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

Giovanni Campagna  Silei Xu  Mehrad Moradshahi  Monica S. Lam
Computer Science Department
Stanford University
Stanford, CA
{gcampagn,silei,mehrad,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 created TT+, a superset of ThingTalk, a language used by the open-source Almond virtual assistant. Genie delivers a parser for TT+ that obtains an 87% accuracy on the paraphrased data set, a 22% increase over state-of-the-art models. In addition, we demonstrated easy extensibility with a music skill, aggregate functions, and access control; all three have program accuracy over 85%.

1 Introduction
Personal virtual assistants let users use natural language to manage their collection of accounts across all their internet services and IoT devices. This paper asks how we can design virtual assistants for extensibility. What should be the interface between natural language (NL) and the execution engine of the assistants? How can we easily extend virtual assistants with new devices and new constructs? More importantly, how can we teach virtual assistants to understand more commands in natural language easily?

Virtual assistants are commonly based on semantic parsing, a machine learning algorithm that converts natural language to a semantic representation in a formal language. The power, complexity and accuracy of semantic parsers depend on the target language of choice.

The Abstract Meaning Representation [5] and the Alexa Meaning Representation Language [28, 36] model natural language phenomena explicitly. They use machine learning to match NL to a representation model which closely matches the semantics of the sentences. 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. Similarly, the virtual assistant execution engine must also deal with the complexity of the language.

At the other end of the spectrum, the open-source Almond virtual assistant uses the formal ThingTalk programming language as an intermediate language; it uses a semantic parser to translate natural language to ThingTalk, which can directly be executed [8]. ThingTalk only has one construct which has three clauses: when some event happens, get some data, and perform some action, each of which can be predicted. 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 If-This-Then-That (IFTTT) [21], which has crowdsourced 250,000 unique compound commands. Fig. 1 shows how a natural-language sentence can
be translated into a ThingTalk program, using the set of services in Thingpedia. However, no usable semantic parser has been developed for this language so far. The Almond parser requires all parameters to be explicitly quoted, e.g., “search ‘pizza’ on Bing”, which is a nonstarter, especially for voice inputs. Furthermore, the accuracy reported was 71% if the commands contain one function, and 51% for two on a dataset based on manual paraphrasing of synthetically generated sentences [8].

1.1 Semantic Parsers for Virtual Assistants

This paper advocates the same execution-based approach used in ThingTalk. We coin the term VAPL (Virtual Assistant Programming Language) to refer to languages designed as a target for natural language translation by virtual assistants. The VAPL exposes the full capability of the virtual assistant to the deep-learning neural network, instead of introducing an intermediate representation that needs to be interfaced with by both the parser and the execution engine.

After experimenting with state-of-the-art machine-learning techniques to build a semantic parser for ThingTalk, we discovered several unique challenges to this problem. In the end, we created TT+, an extension to ThingTalk, which is amenable to natural language translation, and developed new methodologies and techniques to address the following challenges:

Canonical program representation. Semantic parsing has been used mainly for answering questions. Models in that domain can be tested by executing the target program under a test environment and comparing the output to check whether two programs are equivalent. This is unsuitable to any domain where programs have side effects, such as IoT commands. The output of the model has to be the program itself. VAPL programs must have a (unique) canonical form so we can check if the answer is correct syntactically.

Designing VAPL for amenability to NL translation. To make NL translation feasible, VAPL and its libraries need to be designed with the principles of keeping the semantics of components orthogonal, matching non-developers’ mental model, and supporting partial correctness in translation.

Programmed training data synthesis. Machine learning requires training data, which is not available for new languages. To create a semantic parser for question answering over simple domains, Wang et al. [47] 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 model that can match input sentences against possible canonical sentences. This approach is inadequate for virtual assistants, which use powerful constructs to connect many diverse domains, each with its own vocabulary. It is simply 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 introduce an NL-template language that allows developers to direct the generation of synthetic sentences. Developers are expected to iterate on their templates after observing the results of paraphrased sentences.

Compositional Neural Network. It is infeasible to collect paraphrases for all the sentences supported by a VAPL language. To the best of our knowledge, we are the first to show how to combine paraphrased and synthetic data to train an effective compositional semantic parser.

1.2 The Genie Toolkit

To facilitate rapid development of virtual assistant capabilities, we encapsulate the semantic parser methodology we developed in a toolkit called Genie. Given a definition of a VAPL language and a set of NL-templates, Genie generates a semantic parser that translates natural language to code for the new language. As shown in Figure 2, it creates a synthetic data set, crowdsources paraphrases, and augments the training data with parameter values. It trains a neural network model and outputs the parameters of the trained model, and in other words, the semantic parser.

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 TT+, and created a semantic parser that achieves an 87% accuracy on paraphrased data.
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. By adding synthetic data to paraphrased data in training, the use of max-margin, span-level copying, and parameter expanding, our neural model achieves an improvement in accuracy of 22.3%.
4. Demonstration of extensibility by the success of using Genie to generate effective semantic parsers for a Spotify skill, aggregate functions, and access control.

1.4 Paper Organization

The organization of the paper is as follows. Section 2 describes the VAPL principles, using TT+ as example. Section 3 introduces the Genie data acquisition pipeline and Section 4 describes Genie’s deep learning model. We present experimental results on understanding TT+ 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

This section discusses the principles that guide the design of Virtual Assistant Programming Languages (VAPL), using the TT+ language (a superset of ThingTalk) as an example. Our focus of this description is the rationale of the changes we made to ThingTalk to support natural language translation. TT+ is also the base language on which new VAPL languages can be defined using Genie.

2.1 Programming for Virtual Assistants

Virtual assistants are used by individuals to manage their web of personal Internet resources, which have already been configured and logged into. Like ThingTalk, TT+ supports that by allowing users to refer to them as their mail or their car without specifying account names and credentials.

Virtual assistants are also expected to grow when new devices or services are created. TT+ preserves the modularity of ThingTalk by having a simple construct that connects an extensible library of skills, which are high-level APIs for Internet service and IoT devices. The assistant can grow in its capability with new skills. In the rest of the section, we describe how the type system, library design, and construct supports this modularity.

At a high level, the design principles in support of virtual assistant programming can be summed up as (1) strong fine-grained typing to guide parsing and dataset generation, (2) skill library with semantically orthogonal component design to reduce ambiguity, (3) partial correctness support to provide the parser with partial reward for ambiguous sentences, (4) matching the user’s mental model to simplify the translation, and (5) canonicalization of VAPL programs.

2.2 Types and Constants

To improve the compositionality of the semantic parser, VAPLs should be statically typed, include fine-grain domain-specific types, and provide library developers with the ability to define custom types. The TT+ 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 TT+, 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.

2.3 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 ThingTalk, then present the syntax of TT+ classes and the design rationale.

Orthogonality in Function Types. When we ported over the Thingpedia library from ThingTalk to the TT+, we made a major change to the representation. 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.

In TT+, we collapse the distinction between triggers and retrievals into one class: queries, which can be monitored as event or used to get data. The TT+ 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 orthogonal, which is useful to users, crowdsourced paraphrase providers, and the neural model. Without this change, it will be very difficult to get correct paraphrases for training.

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 TT+, skills are IoT devices or a web service. The formal grammar of TT+ 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 for changes.

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 is 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.

**Partial Correctness Support.** It is important that the language be designed to support partial matching. TT+ classes support inheritance; for example, a specific brand of thermostat may inherit from the abstract thermostat class, with extra methods only available with that brand. This allows the parser to understand a generic phrase like "set the temperature" without tying it to a specific brand.

Where possible, developers should collapse multiple functions into one, and use parameters to distinguish between them. For example, Gmail has three different mail inboxes: regular, important and primary inbox. We might design three different queries "get_email", "get_important_email", "get_primary_email" to read them separately. Alternatively, we can combine them in a single "get_email" function, with an optional parameter to indicate the category of mail; if the user does not specify the kind of mail, the full inbox is assumed. The latter design has two advantages: (1) the neural network is not forced to make an immediate hard decision, and can defer the decision after choosing the function (and receiving positive reward for it), (2) if the sentence is ambiguous, the neural network can produce a partially correct answer. This improves training and reduces the negative effect of ambiguous sentences in the training set.

**2.4 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, TT+, like the original ThingTalk, is data focused and not control flow focused. TT+ 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 of TT+ is shown in Fig. 5.

An example of a TT+ program, with parameter passing, is shown in Fig. 1; the example is both a valid ThingTalk and TT+ program. TT+ programs can also use filters; for example, the following program automatically retweets all tweets by a specific user:

\[
\text{monitor}(@\text{com.twitter.timeline()} \text{ filter } \text{ author} = \text{@PLDI}) \Rightarrow @\text{com.twitter.retweet}(\text{tweet_id} = \text{tweet_id})
\]

The program uses the author output parameter of the function @com.twitter.timeline() 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":

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

To match the user’s mental model, instead of explicit looping constructs, TT+ uses implicit looping. 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 introduced by TT+. They generalize the trigger concept in ThingTalk to enable reacting to arbitrary changes in the data accessible by the virtual assistant. A stream can be the degenerate stream now, which triggers the program once immediately, it can be a timer, or it can be a monitor or 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:

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

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.

**Input and Output Parameters.** To allow translation from natural language, TT+ uses named parameters for inputs and outputs, rather than the more conventional positional parameters. When using keyword parameters, the semantic parser only needs to learn the partial signature of the functions, and not the length of the signature. To increase compositionality, we encourage developers to use the same naming conventions so the same parameter names are used for similar purposes.

**Parameter Passing.** TT+ simply uses declared parameter names to indicate parameter passing: output parameters can be named in subsequent method invocations to pass data from one method to the next. For example, in Fig. 1, the output parameter picture_url of the function @com.thecatapi.get() is named in the invocation to the function @com.facebook.post_picture(). The design avoids as much as possible the introduction of variables in the program, so that the user does not need to specify them exactly in their command, and the semantic parser only needs to learn the parameter names associated with each function.

In the case where two functions in the same command use the same parameter names, we resolve the conflict by assuming that the name refers to the rightmost instance. Here we consciously trade-off completeness for accuracy in the common case.

**Filters.** Output parameters can also be used in a boolean predicate applied to the output of each query to filter its output, using equality, comparison, string and array containment. Additionally, the filter can itself refer to an invocation of an arbitrary query function, followed by a boolean predicate on the outputs of that function.

### 2.5 Canonicalization of Programs

As described in Section 1, canonicalization is key to training a neural semantic parser. TT+ 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 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 generate synthetic data, (2) a crowdsourcing framework to collect paraphrases, and (3) a large corpus of values for parameters in programs.

3.1 Synthetic Data Generation

Previous research proposed using the formal grammar to generate possible programs, and semantic rules to translate the program into a canonical NL sentence, which is then shown to crowdsourced workers for paraphrases [47]. We found that this does not work for virtual assistant commands.

From a programming language point of view, TT+ 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, for example, the function “list_folder” in Dropbox, which returns a modified time. This is a different function from “open” in Dropbox, which has a different interface. It has been shown that this does not work for virtual assistant commands.

To address the difficulty in getting a training set for a new primitive, we can add the filter $modified\_time > start\_of\_week$.

We have developed an NL-template language so developers and skill writers can control synthetic data generation. We hypothesize that we can exploit compositionality in natural language to factor synthetic data generation 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 synthetic sentences that are understandable.

### 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:

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

which says that a derivation of non-terminal $lhs$ can be constructed by combining the literals and the non-terminals $rhs$, and then applying the semantic function $sf$ to compute the

<table>
<thead>
<tr>
<th>Natural language</th>
<th>Cat.</th>
<th>TT+ 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>$\lambda(x : \text{PathName}) \rightarrow @com.dropbox.list_folder(folder_name = x)$</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>$\lambda(x : \text{PathName}) \rightarrow @com.dropbox.open(file_name = x)$</td>
</tr>
<tr>
<td>a temporary link to $x$</td>
<td>NP</td>
<td>@com.dropbox.open(file_name = x)</td>
</tr>
<tr>
<td>open $x$</td>
<td>VP</td>
<td>@com.dropbox.open(file_name = x)</td>
</tr>
<tr>
<td>download $x$</td>
<td>VP</td>
<td>@com.dropbox.open(file_name = x)</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 TT+ skill library, with their grammar category. NP, WP, and VP refer to verb phrase, noun phrase and when phrase respectively.
formal language representation. For example, the following two construct templates define the two common ways to express "when - do" commands:

\[
\begin{align*}
\text{COMMAND} &::= \text{wp} : \text{wp} \rightarrow \text{return} \; s \Rightarrow a; \\
\text{COMMAND} &::= \text{a} : \text{vp} \rightarrow \text{return} \; s \Rightarrow a;
\end{align*}
\]

In combination with the following two primitive templates:

\[
\begin{align*}
\text{wp} &::= \text{`when I modify a file in Dropbox'} \rightarrow \text{monitor} \; @\text{com.dropbox.list_folder}() \\
\text{vp} &::= \text{`send a Slack message'} \rightarrow \text{@com.slack.send}()
\end{align*}
\]

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 language representation of the generated natural language, using the intermediate formal language components named between := and →. For example, semantic functions can be used to apply incremental type-checking:

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

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

\[
\begin{align*}
\text{COMMAND} &::= \text{'enumerate'} \; q : \text{np} \rightarrow \{ \\
\text{if} \; q.\text{is\_list} &\rightarrow \text{return} \; \text{now} \Rightarrow q \Rightarrow \text{notify} \\
\text{else} &\rightarrow \text{return} \; \bot
\}
\end{align*}
\]

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).

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, it is also error-prone, especially when the generated commands are too complex and do not make sense to humans. Thus, Genie uses as training data the full synthetic set plus paraphrases of a subset of synthetic sentences chosen by the developer. 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 the full set of synthetic data will also be included in training.

Compound commands putting together two unrelated functions confuse workers. Developers can provide a list of easy-to-understand functions and a list of 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 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 synthetic sentences to paraphrase, Genie produces a file that can be used to create a batch of crowdsourcing 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 synthetic sentence, any hint provided with it, and includes a text box for the worker to enter the paraphrase. Automatic validation is performed based on heuristic rules.

To increase variety, Genie prepares the crowdsourcing tasks so that multiple workers see the same synthetic sentence, and each worker is asked for multiple paraphrases for each synthetic sentence. For TT+, asking two paraphrases for each synthetic is the optimal equilibrium: 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 chance of mistakes by also performing manual validation, by asking other workers whether the prompt and the paraphrase have the same meaning; sentences who 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 free-form text, messages, emails, YouTube video titles and channel
names, news articles, Twitter and Instagram hashtags, people names, country names, currencies, etc. These corpora were collected from various resources on the Web and from previous academic efforts [2, 14, 16, 18, 29, 39, 52]. Overall, Genie’s database includes over 2.4 million distinct parameter values. Genie expands the synthetic and paraphrase data set 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.

4 Neural Semantic Parsing Model

4.1 Model

Our models are based on a sequence-to-sequence neural network [42] with attention [4, 32] and a copying mechanism. The architecture is similar to that of Jia and Liang [24], except the copying action is explicit rather than latent.

The sentence is encoded using word embeddings projected onto a low-dimensional space and passed into a bidirectional recurrent neural network. We then concatenate the forward and backward hidden layers to get the final hidden states:

\[
\begin{align*}
\tilde{x}_t &= W_x \operatorname{embed}(x_t) \\
 h_{E, \text{fw}, t} &= \text{RNN}(h_{E, \text{fw}, t-1}, \tilde{x}_t) \\
 h_{E, \text{bw}, t} &= \text{RNN}(h_{E, \text{bw}, t+1}, \tilde{x}_t) \\
 h_{E, t} &= h_{E, \text{fw}, t} || h_{E, \text{bw}, t}
\end{align*}
\]

where \(x_t\) is the \(t\)-th word, embed returns the pretrained word embedding for each token, \(h_{E, \text{fw}, t}\) and \(h_{E, \text{bw}, t}\) denote encoder hidden states at time \(t\) for forward and backward RNN layer respectively. \(\text{RNN}\) computes next hidden states given token embeddings and hidden states for current time \(t\), and \(\|\) denotes vector concatenation.

The encoding is then passed to a recurrent decoder; the recurrent decoder state is multiplied by each encoder state to produce the alignment score. Each alignment score is multiplied with the recurrent encoder state to produce a context vector, which is combined with the recurrent decoder state and fed into a one-layer feed forward network to produce the score of each parsing action. The model then greedily takes the action with the highest score (\(\tilde{y}_t\)), and feeds its embedding back into the decoder at the next step:

\[
\begin{align*}
\tilde{y}_0 &= \text{<start>} \\
 h_{D, 0} &= h_{E, T} \\
 h_{D, t} &= \text{RNN}(h_{D, t-1}, W_y \operatorname{embed}(\tilde{y}_{t-1})) \\
 s_{t, t'} &= \text{softmax}\left(\left(h_{D, t}\right)^T W_s h_{E, t'}\right) \\
 c_t &= \sum_{t' = 1}^{T} s_{t, t'} h_{E, t'}. \\
 \tilde{h}_{D, t} &= \tanh(W_c c_t + W_h h_{D, t}) \\
 \tilde{y}_t &= W_o \tilde{h}_{D, t} \\
 \hat{y}_t &= \text{arg max} \tilde{y}_t
\end{align*}
\]

where \(T\) is the length of the input sentence, \(<\text{start}>\) refers to the first action required to start the decoding, \(h_{D, t}\) is the decoder hidden state at time \(t\), \(s_{t, t'}\) is the attention score between decoder hidden state and encoder hidden state at time \(t\) and \(t'\) respectively. \(c_t\) refers to the context vector, and \(W_s, W_y, W_o, W_e, W_h, W_o\) are all learned weights.

A parsing action emits a terminal token in the target formal language (such as "now" or "). At training time, the sequence of terminal tokens is generated by the tokenizer of the formal language. In this model, there is a single fixed space from which all actions are taken, which is appropriate for TT+ because the library of supported functions is defined ahead of time.

4.2 Loss Function

Whereas most neural models use cross-entropy loss [23], our models are trained with max-margin loss (inspired by [13]):

\[
-\sum_{t=1}^{T} \max \left(\tilde{y}_t, y_t - 1 [\hat{y}_t = y_t] + 1\right) - \hat{y}_t, y_t
\]

where \(y_t\) is the \(T\)-sized vector of gold actions, \(\tilde{y}_t\) is the predicted action at time \(t\), and \(\hat{y}_t\) is the vector of prediction scores at time \(t\). The \(\hat{y}_t\) vector is computed by the decoder network, and its size is the number of possible terminals in the grammar. \(1\) is the indicator function.

This corresponds to considering each prediction as an independent multi-class linear classifier, whose features are computed by the decoder recurrent network and attention mechanism, and whose classes are the grammar tokens.

We hypothesize that max-margin is more robust in handling the imbalance between training and test distribution, compared to standard cross-entropy loss. The intuition is that, similar to Support Vector Machines [12], for linearly separable data, only the data points closest to the separation hyper-plane affect the prediction; the distribution of other points in the same class does not matter.

4.3 Copying Mechanism

To support arbitrary and out-of-vocabulary parameters in the generated programs, parameters of string or entity types are translated with a copying mechanism.

Following Yin and Neubig [53], we leverage the definition of the formal language to choose whether to copy or emit a fixed token. Concretely, the space of terminal tokens in the target language is extended with the terminal SPAN. Upon decoding SPAN, a span of words is copied from the input and inserted into the final program using a pointer network [46]. The network computes the similarity between the embedding of a word in the input sentence and the current decoder state, and then chooses the most similar word:

\[
\begin{align*}
 p_{\text{begin}, t} &= \text{arg max}_{t'} \left(h_{D, t}\right)^T W_{\text{begin}} h_{E, t'} \\
 p_{\text{end}, t} &= \text{arg max}_{t'} \left(h_{D, t}\right)^T W_{\text{end}} h_{E, t'}
\end{align*}
\]
We use Genie to generate a dataset with 547,676 synthetic sentences, 24,451 paraphrase sentences, and 15,433 sentences obtained through PPDB augmentation. The characteristic of this dataset is shown in Fig. 6. 

### Table 2. Accuracy of the Genie model and the seq2seq model trained on just synthetic data, just paraphrase data, or both.

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Model</th>
<th>Function (%)</th>
<th>Program (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>Seq2Seq</td>
<td>65.4</td>
<td>46.5</td>
</tr>
<tr>
<td></td>
<td>Genie</td>
<td>63.8</td>
<td>46.0</td>
</tr>
<tr>
<td>Paraphrase</td>
<td>Seq2Seq</td>
<td>79.9</td>
<td>64.4</td>
</tr>
<tr>
<td></td>
<td>Genie</td>
<td>85.6</td>
<td>75.0</td>
</tr>
<tr>
<td>Combined</td>
<td>Seq2Seq</td>
<td>92.1</td>
<td>80.9</td>
</tr>
<tr>
<td></td>
<td>Genie</td>
<td>95.0</td>
<td>86.7</td>
</tr>
</tbody>
</table>

where $W_{\text{begin}}$ and $W_{\text{end}}$ are learned weights. The model then copies the words indexed by $p_{\text{begin},t} \ldots p_{\text{end},t}$ from the input.

Unlike previous work [24, 53] that only copies one word at a time, our model is able to copy entire spans in a single step, by pointing to the first and to the last word of the span. The pointers are trained with standard cross-entropy loss.

The copying mechanism is only applied to free-form string parameters and to custom types in the library. For types such as URLs, emails or phone numbers, we use argument identification [15], a technique by which the parameter is first identified, using a general named entity recognition model, and then replaced with a single token. This allows us to leverage existing knowledge in named entity recognition, and reduce the amount of training data needed.

### 5 Experimentation

#### 5.1 Implementation

Genie’s neural network model was implemented using Tensorflow [1] and the Tensor2Tensor library [45]. Preprocessing for tokenization and argument identification was performed using the CoreNLP library [33], and input words are embedded using pre-trained GloVe embeddings [35]. Our model is a 2-layer LSTM [20] with a hidden size of 128; Dropout [40] is applied between LSTM layers. Hyperparameters were tuned on the validation set. All our models are trained to perform the same number of parameter updates (250,000), using the Adam optimizer [27] which takes about 6 hours on a single GPU machine.

#### 5.2 TT+ Dataset

We use Genie to generate a dataset with 547,676 synthetic sentences, 24,451 paraphrase sentences, and 15,433 sentences obtained through PPDB augmentation. The characteristic of this dataset is shown in Fig. 6; in the figure, primitive commands are commands that use one function, while compound commands use two. The commands span a subset of the current TT+ skill library, corresponding to 131 functions and 178 distinct parameters, over 44 skills; this subset corresponded to the full library when the dataset was created. Each command uses one or two functions from the library. Sentences in the dataset use 2,795 distinct words; the number increases to 157,360 after parameter expansion. The dataset will be released upon publication.

### 5.3 Synthetic and paraphrase training strategy

To handle the large number of combinations expected for virtual assistants, our neural model need to be compositional, capable of parsing commands combining functions in ways different from those in training. We evaluate how our neural model can handle a test set showing unique combinations of functions, irrespective of parameter values, from those in the training set. Our test and validation sets contain 1,017 and 914 such paraphrased sentences, respectively. The training set, after parameter expansion, contains 2,941,858 sentences. Note that all paraphrases in the TT+ dataset are obtained after the last iteration of template creation is performed.

To evaluate our proposed training strategy, we run three sets of experiments, all on the same test set of paraphrases: (1) trained with just data synthesized with the supplied templates, (2) trained with just paraphrased data, (3) trained with a combination of both synthesized and paraphrased data. For each training strategy, we test two models to evaluate the effectiveness of our proposed modifications to the language and the models:

- seq2seq: a state-of-the-art sequence-to-sequence model with cross-entropy loss and word-level copying on TT+, but with positional parameters instead of keywords.
- Genie, which is seq2seq with span-level copying, and max-margin loss, on TT+.

Our ultimate metric is program accuracy, which considers the result to be correct only if the output is an executable program with correct functions, parameters, and filters. For comparison with previous work on IFTTT [37], we also evaluate the function accuracy, which measures only if the functions are correct. The results are shown in Table 2. We include training with just synthesized data, not because it is a plausible strategy, but because it indicates the difference between synthetic and paraphrased sentences. This is a valid question given synthetic data templates are refined using earlier paraphrases. Both seq2seq and Genie provide similar results, around 46%. The results indicate that the paraphrased sentences obtained with this methodology are different enough from the synthetic data that training with just the latter is inadequate. We want to note that without template refinement, the result would have been much worse.

The traditional methodology is to train with just paraphrased sentences. The paraphrase set, while smaller in size,
delivers improved result over just synthetic data, with a program accuracy of 64% by seq2seq and 75% by Genie.

Our approach of combining synthetic and paraphrasing data pays off, with seq2seq and Genie achieving 81% and 87%, respectively. This result suggests that synthesized data are not just useful as inputs to paraphrasing, but can be used to expand the training data set inexpensively. The synthetic data teaches the neural model many combinations not seen in the paraphrases, and the paraphrases teach the model natural language usage. The increase in data lessens the advantage of Genie over seq2seq, but there is still a 6% difference.

The previous best result reported on ThingTalk was 51% [8], which was obtained with a training set that mixed paraphrases with manually written primitive sentences. It is instructive to see that even just the synthetic set generated by Genie is almost equivalent to the previous best model for ThingTalk - and in fact better, if you consider that the previous model required quoting and marking parameters, while Genie's model can identify free-form parameters. The reason of this result lies in (1) the design of TT+ and the new skill library, which collapses the triggers and retrievals in Thingpedia into a single query class; (2) the power of the natural language templates, which allows the developer to write constructs that match the user's way of expressing the command, and then generalize them across all the library; (3) careful paraphrasing and improved machine learning. It allows Genie to deliver the first usable results for compound virtual assistant commands, with a program accuracy of 87%, and a function accuracy of 95%.

It is also instructive to compare the result on TT+ 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% [37]. Yet, even when we compare the function accuracy, the best reported result (an ensemble of 10 models, tested on the high quality subset of the test set) is 87.5% [31]. This result is lower than TT+, which confirms that it is not enough for the test set to be precise, if the training set is noisy.

Error analysis on the Genie model reveals that the major cause of errors is wrong parameter names, filters and operators, which account for 56% of the errors. 6% of the errors are caused by the copying mechanism choosing the wrong span of the input sentence. 5% of the errors are due to mistaking a primitive for a compound (or vice versa), 8% of the errors are caused by an incorrect device, and 20% by choosing the incorrect function. Finally, only 4% of the errors are caused by producing a syntactically incorrect program.

### 5.4 Ablation Studies

To understand the contribution of each component of Genie, we perform an ablation study, in which we remove one feature from Genie at a time and compare against the full model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Function (%)</th>
<th>Program (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genie</td>
<td>91.6</td>
<td>86.2</td>
</tr>
<tr>
<td>− keyword param.</td>
<td>92.0</td>
<td>84.1</td>
</tr>
<tr>
<td>− span copying</td>
<td><strong>92.5</strong></td>
<td>85.1</td>
</tr>
<tr>
<td>− max-margin</td>
<td>91.1</td>
<td>83.8</td>
</tr>
<tr>
<td>− param. expansion</td>
<td>90.7</td>
<td>82.1</td>
</tr>
</tbody>
</table>

**Table 3.** Accuracy results for the ablation study. Results are from the validation set of the New Combination task. Each “−” row removes one feature from Genie independently.

We evaluate on the development set, as the results inform the selection of the model, and using the test set would risk overfitting. We evaluate the following models:

- “− keyword parameters”: a variant of TT+ that uses positional rather than keyword input parameters (output parameters and join combinations are unaffected).
- “− span copying”: a model with word-level copying (similar to Jia and Liang [24]) rather than span-level.
- “− max-margin”: a model trained with standard cross-entropy loss rather than max-margin.
- “− parameter expansion”: a variant of the dataset where each paraphrase or synthetic sentence appears only once in the training set, rather than multiple times with different parameters; different input sentences still use different parameters.

Results are shown in Table 3. The largest drop in performance (4%) occurs without parameter expansion: this shows that data augmentation technique is effective in making better use of the limited data. A significant drop also occurs when training with max-margin (3%), which validates our hypothesis that the max-margin loss is more robust to training-testing skew, and allows us to mix synthetic and augmented data in the training set. Finally, the use of keyword parameters and span copying provides a small but notable effect (2% and 1% respectively); this is interesting as these changes are training tricks that have zero cost.

### 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 language extension to TT+ for aggregates, and a new VAPL language for policies modeled after TT+.

#### 6.1 The Most 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, created not by one of the authors in this paper, contains 15 queries and 17 actions. By using the TT+ 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 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.

Genie generates a dataset with 30,980 synthetic sentences, 28,810 paraphrases, and 11,770 augmented sentences. We sample 2,415 paraphrase sentences as the test set, 2,415 as validation set, and train the model with the rest of the data. Unlike in previous experiments, since the parameter value is meaningful in identifying the function, we use multiple instances of the same paraphrase with different parameters.

Genie achieves an 85% program accuracy as shown in Fig. 7. The seq2seq model achieves 75.5%. Parameter expansion is mainly responsible for the improvement because it is critical to identify the song or artist in the sentences. This suggests that Genie can be used by skill developers wishing to leverage compound commands.

6.2 Adding Aggregation to TT+

One of our goals in building Genie is to grow virtual assistants’ ability to understand more commands. Our next case study attempts to add aggregation, finding min, max, sum, average, etc, to TT+. Our new language TT+A extends TT+ with the following grammar:

Query $q$:

- $\text{agg [max | min | sum | avg] pn of (q) | agg count of (q)}$

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

```
now \Rightarrow \text{agg \ sum file size of (@com.dropbox.list_folder())}
\Rightarrow \text{notify}
```

Even though this query can be used as a clause in a compound command, we only train and test the neural network with aggregation on primitive queries. It is hard for users to compose long sentences with many clauses, we expect additions will be made in the future to teach virtual assistants how to chain the result from one sentence to another. We wrote 6 templates for this language.

The TT+ 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), so all possible combinations of aggregations and query functions are covered exhaustively by the 33,437 synthetic sentences generated. We collect 2,421 paraphrases of aggregation commands. For training, we add to the full TT+ dataset 205,905 aggregation sentences: 19,460 are expanded paraphrases and the rest are expanded synthetic and augmented. We then test on a paraphrase set of 250 aggregation commands. The Genie achieves a program accuracy of 89.6% without any manual tuning (Fig. 7). This suggests that Genie can support extending the language ability of virtual assistants effectively.

6.3 TTACL: TT+ Access Control Language

Campagna et al. [9] showed fine-grain sharing of one’s resources can be achieved by instructing one’s virtual assistant who, what, when, where, and how his data can be shared. They introduce TACL (Thing Access Control Language) as a control policy language for ThingTalk. They report that their semantic parser delivers an accuracy of 74%, but no formal description of their model is given. Moreover, they expect all inputs to have their parameters quoted, which would render the result meaningless in practice, especially with spoken commands.

For our last use case in this paper, we use Genie to add access control to TT+ to create TTACL. We add a source, the person requesting access, to primitive TT+ commands; the source can be used as parameters as well as filters. For example, the command “my secretary is allowed to see my work emails” is expressed as:

```
\sigma = "secretary": now
\Rightarrow @com.gmail.inbox filter labels contains "work"
\Rightarrow notify
```

The formal definition of TTACL, shown in Fig. 8, shares the same scope as the model built by Campagna et al. [9].

We use the same dataset of policy commands used by Campagna et al. [9], which consists of 4,742 paraphrases of primitive policies and combine it with 39,972 synthetic and augmented policy sentences. All the quotes are removed from
the original dataset, because Genie does not need parameters to be quoted. We split the dataset in 43,311 sentences for training, 701 for validation, and 702 for testing; as before, the test consists exclusively of paraphrases unique to the whole set, even when ignoring the parameter values. The training set was expanded to 318,361 sentences by replacing parameters with sampled values from the parameter lists supplied with corresponding library functions. Whereas previous work reports 74% accuracy on quoted inputs, Genie can parse unquoted inputs, a considerably harder task, with an even higher accuracy of 89.7% (Fig. 7).

7 Related Work

Almond. The original publication of the Almond virtual assistant and ThingTalk language [8] used classic semantic parsing to understand compound commands, based on the SEMPRE algorithm [34, 47], and reported an accuracy of 71% for primitive commands and 51% for compounds. The dataset was never released, so it is not possible to compare directly. As discussed in Section 5.3, we improve their result by refining the ThingTalk language into TT+, and by collecting an improved, larger and more varied dataset.

Alexa. The Alexa virtual assistant is based on the Alexa Meaning Representation Language [28], a language they designed to support semantic parsing of Alexa commands.

Alexa MRL has been developed on a closed ontology of 17 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% [36].

Alexa MRL 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 TT+ action with parameters. The full power of TT+ and Genie is instead available to all contributors to the library.

IFTTT. If-This-Then-That [21] is a service that allows users to combine services into trigger-action rules. Previous work [37] attempted to translate the English description of IFTTT rules into executable code. Their method, and successive work using the IFTTT dataset [3, 6, 15, 31, 53], 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, crowdsource paraphrases them and then use the paraphrases as training data [47]. Su et al. [41] 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 context-free grammar that is automatically inferred from language. Hence, their work requires an initial dataset to infer the grammar, and cannot be used to bootstrap a parser for a new formal language from scratch. Future work will investigate the possibility of combining Genie with their technique, potentially to offer a set of candidate constructs to the developer to choose from during the initial phase of template design and refinement.

Semantic parsing. Genie’s model is based upon previous work in semantic parsing, which was used for queries [7, 22, 34, 43, 47, 49–51, 54–57], instructions to robotic agents [10, 25, 26, 48], and trading card games [30, 38, 53].

Full SQL does not admit a canonical form, because query equivalence is undecidable [11, 44], so previous work on database queries have targeted restricted but useful subsets [57]. TT+ instead is designed to have a canonical form.

The state-of-the-art algorithm is sequence-to-sequence decoding [3]. In our experiments, we found that the use of grammar structure provided no additional benefit to TT+.

8 Conclusion

Compositional virtual assistants 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 toolkit 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 synthetic sentences and paraphrase sentences for training.

We also propose design principles for Virtual Assistant Programming Languages that can improve semantic parsing accuracy. We propose TT+ as an example and show that Genie delivers a semantic parser with 87% accuracy. We also evaluate Genie on a sophisticated music skill, an extension of TT+ with aggregation support, and a new formal language for access control. The semantic parsers generated by Genie obtain at least 85% accuracy for all three tasks, which shows the power and generality of Genie.
References


