is allowed to see my work emails" is expressed as:

σ = "secretary" : now
⇒ @com.gmail.inbox filter *labels* contains "work"
⇒ notify

The previously reported semantic parser handles only primitive TACL policies, whose formal grammar is shown in Fig. 10. All parameters are expected to be quoted, which renders the model impractical with spoken commands. An accuracy of 74% was achieved on a dataset consisting of 4,742 paraphrased policy commands; the number includes both training and test.

Our first experiment reuses the same dataset above, but with quotes removed. From 6 construct templates, Genie synthesizes 432,511 policies, and combines them with the existing dataset to form a total training set of 543,566 policies, after augmentation. We split the dataset into 526,322 sentences for training, 701 for validation, and 702 for testing; the test consists exclusively of paraphrases unique to the whole set, even when ignoring the parameter values. On this set, Genie achieves a high accuracy of 96%.

For a more realistic evalution, we create a test set of 132 policy sentences, collected using the cheatsheet technique. We reuse the same cheatsheet as the main ThingTalk experiment; instructions are modified to elicit access control policies rather than commands. On this set, Genie achieves an accuracy of 82%, with a 25% improvement over the Baseline model. This shows that the data augmentation technique in Genie is effective in improving the accuracy, even if the paraphrase portion of the training set is relatively small. The high accuracy is likely due to the limited scope of policy sentences: the number of meaningful primitive policies is relatively small compared to the full Thingpedia.

6.3 Adding Aggregation to ThingTalk

One of our goals in building Genie is to grow virtual assistants' ability to understand more commands. Our final case study adds aggregation to ThingTalk, i.e. finding min, max, sum, average, etc. Our new language TT+A extends ThingTalk with the following grammar:

Query q: `agg` [`max` | `min` | `sum` | `avg`] *pn* `of` (q) |
`agg count of` (q)

Users can compute aggregations over results of queries, such as "find the total size of a folder", which translates to

```
now ⇒ agg sum file_size of (@com.dropbox.list_folder())
    ⇒ notify
```

While this query can be used as a clause in a compound command, we only test the neural network with aggregation on primitive queries. We wrote 6 templates for this language.

The ThingTalk skill library currently has 4 query functions that return lists of numeric results, and 20 more that returns lists in general (on which the count operator can be applied). We synthesize 82,875 sentences and collect 2,421 paraphrases of aggregation commands. For training, we add to the full ThingTalk dataset 270,035 aggregation sentences: 23,611 are expanded paraphrases and the rest are synthesized. The accuracy on the paraphrase test set is close to 100% for both Genie and the Baseline; this is due to the limited set of possible aggregation commands and the fact that programs in the paraphrase test set also appear in the training set.

We then test on a small set of 64 aggregation commands, obtained using the cheatsheet method. In this experiment, the cheatsheet is restricted to only queries where aggregation is possible. We note that the cheatsheet, in particular, does not show the output parameters of each API, so crowdsource workers guess which parameters are available to aggregate based on their knowledge of the function; this choice makes the data collection more challenging, but improves the realism of inputs because workers are less biased. On this set, Genie achieves a program accuracy of 67% without any iteration on templates (Fig. 9), an improvement of 19% over the Baseline. This accuracy is in line with the general result on ThingTalk commands, and suggests that Genie can support extending the language ability of virtual assistants effectively.

7 Related Work

Alexa The Alexa assistant is based on the Alexa Meaning Representation Language (AMRL) [30], a language they designed to support semantic parsing of Alexa commands.

AMRL models natural language closely: for example, the sentence "find the sharks game and find me a restaurant near it" would have a different representation than "find me a restaurant near the sharks game" [30]. In ThingTalk, both sentences would have the same executable representation, which enables paraphrasers to switch from one to the other.

AMRL has been developed on a closed ontology of 93 actions and 60 intents, using a dataset of sentences manually annotated by experts (not released publicly). The best accuracy reported on this dataset is 77% [44].

AMRL is not available to third-party developers. Third-party skills have access to a joint intent-classification and slot-tagging model [20], which is equivalent to a single ThingTalk action. Free-form text parameters are further limited to one per sentence and must use a templatized carrier phrase. The full power of ThingTalk and Genie is instead available to all contributors to the library.

IFTTT If-This-Then-That [23] is a service that allows users to combine services into trigger-action rules. Previous work [46] attempted to translate the English description of IFTTT rules into executable code. Their method, and successive work using the IFTTT dataset [2, 5, 15, 35, 63, 64], showed moderate success in identifying the correct functions on a filtered set of unambiguous sentences but failed to identify the full programs with parameters. They found that the descriptions are too high-level and are not precise commands. For this reason, the IFTTT dataset is unsuitable to train a semantic parser for a virtual assistant.

Data Acquisition and Augmentation Wang et al. propose to use paraphrasing technique to acquire data for semantic parsing; they sample canonical sentences from a grammar, crowdsource paraphrases them and then use the paraphrases as training data [57]. Su et al. [52] explore different sampling methods to acquire paraphrases for Web API. They focus on 2 APIs; our work explores a more general setting of 44 skills.

Previous work [25] has proposed the use of a grammar of natural language for data augmentation. Their work supports

data augmentation with power close to Genie's parameter expansion, but it does so with a grammar that is automatically inferred from natural language. Hence, their work requires an initial dataset to infer the grammar, and cannot be used to bootstrap a new formal language from scratch.

A different line of work by Kang et al. [65] also considered the use of generative models to expand the training set; this is shown to increase accuracy. Kang et al. focus on joint intent recognition and slot filling. It is not clear how well their technique would generalize to the more complex problem of semantic parsing.

Semantic Parsing Genie's model is based upon previous work in semantic parsing, which was used for queries [6, 24, 42, 54, 57, 58, 60, 61, 66–69], instructions to robotic agents [12, 26, 27, 59], and trading card games [34, 47, 64].

Full SQL does not admit a canonical form, because query equivalence is undecidable [13, 55], so previous work on database queries have targeted restricted but useful subsets [69]. ThingTalk instead is designed to have a canonical form.

The state-of-the-art algorithm is sequence-to-sequence with attention [3, 15, 53], optionally extended with a copying mechanism [25], grammar structure [47, 64], and tree-based decoding [2]. In our experiments, we found that the use of grammar structure provided no additional benefit.

8 Conclusion

Virtual assistants can greatly simplify and enhance our lives. Yet, building a virtual assistant that supports novel capabilities is challenging because of a lack of annotated natural language training data. Commercial efforts use formal representations motivated by natural languages and labor-intensive manual annotation [30]. This is not scalable to the expected growth of virtual assistants.

We advocate using a formal VAPL language to represent the capability of a virtual assistant and use a neural semantic parser to directly translate user input into executable code. A previous attempt of this approach failed to create a good parser [8]. This paper shows it is necessary to design the VAPL language in tandem with a more sophisticated data acquisition methodology.

We propose a methodology and a toolkit, Genie, to create semantic parsers for new virtual assistant capabilities. Developers only need to acquire a small set of manually annotated realistic data to use as validation. They can get a high-quality training set by writing construct and primitive templates. Genie uses the templates to synthesize data, crowdsources paraphrases for a sample, and augments the data with large parameter datasets. Developers can refine the templates iteratively to improve the quality of the training set and the resulting model.

We identify generally-applicable design principles that make VAPL languages amenable to natural language translation. By applying Genie to our revised ThingTalk language, we obtain an accuracy of 62% on realistic data. Our work is the first virtual assistant to support compound commands with unquoted free-form parameters. Previous work either required quotes [8] or obtained only 3% accuracy [46].

Finally, we show that Genie can be applied to three use cases in different domains; our methodology improves the accuracy between 19% and 33% compared to the previous state of the art. This suggests that Genie can be used to bootstrap new virtual assistant capabilities in a cost-effective manner.

Genie is developed as a part of the open-source Almond project [7]. Genie, our data sets, and our neural semantic parser, which we call LUInet (Linguistic User Interface network), are all freely available. Developers can use Genie to create cost-effective semantic parsers for their own domains. By collecting contributions in VAPL constructs, Thingpedia entries, and natural language sentences from developers in different domains, we can potentially grow LUInet to be the best and publicly available parser for virtual assistants.

Acknowledgments

We thank Rakesh Ramesh for his contributions to a previous version of Genie, and Hemanth Kini and Gabby Wright for their Spotify skill. Finally, we thank the anonymous reviewers and the shepherd for their suggestions.

This work is supported in part by the National Science Foundation under Grant No. 1900638 and the Stanford MobiSocial Laboratory, sponsored by AVG, Google, HTC, Hitachi, ING Direct, Nokia, Samsung, Sony Ericsson, and UST Global.

References

[1] Tiago A. Almeida, José María G. Hidalgo, and Akebo Yamakami. 2011. Contributions to the study of SMS spam filtering. In *Proceedings of the 11th ACM symposium on Document engineering - DocEng '11*. ACM Press. https://doi.org/10.1145/2034691.2034742

[2] David Alvarez-Melis and Tommi S Jaakkola. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR-2017)*.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[4] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. 178–186.

[5] I. Beltagy and Chris Quirk. 2016. Improved Semantic Parsers For If-Then Statements. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/p16-1069

[6] Jonathan Berant and Percy Liang. 2014. Semantic Parsing via Paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.3115/v1/p14-1133

[7] Giovanni Campagna, Michael Fischer, Mehrad Moradshahi, Silei Xu, Jackie Yang, Richard Yang, and Monica S. Lam. 2019. Almond: The Stanford Open Virtual Assistant Project. https://almond.stanford.edu.

[8] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, New York, New York, USA, 341–350. https://doi.org/10.1145/3038912.3052562

[9] Giovanni Campagna, Silei Xu, Rakesh Ramesh, Michael Fischer, and Monica S. Lam. 2018. Controlling Fine-Grain Sharing in Natural Language with a Virtual Assistant. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (sep 2018), 1–28. https://doi.org/10.1145/3264905

[10] Rich Caruana. 1998. Multitask Learning. (1998), 95–133. https://doi.org/10.1007/978-1-4615-5529-2$_5$

[11] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. 2013. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. *CoRR* abs/1312.3005 (2013). arXiv:1312.3005 http://arxiv.org/abs/1312.3005

[12] David L Chen and Raymond J Mooney. 2011. Learning to Interpret Natural Language Navigation Instructions from Observations. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-2011)*. 859–865.

[13] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

[14] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[15] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/p16-1004

[16] Floydhub. 2019. Word Language Model . https://github.com/floydhub/word-language-model.

[17] Winthrop Nelson Francis. 1965. *A Standard Corpus of Edited Present-Day American English*. Vol. 26. JSTOR. 267 pages. https://doi.org/10.2307/373638

[18] Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. PPDB: The Paraphrase Database. In *Proceedings of NAACL-HLT*. Association for Computational Linguistics, Atlanta, Georgia, 758–764. http://cs.jhu.edu/~ccb/publications/ppdb.pdf

[19] Fabio Gasparetti. 2016. Modeling user interests from web browsing activities. *Data Mining and Knowledge Discovery* 31, 2 (nov 2016), 502–547. https://doi.org/10.1007/s10618-016-0482-x

[20] Anuj Kumar Goyal, Angeliki Metallinou, and Spyros Matsoukas. 2018. Fast and Scalable Expansion of Natural Language Understanding Functionality for Intelligent Agents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n18-3018

[21] Kazuma Hashimoto, caiming xiong, Yoshimasa Tsuruoka, and Richard Socher. 2017. A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://doi.org/10.18653/v1/d17-1206

[22] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*. 1693–1701.

[23] If This Then That 2011. If This Then That. http://ifttt.com.

[24] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/p17-1089

[25] Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/p16-1002

[26] Rohit J. Kate and Raymond J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the ACL - ACL '06*. Association for Computational Linguistics. https://doi.org/10.3115/1220175.1220290

[27] Rohit J Kate, Yuk Wah Wong, and Raymond J Mooney. 2005. Learning to transform natural to formal languages. In *Proceedings of the 20th national conference on Artificial intelligence-Volume 3*. AAAI Press, 1062–1068.

[28] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[29] Hemanth Kini and Gabby Wright. 2018. Thingpedia Spotify Skill. https://almond.stanford.edu/thingpedia/devices/by-id/com.spotify.

[30] Thomas Kollar, Danielle Berry, Lauren Stuart, Karolina Owczarzak, Tagyoung Chung, Lambert Mathias, Michael Kayser, Bradford Snow, and Spyros Matsoukas. 2018. The Alexa Meaning Representation Language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n18-3022

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[32] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*. ACM Press. https://doi.org/10.1145/1557019.1557077

[33] Percy Liang. 2013. Lambda Dependency-Based Compositional Semantics. *CoRR* abs/1309.4408 (2013). arXiv:1309.4408 http://arxiv.org/abs/1309.4408

[34] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. (2016). https://doi.org/10.18653/v1/p16-1057

[35] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent Attention For If-Then Program Synthesis. In *Advances in Neural Information Processing Systems*. 4574–4582.

[36] Douglas MacMillan. 2018. Amazon Says It Has Over 10,000 Employees Working on Alexa, Echo. https://www.wsj.com/articles/amazon-says-it-has-over-10-000-employees-working-on-alexa-echo-1542138284. *The Wall Street Journal* (2018).

[37] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics. https://doi.org/10.3115/v1/p14-5010

[38] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. 2017. Learned in translation: Contextualized word vectors. In *Advances in Neural Information Processing Systems*. 6294–6305.

[39] Bryan McCann, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. 2018. The Natural Language Decathlon: Multitask Learning as Question Answering. *arXiv preprint arXiv:1806.08730* (2018).

[40] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

[41] Cesc Chunseong Park, Byeongchang Kim, and Gunhee Kim. 2017. Attend to You: Personalized Image Captioning with Context Sequence Memory Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. https://doi.org/10.1109/cvpr.2017.681

[42] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.3115/v1/p15-1142

[43] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 1532–1543. https://doi.org/10.3115/v1/d14-1162

[44] Vittorio Perera, Tagyoung Chung, Thomas Kollar, and Emma Strubell. 2018. Multi-task learning for parsing the alexa meaning representation language. In *American Association for Artificial Intelligence (AAAI)*. 181–224.

[45] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/n18-1202

[46] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics. https://doi.org/10.3115/v1/p15-1085

[47] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 1139–1149. https://doi.org/10.18653/v1/p17-1105

[48] Prajit Ramachandran, Peter Liu, and Quoc Le. 2017. Unsupervised Pretraining for Sequence to Sequence Learning. (2017). https://doi.org/10.18653/v1/d17-1039

[49] Jitesh Shetty and Jafar Adibi. 2004. The Enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California* 4, 1 (2004), 120–128.

[50] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[51] Mark Steedman and Jason Baldridge. 2011. Combinatory Categorial Grammar. In *Non-Transformational Syntax*. Wiley-Blackwell, 181–224. https://doi.org/10.1002/9781444395037.ch5

[52] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. 2017. Building Natural Language Interfaces to Web APIs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management - CIKM '17*. ACM Press. https://doi.org/10.1145/3132847.3133009

[53] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[54] Lappoon R. Tang and Raymond J. Mooney. 2001. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *Machine Learning: ECML 2001*. Springer Berlin Heidelberg, 466–477. https://doi.org/10.1007/3-540-44795-4$_4$0

[55] Boris A. Trakhtenbrot. 1950. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR* 70 (1950), 569–572. https://doi.org/10.1090/trans2/023/01

[56] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems - CHI '16*. ACM Press, 3227–3231. https://doi.org/10.1145/2858036.2858556

[57] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 1332–1342. https://doi.org/10.3115/v1/p15-1129

[58] Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. 960–967.

[59] Yuk Wah Wong and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. Association for Computational Linguistics, 439–446. https://doi.org/10.3115/1220835.1220891

[60] Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based Structured Prediction for Semantic Parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 1341–1350. https://doi.org/10.18653/v1/p16-1127

[61] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436* (2017).

[62] Jaewon Yang and Jure Leskovec. 2011. Patterns of Temporal Variation in Online Media. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM '11)*. ACM, New York, NY, USA, 177–186. https://doi.org/10.1145/1935826.1935863

[63] Ziyu Yao, Xiujun Li, Jianfeng Gao, Brian Sadler, and Huan Sun. 2018. Interactive Semantic Parsing for If-Then Recipes via Hierarchical Reinforcement Learning. *arXiv e-prints*, Article arXiv:1808.06740 (Aug 2018), arXiv:1808.06740 pages. arXiv:cs.CL/1808.06740

[64] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 440–450.

[65] Kang Min Yoo, Youhyun Shin, and Sang-goo Lee. 2018. Data Augmentation for Spoken Language Understanding via Joint Variational Generation. *CoRR* abs/1809.02305 (2018). arXiv:1809.02305 http://arxiv.org/abs/1809.02305

[66] John M Zelle and Raymond J Mooney. 1994. Inducing deterministic Prolog parsers from treebanks: A machine learning approach. In *AAAI*. 748–753.

[67] John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 2*. AAAI Press, 1050–1055.

[68] Luke S Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 658–666.

[69] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint arXiv:1709.00103* (2017).