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

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

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

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

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

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

Abstract
This paper presents Genie, a toolkit that helps developers expand the natural language capability of virtual assistants. We propose formalizing the capability of virtual assistants with a Virtual Assistant Programming Language (VAPL), whose design is informed by principles that make it amenable to natural language translation. We also propose a Natural Language Template Language so developers can easily describe how programs in a VAPL can be expressed in natural language. Given a VAPL language definition and curated natural language templates, Genie generates a semantic parser that translates natural language to code for the new language. The parser is a deep neural network trained on synthetically generated sentences and their crowdsourced paraphrases.

By adopting good VAPL design principles and applying Genie, we improved the design of ThingTalk, a language used by the open-source Almond virtual assistant. Genie delivers a parser for ThingTalk that obtains an 87% accuracy on the paraphrase dataset, and 63% accuracy on a crowdsourced test set; this represents a 16% increase over state-of-the-art models. In addition, we demonstrated easy extensibility with a music skill, aggregate functions, and access control; each improves between 26% and 28% over the baseline.

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

Keywords data augmentation, semantic parsing, virtual assistants, natural language user interfaces

ACM Reference Format:

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

1 Introduction

Personal virtual assistants provide users a natural language interface to a wide variety of web services and IoT devices. Virtual assistants are commonly based on semantic parsing, a machine learning algorithm that converts natural language to a semantic representation in a formal language. Not only must they understand primitive commands across many domains, but they also need to understand the composition of these commands to perform full tasks. This breadth of the interface in a virtual assistant 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, and presents Genie, a tool for creating the first semantic parser for new virtual assistant capabilities to bootstrap the acquisition of real user data.

1.1 Virtual Assistant Programming Languages

Commercial assistants typically translate natural language into an intermediate representation that matches the semantics of the sentences closely [4, 29, 32, 42, 49]. For example, the Alexa Meaning Representation Language [29, 42] 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, which cannot make use of free-form text parameters.

Our approach is to represent the capability of the virtual assistant fully and formally as a VAPL; we use a semantic parser to translate natural language into VAPL code, which can directly be executed by the assistant. New capabilities can be supported by extending the VAPL. VAPLs expose the full capability of the virtual assistant to the deep-learning neural network, eliminating the need for the design and implementation of an intermediate representation.

The ThingTalk language designed for the open-source Almond virtual assistant is an example of a VAPL [7]. 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 200 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 [54]. 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. Our research to create an effective semantic parser for ThingTalk taught us the design principles for good VAPL design. It is important that the language is designed to keep the semantics of components orthogonal, and match non-developers’ mental model. In traditional semantic parsing used in answering questions, the output of the neural model is simply the answer, which can easily be checked for correctness. In our case, the output of the neural model is the VAPL code itself, since we cannot execute the code and check the results. Thus, VAPL programs must have a (unique) canonical form so we can check if the answer is correct syntactically. We applied these principles to overhaul 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

To create a semantic parser for question answering over simple domains, Wang et al. [55] 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 paraphrased sentences 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 that all possible logical forms can be enumerated.

This approach was used in the original ThingTalk semantic parser and has been shown to be inadequate [7]. 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 whose 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 an 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 crowdsourced workers to make them more natural. The paraphrases further inform more useful templates, which in turn derives more diverse sentences 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 propose 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 an 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 14% on IFTTT and 16% on realistic inputs, compared to training with paraphrase data alone.

4. Demonstration of extensibility by the success of 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. 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 ThingTalk as an example. These design principles can be summed up 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, include fine-grained domain-specific types, and provide library developers with the ability to define 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 single 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. Numbers, dates and times, which can be identified and normalized using a rule-based algorithm in the input sentence, are represented as named constants of the form “NUMBER_0”, “DATE_1”, etc., using a technique called argument identification [14]. 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. 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 [7]. 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.

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 give the system more functionality, but we also make the language more regular, which is useful to users, crowdsourced paraphrase providers, and the neural model. Without this change, it will be very difficult to get correct user inputs for training and evaluations, as too much functionality would be supported inconsistently.

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

Classes have functions. There are two kinds of functions: *query functions* retrieve data and have no side-effects, while *action functions* have side-effects but do not return data.

A query function can be *monitorable*, which means the result returned by the query can be monitored for changes. Queries are monitorable if the result changes in a predictable fashion: either they can be polled at low frequency, or the API supports push notification; queries that return a random result, such as the function to get a random cat picture shown in Fig. 1, are not monitorable. A query can return a single result, or return a list of results. Each parameter indicates its type statically.

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 User's Mental Model

VAPLs should be designed to reflect the common way in which commands are specified by users. It is more natural for users to think about subsets of data with certain characteristics, rather than execution paths. For this reason, ThingTalk is data focused and not control-flow focused. ThingTalk has a single control construct:

\[ \text{stream} \Rightarrow \text{query} \Rightarrow \text{action} \]

The **stream** clause defines when the program should be evaluated, as a continuous stream of events. The optional **query** clause defines what data should be retrieved when the program is triggered. The **action** defines what the program should do. The formal grammar is shown in Fig. 5.

An example illustrating parameter passing is shown in Fig. 1. The following example that automatically retweets all tweets by a specific user shows the use of filters:

```
monitor(@com.twitter.timeline() filter author = @PLDI)
```
The program uses the author output parameter of the function @com.twitter.retweet(tweet_id) to filter the set of tweets, and passes the tweet_id output parameter to input parameter with the same name of @com.twitter.retweet() .

Queries and Actions. A query is constructed by invoking a query function, and evaluates to a list of results; if a function returns a single result, it is converted into a singleton list. The query result can be filtered with a boolean predicate of the result. A query can also be constructed as the join of two smaller queries, optionally with parameter passing from the first to the second; the join evaluates to the cross product of the two query results and contains all output parameters of both queries. The action can invoke the builtin notify, which presents the result to the user, or one or more action functions defined in the library.

The introduction of join generalizes ThingTalk to support more than one retrieval function in the same program, and enables useful compositions such as “translate the title of New York Times articles”:

\[
\text{now} \Rightarrow \text{@com.nytimes.get_front_page()} \text{ join } \text{@com.yahoo.translate()} \text{ on text = title } \Rightarrow \text{ notify}
\]

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; lists returned by queries are implicitly traversed and the output parameters of the function are bound to each element in the list one at a time.

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 result of evaluating the query changes. Any query that uses monitorable functions can be monitored, including queries that use joins or filters.

A stream can also be constructed by applying the edge filter operator to another stream; the resulting stream triggers when the underlying stream triggers and a boolean predicate that was previously false now becomes true. An example of an edge filter operation is:

\[
\text{edge(\text{monitor } \text{@weather.current()}) on temperature < 60^\circ F} \Rightarrow \text{ notify};
\]

This program notifies the users any time the weather temperature crosses the 60 Fahrenheit threshold and becomes lower; changes to the temperature that do not cross the threshold do not cause a notification.

2.4 Canonicalization of Programs

As described in Section 1, canonicalization is key to training a neural semantic parser. ThingTalk is designed to allow a canonical form for each program. This form is constructed by applying semantic-preserving transformation rules. For example, query joins without parameter passing are a commutative operation, and are canonicalized by the lexical ordering of the operands. Nested filter operators on a single query 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 crowd-sourced 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 my Twitter profile at Facebook”. or “get a cat picture” cannot).

We have developed an NL-template language so developers and skill writers can generate variety in 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, consisting of code using that function, 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:

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

where \( \text{cat} \) is the grammar category and \( u \) is the utterance.

Examples of primitive templates, for the 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.

While designing primitive templates, it is important to choose grammar categories that compose naturally. The original design of ThingTalk used only verb phrases for queries (e.g., “get X”); we switched to primarily noun phrases as they support composition by replacement (e.g., “a cat picture” can be replaced in “post X on Twitter” and “post X on Facebook”, while “get a cat picture” cannot).

**Construct Templates.** To combine the primitives into full programs in the target formal language, the language designer additionally provides a set of construct templates, which define the mapping between formal language operators and natural language compositional constructs. A construct template has the form:

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

which says that a derivation of non-terminal \( \text{lhs} \) can be constructed by combining the literals and the non-terms \( \text{rhs} \), and then applying the semantic function \( \text{sf} \) to compute the formal language representation. For example, the following two construct templates define the two common ways to express “when - do” commands:

\[
\text{COMMAND} := s : \text{wp}^* ; a : \text{vp} \rightarrow \text{return} s \Rightarrow a;
\text{COMMAND} := a : \text{vp} ; s : \text{wp} \rightarrow \text{return} s \Rightarrow a;
\]

Together with the following two primitive templates:

\[
\text{wp} := \text{\textquote{\textquote{\textquote{when I modify a file in Dropbox}}}} \rightarrow \text{monitor} \@\text{com.dropbox.list_folder()}
\text{vp} := \text{\textquote{\textquote{send a Slack message}} } \rightarrow \text{@com.slack.send()}
\]

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”.

<table>
<thead>
<tr>
<th>Natural language</th>
<th>Cat.</th>
<th>ThingTalk Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>my Dropbox files</td>
<td>NP</td>
<td>@com.dropbox.list_folder()</td>
</tr>
<tr>
<td>my Dropbox files that changed most recently</td>
<td>NP</td>
<td>@com.dropbox.list_folder(order_by = modified_time_decreasing)</td>
</tr>
<tr>
<td>my Dropbox files that changed this week</td>
<td>NP</td>
<td>@com.dropbox.list_folder(order_by = modified_time_decreasing) filter modified_time &gt; start_of_week</td>
</tr>
<tr>
<td>files in my Dropbox folder $x$</td>
<td>NP</td>
<td>( \lambda(x : \text{PathName}) \rightarrow @\text{com.dropbox.list_folder(folder_name = x)} )</td>
</tr>
<tr>
<td>when I modify a file in Dropbox</td>
<td>WP</td>
<td>monitor @com.dropbox.list_folder()</td>
</tr>
<tr>
<td>when I create a file in Dropbox</td>
<td>WP</td>
<td>monitor @com.dropbox.list_folder() on new file_name</td>
</tr>
<tr>
<td>the download URL of $x$</td>
<td>NP</td>
<td>( \lambda(x : \text{PathName}) \rightarrow @\text{com.dropbox.open(file_name = x)} )</td>
</tr>
<tr>
<td>a temporary link to $x$</td>
<td>VP</td>
<td>@com.dropbox.open(file_name = x)</td>
</tr>
<tr>
<td>download $x$</td>
<td>VP</td>
<td></td>
</tr>
</tbody>
</table>

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 verb phrase, noun phrase and when phrase respectively.
Semantic functions allow developers to write arbitrary code that computes the formal representation of the generated natural language, using the intermediate formal language components named between \( := \) and \( \rightarrow \). For example, semantic functions can apply incremental type-checking:

\[
\text{wp} := \text{when } q \rightarrow \text{change} \rightarrow \{
\begin{align*}
\text{if } q.\text{is\_monitorable} & \rightarrow \text{return } \text{monitor } q \\
\text{else} & \rightarrow \text{return } \perp
\end{align*}
\}
\]

Developers can also use semantic functions to apply restrictions that go beyond natural language category or formal language types; for example, in the template:

\[
\text{command} := \text{enumerate } q \rightarrow \{
\begin{align*}
\text{if } q.\text{is\_list} & \rightarrow \text{return } \text{now } \Rightarrow q \Rightarrow \text{notify} \\
\text{else} & \rightarrow \text{return } \perp
\end{align*}
\}
\]

the semantic function checks that \( q \) returns a list.

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. [55] only considered a small grammar and knowledge base, and in the domains it considers it is feasible to recursively enumerate all possible derivations, up to a certain depth of the derivation tree. Because the number of derivations is exponential in the depth of the tree, and grows quickly with the size of Thingpedia, full recursive enumeration is not feasible for ThingTalk. Instead, Genie uses a randomized synthesis algorithm, where only a subset of derivations produced by each construct template is preserved. The desired size is configurable, and decreases exponentially with increasing depth; this allows Genie to generate a sufficient number of programs of low depth, which provides broad coverage, while also generating some programs at high depth, which injects variance in the synthesized set and expands the set of recognized programs. Developers using Genie can control the sampling strategy by splitting or combining construct templates, using intermediate derivations.

### 3.2 Paraphrase Data

Sentences generated by templates are not representative of real human input, thus we use manual paraphrasing to generate more natural sounding sentences. However, manual paraphrasing is not only expensive, but it is also error-prone, especially when the generated commands are too complex and do not make sense to humans. Thus, Genie lets the developer choose a subset of synthesized sentences to paraphrase. We have also developed a crowdsourcing framework designed to improve the quality of paraphrasing.

**Choosing Sentences to Paraphrase.** It is important that developers get to control the subset of templates to paraphrase as well as their sampling rates. For example, while it is important to obtain some paraphrases for all primitives, we do not need to sample combinations of functions evenly. Our first priority is to use sentences that workers can understand and provide a high-quality paraphrase. Coverage is secondary because all synthesized data will also be included in the training dataset.

Compound commands putting together two unrelated functions confuse workers. 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.

Developers can also specify the values used in input parameters of the sentences so they sound natural to paraphrases. 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. Genie contains a website with the user interface for workers; the workers access the website to perform the paraphrasing task. The website’s user interface shows the synthesized sentence, any hint provided with it, and includes a text box to enter the paraphrase. Automatic validation is performed based on heuristic rules.

To increase variety, Genie prepares the crowdsource tasks so that multiple workers see the same synthesized sentence, and each worker is asked for multiple paraphrases for each synthesized sentence. For ThingTalk, asking two paraphrases for each synthesized sentence is optimal: people will have a hard time writing three different paraphrases, and conversely with only one they will tend to make the most obvious changes.

Even then, due to ambiguity in natural language, and the usual caveats of crowdsourcing (workers will try to cheat, they will try to get away with minimal work, they will not read the instructions), workers can make mistakes. Genie reduces the number of mistakes by asking other workers whether the prompt and the paraphrase have the same meaning; sentences which are not marked as equivalent are discarded.

### 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, 10, 13, 16, 18, 21, 30, 31, 39, 47, 60]. 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 fall-back for parameters that do not have a more specific type. 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 [17] to the paraphrases, to generate the augmented dataset.

3.4 Combining Synthesized and Paraphrase Data
Synthesized data is not only used to obtain paraphrases, but it is also fed to the model directly. During training, synthesized data and paraphrase data perform two different roles: 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 control how synthesized and paraphrase data are weighted at training time by controlling the expansion factor of each class of data — that is, how many copies of the same sentence with different parameters should be included in the training set. Furthermore, developers can generate different sets of synthesized data, with different flags, for training and for paraphrasing; this allows the developers to write templates that are correct but hard to understand by a non-expert, and allow to increase or decrease the proportion of certain programming language features (such as compound commands, filters, timers, etc.) in training by generating sets with those features enabled or disabled.

4 Neural Semantic Parsing Model
Genie’s semantic parser is based on Multi-Task Question Answering Network (MQAN), a previously-proposed model architecture that was found effective on a variety of NLP tasks [37] such as Machine Translation, Summarization, Semantic Parsing, and etc. MQAN frames all NLP tasks as contextual question-answering. A single model can be trained on multiple tasks (multi-task training [9]), 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 is first embedded by concatenating its pre-trained word and character embeddings, and then fed to the encoder to construct the context and questions representations. In the decoder, each answer is embedded by passing through a pre-trained language model layer. 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 [37] for more details.

4.2 ThingTalk Language Model
Genie applies a pre-trained recurrent Language Model (LM) [38, 46] to encode the answer before feeding to MQAN. The high-level model architecture is shown in Fig. 6. Previous work has shown that using supervised [36, 46] and un-supervised [43] 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. ThingTalk language model 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 [37], an open-source library. Preprocessing for tokenization and argument identification was performed using the CoreNLP library [35], and input words are embedded using pre-trained 300-dimensional GloVe [41] and 100-dimensional character n-gram embeddings [20]. The decoder embedding is pretrained using a 1-layer LSTM language model, provided by the floydhub open source library [15], and trained on a large synthesized set. Dropout [48] 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

In this section, we first introduce the datasets that we use to test and train Genie on ThingTalk. We then evaluate Genie with the full model and training set on ThingTalk, and perform error analysis and comparison with prior work. We compare Genie’s training strategy against baseline models which trained with only synthesized data and only paraphrase data, and finally perform an ablation study on language and model features.

We use a subset of the current Thingpedia as the ThingTalk skill library, corresponding to 131 functions and 178 distinct parameters, over 44 skills; this subset corresponds to the full Thingpedia when we started creating the training set.

5.1 Evaluation Data

Previous neural models trained on paraphrase data are only evaluated with paraphrase data [55]; furthermore, because each domain is small, all logical forms that appear in testing also appear in training. Testing on paraphrase data is undesirable because paraphrases may not represent real usage data. Moreover, the space of VAPL programs is so large that the user will not provide the same exact command that has been seen in training or even a paraphrase of it.

We used three methods to gather data that mimic real usage: from the developers, from users shown a cheatsheet, and IFTTT users. Because of the cost in acquiring such data, we managed to gather only altogether 1820 sentences, partitioned into 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.

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 putting 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.

5.2 Training Set

Using Genie, we synthesized 1,724,553 sentences, 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. The characteristic of this dataset is shown in Fig. 7; in the figure, primitive commands are commands that use one
To compare with previous work, we first test on paraphrased in the training set. To achieve good accuracy on data that has no representation performance of the model on realistic data. Yet, Genie is able paraphrase data, which can significantly over-estimate the underscores the difference, and the danger, of testing on sheet test, Genie achieves 62%, and on IFTTT 65%. This can generalize to compound commands not seen in training. Results are shown in Fig. 8; in which we report the average of 3 runs of training, as well as the minimum and maximum results. On the paraphrase test set, Genie obtains a program accuracy of 87%; this shows that the model trained by Genie can generalize to compound commands not seen in training.

Our 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 exact match of the canonicalized generated program to the target program. To account for ambiguity in the input sentence, we manually annotate ambiguous sentences with all programs that are a valid interpretation of the sentence. Results are shown in Fig. 8; in which we report the average of 3 runs of training, as well as the minimum and maximum results. On the paraphrase test set, Genie obtains a program accuracy of 87%; this shows that the model trained by Genie can generalize to compound commands not seen in training.

On validation data, Genie achieves 68%, while on the cheat-sheet test, Genie achieves 62%, and on IFTTT 65%. This underscores the difference, and the danger, of testing on paraphrase data, which can significantly over-estimate the performance of the model on realistic data. Yet, Genie is able to achieve good accuracy on data that has no representation in the training set.

Error analysis on the validation set reveals that Genie can produce a syntactically correct and type-correct program for 96% of the inputs. Genie identifies correctly whether the input is a primitive or a compound with 91% accuracy, and can identify the correct devices 87% of the times. The generated program contains the correct function for 82% of the inputs; this metric corresponds to the function accuracy introduced in previous work of IFTTT [44]. Finally, less than the 1% of the inputs have the correct functions, parameter names, and filters but copy the wrong parameter value from the input.

Note that, in the validation set, there are 1205 commands whose programs appear in the training set exactly (up to the parameter values between quotes), and 272 sentences whose programs are new. If we test only on programs seen during training, Genie achieves an accuracy of 77%. Conversely, if we test only on new programs, the accuracy drops to 30%; this suggests that future work is needed to enable the model to understand the compositionality of natural language.

5.4 Comparison to Previous Work

Our new design of ThingTalk improves the power of the assistant. In the original ThingTalk, Thingpedia supported 35 trigger functions and 25 retrieval functions; only 11 APIs had both trigger and retrieval functions. After our work, all query APIs include both trigger and retrieval ability, and multiple queries can be chained together.

The previous best result reported on ThingTalk was 51% [7]. This result was obtained on paraphrase test data, using a training set that mixed paraphrases with manually written primitive sentences, and with a model that requires quoting all parameters in the input sentence.

We can also compare the result on ThingTalk with previous work on generating IFTTT trigger-action rules. Because previous work started from high-level descriptions of the rules, and not precise virtual assistant commands, they did not succeed at recognizing the full program including parameters, and reported an accuracy of 3% [44].

5.5 Synthesized and Paraphrase Training Strategy

To evaluate our proposed training strategy, we run two more experiments: we train (1) with just data synthesized with the supplied templates, and (2) with just paraphrase data. Results are shown in Fig. 8.

We include training with just synthesized data to measure the difference between synthesized and paraphrased sentences. This is a valid question given synthesized data templates are refined using earlier paraphrases. Genie obtains 47% on the paraphrase test, 56% on the validation set, 54% on cheat-sheet data, and 51% on IFTTT. The results indicate that the paraphrased sentences are different enough from the synthesized data that training with just the latter is inadequate; yet, the templates have been tuned enough that they can deliver good results on realistic data.
The traditional methodology is to train with just paraphrased sentences, and doing so improves the accuracy on the paraphrase test (82%) compared to training with synthesized data. This result indicates that the model can fit the paraphrase data well, and learn to generalize to unseen paraphrases (including unseen combinations of functions). Yet, paraphrases are insufficient to achieve usable accuracy on the validation set (55%), the cheetsheet test set (47%), or the IFTTT test set (48%). The smaller size of the paraphrase set causes the model to overfit, even after data augmentation; overfitting is not a problem on paraphrase data, but the model fails to generalize on realistic data.

Our approach of combining synthesized and paraphrasing overcomes the overfitting, and it allows the model to generalize further. This result suggests that synthesized data are not just useful as inputs to paraphrasing, but can expand the training dataset inexpensively. The synthesized data teaches the neural model many combinations not seen in the paraphrases, and the paraphrases teach the model natural language usage.

We note that all models incur significantly higher variance in the test sets than in the validation or paraphrase set; this is again a sign of overfitting, as the model is fit to both the training set, which closely matches the paraphrase set, and to the validation set through early stopping.

### 5.6 VAPL and Model Design Space Exploration

To understand the contribution of each component of Genie and the various features of ThingTalk, we perform an ablation study, in which we remove one feature from Genie and ThingTalk at a time and compare against the full model. We evaluate on the validation set, on the paraphrase test set, and on the portion of the validation set that consists of combinations of functions, filters, and parameters not seen in training. Results are shown in Table 3; in the table, we report the average result of three models, and the error represents the half-range between minimum and maximum.

#### VAPL Features

We evaluate the contribution of canonicalization. To simulate a variant of ThingTalk that is not canonicalizable, and a training set that is annotated inconsistently, we train a model with a training set where keyword parameters are shuffled independently on each training example. The result (Table 3) shows that canonicalization to be the most important VAPL feature. A language without canonicalization, and inconsistently annotated datasets would see decreases in accuracy across the board, as the model struggles to fit the training data. Note that programs are canonicalized during evaluation, so this effect has to occur during training.

Our next experiment compares positional and keyword input parameters. Positional parameters decrease performance across the board, by 3% on paraphrases and 5% on new validation programs; this suggests that keyword parameters improve generalization to new programs.

Finally, we evaluate the role of type annotations, and we find they have no measurable effect on the accuracy (all differences are within the margin of error). Keyword names are an effective substitute for type annotations, as the parameter name mostly correlates with the type, and the language model can learn ThingTalk’s typing rules from data. Note that errors in the type annotations are ignored during evaluation, whether the model generates the annotations or not, as ThingTalk uses full type inference.

#### Model Features

To evaluate the role of parameter expansion, we compare with a training strategy where each sentence appears only once in the training set, rather than multiple times with different parameters; different input sentences still use different parameters. Results in Table 3 show that removing parameter expansion decreases performance by 9% on paraphrases, and increases the error range across the three experiments. We interpret this as a symptom of overfitting. The effect of parameter expansion is small on the validation data because this data is manually written and annotated, and only uses a small set of parameters that annotators could make up on the spot.

Finally, we evaluate the use of pretrained language model to embed programs during decoding, which our proposed modification to the MQAN model. We compare against a model that uses a randomly initialized and jointly trained

<table>
<thead>
<tr>
<th>Model</th>
<th>Paraphrase Validation New Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genie</td>
<td>87.1 ± 1.8</td>
</tr>
<tr>
<td>− canonicalization</td>
<td>80.0 ± 1.3</td>
</tr>
<tr>
<td>− keyword param.</td>
<td>84.0 ± 0.6</td>
</tr>
<tr>
<td>− type annotations</td>
<td>86.9 ± 3.6</td>
</tr>
<tr>
<td>− param. expansion</td>
<td>78.3 ± 4.8</td>
</tr>
<tr>
<td>− decoder LM</td>
<td>88.7 ± 1.0</td>
</tr>
</tbody>
</table>

Table 2. IFTTT dataset cleanup rules.
embedding matrix for the program tokens. Our model contribution appears to improve the performance primarily on new programs, where it leads to a 3%-improvement; we believe this is because the model is exposed to a larger space of programs. The model also improves on the overall validation set by 1%. On paraphrases, the use of the pretrained decoder language model has a small but detrimental effect; this is because the model can already learn effective token embeddings directly from the data, and can fit the paraphrase distribution well, even on unseen function combinations.

6 Case Studies of Genie

Here we show how we can use Genie to generate a natural parser for three very different use cases: a sophisticated music playing skill, a new language for policies modeled after ThingTalk, and an extension to ThingTalk for aggregates.

We compare a model trained using Genie against a baseline trained with only paraphrase data and without any data augmentation or parameter expansion; this baseline closely matches the methodology introduced by Wang et al. [55].

6.1 A Comprehensive Spotify Skill

The popular music service Spotify exposes many APIs that are inaccessible in today’s virtual assistants. Our first use case is to develop a Spotify skill service that supports combinations of Spotify primitives. The Spotify skill was created by two students in our university. The skill contains 15 queries and 17 actions. By using the ThingTalk syntax, this skill can support compound commands such as “add what is playing on the playlist”, by combining the primitives “get what’s playing” and “add songs to playlist”. Other useful compound sentences include “add all songs faster than 500 bpm to the playlist dance dance revolution”, and “wake me up at 8 am by playing wake me up inside by evanescence”.

This skill illustrates the importance of Genie’s ability to handle quote-free sentences. The original semantic parser for ThingTalk requires all arguments to be quoted and then replaced 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 students wrote 187 templates (5.8 per function on average), and through Genie we collected 1,553 paraphrases. We then used Genie to generate a dataset with 165,778 synthesized sentences, 205,923 paraphrases, and 11,335 augmented sentences.

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 at 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 76% accuracy, an improvement of 26% 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

For our next use case, we use Genie to build a semantic parser for primitive TACL policies. TACL is an access control policy language we introduced in our previous work [8]. The policy describes who, what, when, where, and how his data can be shared; it does so by combining a source, the person requesting access, and a ThingTalk command. For example, the policy “my secretary is allowed to see my work emails” is expressed as:

\[
\sigma = \text{"secretary"} : \text{now} \\
\Rightarrow @\com\gmail\inbox\ filter\ labels\ contains\ "work" \\
\Rightarrow \text{notify}
\]

Our previous work introduced a semantic parser for primitive TACL policies; the formal definition of the language targeted by that parser is shown in Fig. 10. That parser obtained an accuracy of 74%, and in that work, all parameters are expected to be quoted, which renders the result imper- ticular with spoken commands.

To apply Genie, we reuse the same dataset of policy commands, which consists of 4,742 paraphrases. We write 6 construct templates, and use Genie to generate the dataset; the result contains 543,566 policies, of which 432,511 are synthetic and the rest are paraphrase and augmented. All the quotes are removed from the original dataset.
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 63% without any iteration on templates (Fig. 9), an improvement of 28% 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 virtual assistant is based on the Alexa Meaning Representation Language (AMRL) [29], 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” [29]. 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 78% [42].

AMRL is not available to third-party developers. Third-party skills have access to a joint intent-classification and slot-tagging model [19], 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 [22] is a service that allows users to combine services into trigger-action rules. Previous work [44] attempted to translate the English description of IFTTT rules into executable code. Their method, and successive work using the IFTTT dataset [2, 5, 14, 34, 61], 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 the practical task of training 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,
crowdsourcing paraphrases them and then use the paraphrases as training data [55]. Su et al. [50] 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 [24] 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 feature, but it does so with a synchronous grammar that is automatically inferred from the language. Hence, their work requires an initial dataset to infer the grammar, and cannot be used to bootstrap a new formal language from scratch. Future work will investigate combining Genie with their technique, to offer a set of candidate constructs to the developer during template design and refinement.

A different line of work by Kang et al. [62] also considered the use of generative models to paraphrase the training data automatically; this is shown to increase accuracy. The challenge in this technique (and similar generative data augmentation techniques [25, 66]) is to ensure that each generated sentence has the same semantics as the one it is based from. 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, 23, 40, 52, 55, 57–59, 63–65, 67], instructions to robotic agents [11, 26, 27, 56], and trading card games [33, 45, 61].

Full SQL does not admit a canonical form, because query equivalence is undecidable [12, 53], so previous work on database queries have targeted restricted but useful subsets [67]. ThingTalk instead is designed to have a canonical form.

The state-of-the-art algorithm is sequence-to-sequence with attention [3, 14, 51], optionally extended with a copying mechanism [24], grammar structure [45, 61], and tree-based decoding [2]. In our experiments, we found that the use of grammar structure provided no additional benefit.

8 Conclusion

Virtual assistants that support compound commands can greatly simplify and enhance our lives, but no usable semantic parser has previously been developed due to the complexity of the commands. In this paper, we introduce Genie, a tool to help developers generate a semantic parser with high accuracy for a given virtual assistant formal language.

We introduce the concept of natural language templates to bridge the gap between natural language and formal language. By supplying primitive and construct templates in an iterative fashion, developers can use Genie to create a large high-quality dataset consisting of synthesized sentences and paraphrased sentences for training.

We also propose design principles for Virtual Assistant Programming Languages that can improve semantic parsing accuracy. We extend ThingTalk to follow these principles, and show that Genie delivers a semantic parser with 62% accuracy on realistic inputs, and 63% on IFTTT data. We also evaluate Genie on a sophisticated music skill, an extension of ThingTalk with aggregation support, and a new formal language for access control. The semantic parsers generated by Genie obtain at least 63% accuracy for all three tasks, and improve the baseline by at least 26%, which shows the power and generality of Genie.

References


