Virtual assistants such as IFTTT and Almond support complex tasks that combine open web APIs for devices and web services. In this work, we explore semantic parsing to understand natural language commands for these tasks and their compositions. We present the ThingTalk dataset, which consists of 22,362 commands, corresponding to 2,681 distinct programs in ThingTalk, a language for compound virtual assistant tasks. To improve compositionality of multiple APIs, we propose SEQ2TT, a Seq2Seq extension using a bottom-up encoding of grammar productions for programs and a max-margin loss. On the ThingTalk dataset, SEQ2TT obtains 84% accuracy on trained programs and 67% on unseen combinations, an improvement of 12% over a basic sequence-to-sequence model with attention.
ata (2016)’s sequence-to-sequence neural network with attention. To improve compositionality so as to handle a large, extensible repository of skills, we represent the target programming language as a sequence of grammar productions. We experiment with both top-down and bottom-up grammar productions, and find that bottom-up supports extensibility better. We teach the model about the ThingTalk programming constructs by generating a large synthetic set of combinations as training data. We use a max-margin loss function instead of the commonly used cross-entropy loss to be more resilient to an artificially generated distribution in this synthetic set.

We show that our model can learn to generate unseen combinations with an accuracy that is 12% higher than a basic sequence-to-sequence model with attention.

The models, datasets and code will be released upon publication.

2 Related Work

The body of previous work in semantic parsing is abundant, in domains such as database queries (Zelle and Mooney, 1994, 1996; Tang and Mooney, 2001; Zettlemoyer and Collins, 2005; Wong and Mooney, 2007; Berant and Liang, 2014; Pasupat and Liang, 2015; Wang et al., 2015; Xiao et al., 2016; Zhong et al., 2017), instructions to robotic agents (Kate et al., 2005; Kate and Mooney, 2006; Wong and Mooney, 2006; Chen and Mooney, 2011), and trading card games (Ling et al., 2016; Yin and Neubig, 2017).

Previous work (Quirk et al., 2015) attempted to translate the English description of IFTTT rules into executable code. They found that the descriptions are too high-level and are not precise commands. Their method, and successive work using the IFTTT dataset (Beltagy and Quirk, 2016; Dong and Lapata, 2016; Liu et al., 2016; Yin and Neubig, 2017; Alvarez-Melis and Jaakkola, 2017), showed moderate success in identifying the correct functions on a filtered set of unambiguous sentences, but failed to identify the full programs with parameters.

Previous work in the Almond virtual assistant used semantic parsing to understand compound commands (Campagna et al., 2017). They used the SEMPRE algorithm (Wang et al., 2015), and reported an accuracy of 71% for primitive commands and 51% for compounds. Their work does not analyze the results, and does not discuss generalization. We also note that both the ThingTalk language and the Thingpedia repository have changed since the results were published.

The state of the art in semantic parsing uses the sequence-to-sequence algorithm (Dong and Lapata, 2016; Liu et al., 2016; Jia and Liang, 2016). In this algorithm, the input sentence is encoded with a recurrent neural network, then the parser produces a sequence of actions (which could be emitting a token, copying a word from the input, or choosing a grammar production) using a recurrent decoder with attention.

Previous work (Yin and Neubig, 2017; Rabinovich et al., 2017) explored the use of grammar structure in sequence-to-sequence parsing, predicting AST nodes. They encode the sentence in a top-down fashion, and use a custom architecture with special connections for the different nodes. In this work, we instead represent the grammar structure without modifying the sequence-to-sequence model. This is a simpler solution, and reduces training and implementation time.

Sequence-to-sequence for semantic parsing is a subset of the more general field of neural structured prediction, in which a recurrent decoder is trained to produce sequences with dependencies. This has been applied to machine translation (Sutskever et al., 2014; Bahdanau et al., 2014), text summarization (Chopra et al., 2016), syntactic parsing (Vinyals et al., 2015).

Sequence-to-sequence networks are commonly trained with maximum-likelihood (cross-entropy) loss. Previous work explored loss functions more suitable to a search strategy (Wiseman and Rush, 2016; Daumé et al., 2009), and the use of reinforcement learning (Norouzi et al., 2016; Ranzato et al., 2015; Bahdanau et al., 2016). In this paper, we explore a loss that is more robust than cross-entropy, but also preserves the fast convergence.

For semantic parsing models with strong supervision, the dominant technique to acquire data is the paraphrasing technique, introduced by Wang et al. (2015). In this technique, canonical sentences are sampled from a grammar, together with their supervision; paraphrases are then crowdsourced to acquire the real inputs to train on. Su et al. (2017) explore the concept of acquiring paraphrases for Web API with different sampling techniques. Their work focuses on 2 APIs; our work explores the full generality of Thingpedia.
3 Problem Statement

The goal of SEQ2TT is to automatically translate natural language sentences into executable ThingTalk programs, constructed out of APIs in the Thingpedia repository. SEQ2TT must choose the appropriate APIs and then generate the correct parameters and filters.

ThingTalk programs have three clauses:

\[
\text{when } \Rightarrow \text{get } \Rightarrow \text{do}
\]

Each clause invokes a function in Thingpedia. WHEN specifies the time or event that triggers the operation, GET (optional) retrieves data and DO performs a side effect. WHEN defaults to \text{now} which indicates the execution should take place once, now; the keyword \text{monitor} indicates that the program reacts to changes in the data. DO defaults to \text{notify}, meaning that the result will be returned to the user. One or more parameters can be passed between the functions. Results can be filtered with one or more predicates (based on equality, comparison, or containment). The full definition of ThingTalk is omitted due to space.

4 ThingTalk Dataset

4.1 Natural-Language Primitive Utterances

Contributors to Thingpedia are responsible for supplying natural-language utterances for each function. A single Thingpedia function can be used for different purposes depending on the parameters and filters applied to it. For example, the \text{@com.dropbox.list_folder} function can be used to list files in different orders, filter on size or modification time, and react to file creation and modification (Fig. 2). This shows that semantic parsers for ThingTalk must handle parameters and filters right, and not just identify the right functions.

4.2 Synthetic Programs

The space of ThingTalk programs grows quickly with the number of functions, because different parameters, filters, operators can be used for a single function and multiple functions can be combined with different parameter passing.

To get a good coverage of the program space, we generated 164,584 different programs by randomly applying the ThingTalk grammar on top of Thingpedia functions. As shown in the first column of Fig. 3, there are 234 possible primitive programs with no input parameters: there are 57 DO functions, the 66 GET functions can be used in the WHEN or GET clause, and some GET functions have several variants to monitor different parameters. We combine primitives to derive 32,633 compound programs in which only constant parameters are used. We add parameter passing and filters by sampling them to create more complex programs. In total, we generated 145,194 compound programs, where 62% of them involve two functions and the rest have three.

4.3 Synthetic Sentences

For each of the programs we generated, we construct the corresponding synthetic sentences using the natural language utterances of functions provided by the developers. The utterances are combined via a set of templates, which consists of 56 rules for the main \text{WHEN-GET-DO} construct and 27 rules for filters, to produce diverse while reasonably natural sentences (Fig. 4(a)). The details of this generation are omitted due to space. To further increase the lexical variety of the sentences, we randomly substitute some of the words in the generated sentences using PPDB (Ganitkevitch et al., 2013). Overall, 1,017,784 sentences are generated, and we refer to this dataset as the Synthetic set.

4.4 Paraphrasing

To obtain more natural and diverse sentences, we sample 4000 sentences from the Synthetic set and ask Mechanical Turk workers to paraphrase them (Fig. 4(b)). We limit paraphrasing to sentences with at most two functions, because synthetic sentences with three functions are hard to understand. To improve the quality of the sentences, only

<table>
<thead>
<tr>
<th>Natural language</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>my Dropbox files</td>
<td>@com.dropbox.list_folder()</td>
</tr>
<tr>
<td>my Dropbox files in alphabetical order</td>
<td>@com.dropbox.list_folder(order_by = name, increasing)</td>
</tr>
<tr>
<td>my Dropbox files that changed most recently</td>
<td>@com.dropbox.list_folder(order_by = modified_time_decreasing)</td>
</tr>
<tr>
<td>my Dropbox files that changed this week</td>
<td>@com.dropbox.list_folder(order_by = modified_time_decreasing) filter modified_time ≥ start_of_week</td>
</tr>
<tr>
<td>my Dropbox files larger than file size</td>
<td>@com.dropbox.list_folder(filter file size ≥ file size)</td>
</tr>
<tr>
<td>files in my Dropbox folder $folder</td>
<td>@com.dropbox.list_folder(folder $folder)</td>
</tr>
<tr>
<td>when I modify a file in Dropbox</td>
<td>@com.dropbox.list_folder() monitor when I modify a file in Dropbox</td>
</tr>
<tr>
<td>when I create a file in Dropbox</td>
<td>@com.dropbox.list_folder() monitor when I create a file in Dropbox</td>
</tr>
</tbody>
</table>

Figure 2: Examples of developer-supplied annotations for the \text{@com.dropbox.list_folder} function.
Complexity | Synthetic set | Paraphrase set | Augmented set
--- | --- | --- | ---
Primitives | Programs | Sentences | Programs | Sentences | Programs | Sentences
238 | 15,161 | 96 | 1,860 | 89 | 1,156
+ 1 parameter | 924 | 65,467 | 164 | 2,283 | 124 | 1,113
+ ≥ 2 parameters | 922 | 6,878 | 114 | 899 | 46 | 329
+ 1 filter | 11,601 | 158,835 | 1,378 | 9,600 | 561 | 3,734
+ ≥ 2 filters | 3,317 | 23,127 | 20 | 147 | 8 | 36
Compounds | Programs | Sentences | Programs | Sentences | Programs | Sentences
30,562 | 184,493 | 851 | 5,369 | 928 | 4,863
+ parameter passing | 6,068 | 118,021 | 121 | 722 | 142 | 742
+ 1 filter | 20,091 | 89,038 | 555 | 3,117 | 606 | 2,949
+ ≥ 2 filters | 3,549 | 9,600 | 86 | 431 | 100 | 425
+ filter & parameter passing | 2,249 | 23,958 | 25 | 138 | 30 | 131
Total | 79,521 | 694,961 | 3,410 | 24,566 | 2,634 | 15,478

Figure 3: Characteristics of the ThingTalk Dataset

Figure 4: Data collection flow.

workers with 99% approval rate are employed. Each sampled sentence is shown to 3 workers, and each worker is asked to provide 2 paraphrases for each prompt. The paraphrases are accepted only if three other workers agree that they retain the original meaning. We also check and reject sentences automatically if numbers, quotes, emails, usernames, or hashtags are removed.

Through this method, we collected 22,362 paraphrase sentences over 2,681 programs (middle two columns in Fig. 3). As in the case with the Synthetic set, we adopt PPDB to randomly substitute words in the paraphrases and collect another 15,478 sentences. We refer this set of data as the Augmented set (last two columns in Fig. 3).

Even though the Paraphrase set is significantly smaller than the Synthetic set, it has more lexical variety even without the help of PPDB. Fig. 5 shows the number of distinct words for each function, in primitive sentences of the Synthetic and the Paraphrase sets. On average, paraphrasing increases the lexical variety by 92%.

5 Model

The SEQ2TT parser is based on a sequence-to-sequence neural network (Sutskever et al., 2014) with attention (Bahdanau et al., 2014; Luong et al., 2015).

1. **Input:** get a cat picture and post it on Facebook with caption “Funny Cat!”
2. **Preprocess:** `<s>` get a cat picture and post it on facebook with caption QUOTED_STRING_0 `</s>`
3. **Grammar productions:**
   - `$prim_get -> @com.thecatapi.get`
   - `$get -> $prim_get`
   - `$do -> @com.facebook.post_picture`
   - `$constant_String -> QUOTED_STRING_0`
   - `$const_param -> p:caption:String = constant_string`
   - `$do -> $do $const_param`
   - `$input -> now => $get => $do $out_param_Picture`' p:picture_url:Picture = $out_param_Picture
   - `$input -> $input on $param_passing accept`
4. **Program:**
   - `now => @com.thecatapi.get => @com.facebook.post_picture p:caption:String = QUOTED_STRING_0 on p:picture_url:Picture = p:picture_url:Picture`

Figure 5: Unique words for each function, in the Synthetic and Paraphrase sets.

Figure 6: The parsing process for the example in Fig. 1.
The model is similar to that of Dong and Lapata (2016), except for the use of a bidirectional encoder.

The sentence is encoded using word embeddings, which are projected into a low-dimensional space and passed into a bidirectional recurrent neural network:

\[
\hat{x}_t = W_x \text{embed}(x_t) \\
h_{E,fw,t} = \text{RNN}(h_{E,t-1}, \hat{x}_t) \\
h_{E,bw,t} = \text{RNN}(h_{E,t+1}, \hat{x}_t) \\
h_{E,t} = h_{E,fw,t}||h_{E,bw,t}
\]

where \( T \) is the length of the sentence, \( x_t \) is the index of the \( t \)-th word in the sentence.

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 parser then greedily takes the action with the highest score, and feeds an embedding of that action back into the decoder at the next step. Formally:

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

Our model uses a single layer LSTM recurrent cell (Hochreiter and Schmidhuber, 1997) (the \( c \) state of the cell is not shown). \( W_x, W_y, W_s, W_c, W_h, W_o \) are learned parameters.

In this model, there is a single fixed space from which all actions are taken, which is appropriate for ThingTalk because the library of supported functions is defined ahead of time.

### 5.1 Grammar-Based Encoding of ThingTalk

As in previous neural semantic parsing work (Xiao et al., 2016; Krishnamurthy et al., 2017), the target program is encoded as a sequence of productions in the formal grammar of ThingTalk, augmented with explicit type annotations (Fig. 6.3). This sequence of predictions is generated by a deterministic parser for the target programming language.

We explore both a top-down representation, in which the top-most nodes of the syntax tree are predicted first, and a bottom-up representation, in which the leaves are generated before intermediate nodes that combine them. We observe that the decoder learns a conditional probability for the next production given the history. We hypothesize that the bottom-up representation can express type information better, given that type inference in compilers is a bottom-up algorithm.

#### 5.2 Loss Function

Our model is trained with max-margin loss (inspired by Crammer and Singer (2001)):

\[
\mathcal{L}(y, x) = -\sum_{t=1}^{T} \max_y (\hat{y}_t y - \delta_{\hat{y}_t y} + 1) - \hat{y}_{t,y_t}
\]

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.

We hypothesize that this objective can be more robust to imbalances between training and test distribution, compared to standard cross-entropy loss. The intuition is that, similar to Support Vector Machines, 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.

### 6 Experiments

Training a model to handle primitives is significantly easier than compounds, which together with parameter passing and filters, can have many combinations. It is desirable if learning can be factored; namely, can we teach the model (a) the primitives and (b) the concept of compound constructs, and have it automatically handle new compound combinations that it has never seen before?

To explore this question, we experiment with three scenarios:

- **Known commands**, where the test data is similar to the training data.
- **New combinations**, where the combinations of functions in test set are not in the training set.
- **New services**, where the model is tested on compound sentences involving a new service,
and the model has seen only primitives of that service in training.

Our ultimate metric is the full Program Accuracy, which considers the result to be correct only if the output is an executable program with correct functions, parameter passing, and filters. For comparison with prior work, which ignores parameter and has no filters, we also report on Function Accuracy (Quirk et al., 2015). To understand the coverage of the functions, we also report the average F1-score of function prediction across all Thingpedia functions.

6.1 Experimental Setup

Our model is a 75-dimensional BiLSTM encoder, and a 150-dimensional LSTM decoder. Embedded words are linearly projected to size 50 before passing to the encoder LSTM; output productions are embedded in a learned 50-dimensional embedding. Overall, the model has 405,100 parameters. Recurrent Dropout (Srivastava et al., 2014; Gal and Ghahramani, 2016) with parameter 0.5 is applied to all non-linearities, including to the output state of the LSTM (but not to the memory cell). We also apply gradient clipping, with the same value for all models. Models are trained with RMSProp (Tieleman and Hinton, 2012), with exponentially decaying learning rate.

All models are trained for a fixed number of iterations, and we apply early stopping, using a validation set that consists exclusively of paraphrases. The same validation set was also used for tuning.

Following Dong and Lapata (2016), we preprocess all sentences, replacing all constant numbers, quoted strings, and entities with typed constant tokens (Fig. 6.2). We also apply spellchecking based on the Hunspell library (Németh, 2003). We use CoreNLP (Manning et al., 2014) for tokenization and named entity extraction. We use 300-dimensional GloVe word vectors (Pennington et al., 2014) to embed the sentence, which we augment with one-hot type features for the constant tokens. We do not backpropagate into the word vectors during training time.

6.2 Accuracy on Known Commands

Our first experiment evaluates the accuracy on the test set of paraphrase sentences across 4 models:

- **Retrieval**: a baseline model that chooses the closest training sentence to the input, based on a distance metric defined on the bag-of-word embedding of the sentence.
- **Seq2Seq**: a sequence-to-sequence model similar to Dong and Lapata (2016), with no grammar structure and standard cross-entropy loss.
- **TD-Seq2TT**: our model where the grammar productions are predicted in top-down order and max-margin loss.
- **BU-Seq2TT**: similar to TD-SEQ2TT, but with bottom-up grammar productions.

Fig. 7 shows the experiment results based on a test set of 1,812 test sentences randomly selected from the Paraphrase set. For comparison, the best known result on IFTTT (Beltagy and Quirk, 2016) reports 82% function accuracy, on the smallest, manually cleaned test set of 758 sentences (out of 4,294 in their full test set, and 77,495 in their training set). The best reported program accuracy result on IFTTT is 3.7%.

Because we have a higher-quality training set, even the Retrieval model gets 85.3% accuracy on functions and 55.7% on whole programs. Note that the Retrieval model works well because all but 3 sentences map to programs that are already in the training set. The best overall result is obtained by BU-SEQ2TT with a function accuracy of 97% and a program accuracy 89%, with the improvement over Seq2Seq mainly in program accuracy. TD-SEQ2TT gets similar results, suggesting that the grammar structure is effective for this problem.

Across the board, there is a significant drop in accuracy from function to program accuracy, because the latter requires not just the correct identification of functions, but all the parameters and filters. Filters are harder to understand; the drop is larger for primitives primarily because filters in the primitive sentences in the ThingTalk dataset are harder than those for compound sentences.

6.3 Accuracy on Untrained Combinations

To evaluate the models on new combinations of functions, we split the Paraphrase dataset so none of the function combinations in the test set are seen in the training or dev sets. After the split, the Paraphrase training set contains 21,626 sentences, the dev set 500 and the test set 1,303. We also use a cross validation set of 1,137 sentences for early stopping. Combinations in the cross validation set are also present in the training set, but not in dev or test set. Similarly, we remove from the Synthetic and Augmented sets sentences sharing the same
combinations that are in the test set, reducing them to 541,373 and 14,101 sentences, respectively.

The Retrieval model is not applicable, as it cannot generalize to new combinations and will thus yield a 0% function and program accuracy. To analyze the contribution of the different components of the model, we experimented with the Seq2Seq model, with the addition of just grammar structure, or just max-margin loss, and the TD-S Eq 2TT, BU-S Eq 2TT models over three training sets, Paraphrase (P), with Augmented set (A), and Synthetic set (S) as well, as shown in Fig. 8.

The grammar structure and the use of margin loss improve the accuracy of the Seq2Seq model, independently. The combination outperforms each concept on its own. The best program accuracy of 82% is obtained by BU-S Eq 2TT, trained with the entire dataset. It gains 3% from the addition of the Synthetic set to its training. Overall, BU-S Eq 2TT is 10% better than the basic Seq2Seq model with attention. We also observe that the bottom-up model consistently performs 3 to 5% better than the top-down one.

7 Conclusion

This paper presents the first public dataset of high-quality compound virtual assistant commands. Compared to the IFTTT dataset, the ThingTalk dataset makes it possible to understand not only functions, but also parameters, filters and the other subtleties in user commands.

We show that it is possible to train a model that abstracts the compositional structure of the sentence from the individual primitives of the ThingTalk dataset. To the best of our knowledge, our model and formulation is the first attempt to generate completely new combinations of functions. Our model outperforms a basic sequence-to-sequence model with attention by 12%.

We are the first to propose the use of bottom-up grammar prediction for Seq2Seq neural semantic parsing, and we find that it improves compositionality significantly on our task. We also propose the use of max-margin loss, which provides an improvement over the cross-entropy loss baseline in the task of predicting new combinations.

References


