Natural Language Decomposition and Interpretation of Complex Utterances

Harsh Jhamtani, Hao Fang, Patrick Xia, Eran Levy, Jacob Andreas and Benjamin Van Durme
Microsoft
{hjhamtani,hafang,patrickxia,erlevy,jaandrea,ben.vandurme}@microsoft.com

Abstract
Designing natural language interfaces has historically required collecting supervised data to translate user requests into carefully designed intent representations. This requires enumerating and labeling a long tail of user requests, which is challenging. At the same time, large language models (LLMs) encode knowledge about goals and plans that can help conversational assistants interpret user requests requiring numerous steps to complete. We introduce an approach to handle complex-intent-bearing utterances from a user via a process of hierarchical natural language decomposition and interpretation. Our approach uses a pre-trained language model to decompose a complex utterance into a sequence of simpler natural language steps and interprets each step using the language-to-program model designed for the interface. To test our approach, we collect and release DeCU—a new NL-to-program benchmark to evaluate Decomposition of Complex Utterances. Experiments show that the proposed approach enables the interpretation of complex utterances with almost no complex training data, while outperforming standard few-shot prompting approaches.

1 Introduction
Neural sequence models, pre-trained on large datasets of language and code, are extremely effective at parsing natural commands into programs, database queries, and other structured representations of user intent [Chen et al., 2021; Li et al., 2021; Shin et al., 2021; Roy et al., 2022]. However, developing an interface that enables a user to interact with a new API or software system still requires substantial system-specific data collection. Users, meanwhile, may not be aware of the scope of this data collection, and pursue an open-ended set of goals more complicated than those anticipated by system designers.

In this paper, we present DeCINT1, an approach to decompose complex utterances into a sequence of simpler NL steps, each resembling a simpler elementary utterance that an existing language-to-program interpreter for the NL interface can parse to a sub-program. Consider the utterance “Exchange the timing of my meetings with Jane and Smith” (Figure 1). DeCINT breaks the utterance down into four NL steps, using a pre-trained LLM and just a few annotated decompositions. The generated NL steps are parsed into programs, relying primarily on a relevant (to the step being parsed) subset of a larger set of existing elementary utterances associated with simpler programs in the target representation. DeCINT thus enables an NL interface system to handle user requests representing complex goals (never seen by a semantic parser) by breaking them into a series of NL steps that are interpreted into APIs (never seen by an LLM). Our work is related to recent work which demonstrates that large language models (LLMs) encode knowledge that can be used to interpret complex user goals requiring numerous steps to complete, in setups such as question answering [Wolfson et al., 2020; Khot et al., 2022] and embodied agents [Ahn et al., 2022; Huang et al., 2022]. Compared to such past work, we are concerned with generating programs in a carefully designed intent representation. Starting with labeled elementary utterances, we wish to be able to parse complex utterances that are broader in scope compared to the abundant elementary utterances.

To study utterance decomposition in the NL-to-program space, we collect and release DeCU—a new benchmark dataset to evaluate models for Decomposition of Complex Utterance. DeCU consists of (1) a set of elementary utterances and corresponding programs for managing calendar events and emails and (2) a diverse set of complex user utterances annotated with decompositions into sequences of elementary utterances and their corresponding program fragments. Experiments on DeCU show that DeCINT outperforms direct few-shot prompting approaches, making it possible to build NL interfaces that accomplish complex goals without large amounts of complex labeled data.

2 Task Overview
We study the problem of parsing a user utterance $x$ into a program $y$ that correctly reflects user intent (Figure 1). We focus on a version of the problem with the following characteristics:

- A domain developer has already collected a dataset of elementary utterances annotated with corresponding programs. These utterances represent narrow user goals associated with simple and short programs.
The utterances in DeCU focus on calendar events and emails. The dataset contains both elementary utterances and complex utterances. Elementary utterances (§3.2) are paired with declarative Scala3 programs based on a domain library (§3.1) that admits a fixed set of APIs and specified types. Complex utterances (§3.3) are annotated with a corresponding sequence of elementary utterances, each paired with a program. Only a few of these complex utterances are included in the training set; they are mainly used to form a test set.

Figure 1 illustrates an example: “Exchange the timing of my meetings with Jane and Smith”. How such an utterance should be decomposed is domain-dependent: here, the calendar API does not provide a single endpoint that can swap pairs of meetings; instead, the system must search for the two meetings individually, then update each of their times. Figure 1 shows a possible decomposition into four steps. The first generated NL step, “Find the meeting with Jane”, is translated to a program fragment: val s1 = theEvent(with_(“Jane”)). Individual steps typically represent easier-to-solve inputs for the NL-to-program parser that primarily relies on the annotated elementary utterances.

In addition to domain-specific knowledge of APIs, decomposition of complex utterances often relies on domain-general reasoning and common sense knowledge – for example, to avoid double-counting meetings that match two search results (Figure 2, utterance 1), or to recognize that meetings cannot conflict with themselves (utterance 2).
To study how complex utterances are represented on heuristics, treating API names, argument names, and values as individual tokens.

These utterances are counted. Appendix B provides more details.

DeCu contains 841 elementary utterances paired with programs. A few examples are shown in the top box in Figure 1. These utterances are elementary in that they represent narrow user goals such as creating or deleting a single meeting, which can typically be achieved using a single API. As such, they have relatively short programs, generally less than 5 tokens. Examples are written and reviewed by domain experts who are familiar with the domain library (on account of their experience from working with a deployed system leveraging such a library) and annotation guidelines.

3.3 Complex Utterances

To study how complex utterances can be supported by an NL interface, we collect a diverse set of more involved user requests, and annotate these with decompositions into elementary steps, along with programs for each step. As the name suggests, compared to elementary utterances, these utterances represent more complex and broader user goals, with the corresponding programs typically being much longer (an average of 14.5 tokens per program). To collect complex utterances, we employ a mix of manual authoring and automated utterance generation. Manual authoring is performed by domain experts with a focus on diversity and goals that require the composition of multiple calls to the domain APIs. For automated collection techniques, we generate utterances using GPT-3 [Brown et al., 2020] prompted with a few random examples of manually-authored utterances. About 60% of all the collected utterances were generated automatically. Appendix A provides more details on utterance collection. Examples are shown in Figure 1.

Decomposition Annotations: Six annotators familiar with the domain (annotators had past experience working with the domain library) decompose complex utterances into elementary ones. When results from earlier steps must be reused, these NL decompositions may include explicit reference to earlier step outputs (Figure 2). More information about annotator instruction is provided in Appendix A. Each annotation was additionally reviewed by two additional domain experts, separate from the set of 6 annotators.

Data Statistics: We collected a total of 210 unique complex utterances. The dataset is a mix of 126 utterances paired with annotated programs and 84 that are unannotated. As discussed later, in addition to reference-based metrics, we also provide various reference-less metrics that do not require annotations. While it is a relatively small count, note that most of the data (200 out of 210) is used to construct an evaluation set, as we are interested in learning to generalize from very small numbers of training examples. Additionally, we would like to note that our dataset is of similar scale as some other recent datasets: SayCan [Ahn et al., 2022] was evaluated only on 101 examples, and each of the Big-Bench [Suzgun et al., 2022] hard task used less than 250 examples for evaluation. Annotated complex utterances in our full dataset exhibit a

\[\text{Utterance 1:}\]

\text{Change my meetings with Abby and those with Dan this week to start 5 minutes later.}

\text{Decomposition:}

\text{Step 1: Find events with Abby this week}
\text{val s1 = findEvents(with_("Abby"))}
\text{Step 2: Find events with Dan and without Abby this week}
\text{val s2 = findEvents(with_("Dan"))}
\text{Step 3: Set all meetings from the list of events s1 to start 5 minutes later}
\text{val s3 = s1.map((x: Event) \rightarrow modifyEvent(x, \text{startsAt}(x.\text{start.local.time} + 5.\text{minutes})))}
\text{Step 4: Set all meetings from the list of events s2 to start 5 minutes later}
\text{val s4 = s2.map((x: Event) \rightarrow modifyEvent(x, \text{startsAt}(x.\text{start.local.time} + 5.\text{minutes})))}

\[\text{Utterance 2:}\]

\text{Decline any meeting invitations that are scheduled during my weekly team meeting.}

\text{Decomposition:}

\text{Step 1: Find the event called "team meeting" that recurs weekly.}
\text{val s1 = theEvent(called("team meeting") \&\& \text{recurringWeekly})}
\text{Step 2: Find all events.}
\text{val s2 = findEvents()}
\text{Step 3: Filter events from list s2 to only include ones that intersect with event s1 that are not s1.}
\text{val s3 = s2.filter((x: Event) \rightarrow x.\text{interval.intersects}(s1.\text{interval}) \&\& x.id \neq s1.id)}
\text{Step 4: Decline events in the list s3.}
\text{val s4 = s3.map((x: Event) \rightarrow respond(x, \text{ResponseStatusType.declined}))}

Figure 2: Examples of complex utterances in DeCU. Each utterance is accompanied by decompositions consisting of a sequence of NL steps and associated program fragments, annotated by domain experts.

3.1 Domain Library

The domain library defines the set of types and functions available for program annotations. Types model objects such as Person and Event, whereas functions represent actions that can be taken by the agent, including high-level APIs (e.g., createEvent, findEmails), low-level operations (e.g., min, +), predicate constructors (e.g., called, startsAt), etc. The domain library for DeCU is packaged as standard Scala source code, consisting of 33 types and over 200 functions.\(^2\)

3.2 Elementary Utterances

DeCu contains 841 elementary utterances paired with programs. A few examples are shown in the top box in Figure 1. These utterances are elementary in that they represent narrow user goals such as creating or deleting a single meeting, which can typically be achieved using a single API. As such, they have relatively short programs, generally less than 5 tokens. Examples are written and reviewed by domain experts who are familiar with the domain library (on account of their experience from working with a deployed system leveraging such a library) and annotation guidelines.

\(^2\)Some built-in types (e.g., String, Boolean), functions (e.g., map), and control flow statements (e.g., if) are not explicitly defined and counted. Appendix B provides more details.

\(^3\)To compute this statistic, programs are split into tokens based on heuristics, treating API names, argument names, and values as individual tokens.

\(^4\)Appendix available at https://github.com/microsoft/decomposition-of-complex-utterances
**Complex Utterance:** Exchange the timing of my meetings with Jane and Smith

**Step by Step Decomposition**

**Step 1:** Find the meeting with Jane

```
val s1 = theEvent(with_("Jane"))
```

**Step 2:** Find the meeting with Smith

```
val s2 = theEvent(with_("Smith"))
```

**Step 3:** Update event s1 to start and end time of event s2

```
val s3 = modifyEvent(s1, startsAt(s2.start) and endsAt(s2.end))
```

**Step 4:** Update event s2 to start and end time of event s1

```
val s4 = modifyEvent(s2, startsAt(s1.start) and endsAt(s1.end))
```

**A. K (<=10) number of Complex Utterance Decomposition examples.**

**Complex Utterance:** Check if John has accepted our meeting tomorrow and if not then add John’s manager to the call

**Step 1:** Find my meeting with John tomorrow

```
val s1 = theEvent(with_("John") and queryAt(tomorrow))
```

**Step 2:** If John has not accepted the event s1 then update the event s1 to add his manager

```
val s2 = Option.when(!s1.attendees.isAttending(thePerson("John"))) {
    modifyEvent(s1, with_((thePerson("John").manager)) }
```

**B. M (<=25) number of Elementary Utterances similar to “Change event s2 to start and end time of event s1”, chosen from a larger set.**

**Utterance:** Change the title of this morning’s meeting to “Q&A”

**Program:**

```
val s1 = modifyEvent(theEvent(
    queryAt(morning on this[Date]), called("Q&A"))
```

**Utterance:** If list of events s2 is empty then update event s1 to end at 2:30 pm.

**Program:**

```
val s3 = Option.when(s2.isEmpty) {
    modifyEvent(s1, endsAt(2 :: 30).pm))
```

---

Figure 3: DECINT maps complex utterances into elementary steps, each of which is parsed in sequence to arrive at a final program. NL decomposition and program generation steps are interleaved. While parsing a step, up to M similar examples of elementary utterances are retrieved.

---

Diverse range of properties (an utterance can have multiple): 55% use a map operation (for-loop), 36% contain actions based on a condition, 31% use a filter operation, 24% query about calendar/email, 37% contain a create meeting action, 9% contain a delete meeting action, and 31% contain a modify meeting action. The average number of decomposition steps in our data is 3, with a maximum of 7 steps. The average number of tokens in each program is 14.5, while the average number of tokens in the program fragment corresponding to a single step is 4.8. For comparison, the average number of tokens in the programs for elementary utterances is 4.5.

**4 Approach**

The DECINT approach, illustrated in Figure 3, maps a complex utterance x to a sequence of interpretable lower-level NL steps (z₁, z₂, ... ) that resemble elementary utterances. Each step or low-level utterance z_j is parsed into a program fragment y_j. In particular, DECINT maps from commands to programs according to the following iterative generative process:

1. **Natural Language Decomposition:**

   \[ z_j \sim p_\lambda(x, z_{<j}, y_{<j}) \]

2. **Program Generation:**

   \[ y_j \sim p_\lambda(x, z_{\leq j}, y_{<j}) \]

NL Decomposition (§4.1) and program generation (§4.2) steps are interleaved, with later portions of the language decomposition conditioned on earlier program fragments. In principle, one could also condition on the return values of the earlier program fragments (see Limitations section). We do not do so in this paper, as running the programs would require API implementations and input data.

**4.1 Natural Language Decomposition**

The NL decomposition stage generates the next NL step \( z_j \) conditioned on the user utterance \( x \) and any previously generated steps and program fragments. We obtain \( z_j \) by greedy decoding from a pre-trained LLM in a few-shot in-context learning setup [Brown et al., 2020]. The model is prompted with \( K = 10 \) example decompositions, each of which consists of an utterance \( x \) followed by any previous steps \( y_{<j} \) and their program fragments, all concatenated together \( (x, z_1, y_1, z_2, ..., z_N, y_N) \). We additionally found it useful to include a list of up to \( M \) elementary NL utterances at the start of the prompt (before the \( K \) decomposition examples), selecting the ones with highest BM25 similarity to the input utterance. This is intended to inform the model about the kind of elementary steps the NL-to-program parser can handle. (An example constructed prompt is shown in Appendix C.) Example decompositions are taken from the set of 10 complex utterances in the training split of DeCU.

DECINT’s ability to perform NL decomposition thus results from a combination of parametric knowledge about the structure of programs in general (the result of pretraining) and non-parametric knowledge about the domain of interest (obtained via in-context learning). Together, these enable generalization to structurally novel user requests. For example, there are no training examples that involve exchanging the timing of two meetings (the test example in Figure 3), but DECINT nonetheless synthesizes a correct program.
4.2 Program Generation

The program generation step synthesizes a program fragment $y_j$, for a given NL step $z_j$, conditioned on any preceding steps and incomplete program. This is a well-studied semantic parsing problem, and we design the NL-to-program parser largely following past work that applies pre-trained LLMs. We use in-context learning with dynamically selected prompt examples from the set of elementary examples data [Brown et al., 2020]. As before, we use greedy decoding. Generated program fragments may refer to previously generated fragments using named step variables. For a given NL utterance or step, we identify up to $M$ examples from the set of elementary utterances, where each example is an (utterance, program) pair (as shown in box B in Figure 3). The selection of the examples is based on the similarity of the utterance to the intermediate NL step being parsed. To compute similarity, we again use BM25, as in past work [Rubin et al., 2022; Roy et al., 2022]. In pilot experiments on training data, we discovered it was useful to also include the $K$ decomposition examples at the bottom of the prompt (detailed prompt example shown in Appendix C). This may be because the decomposition examples provide a demonstration of how to generate program fragments for a step conditioned on previous steps and help bridge any possible domain shift from elementary to complex utterances.

4.3 Baselines

The DECINT method decomposes a complex utterance into NL steps, separately parsing each step, and using internal variable references to assemble a larger program. The standard few-shot prompting approach for tasks like this one instead directly predicts the parse without generating the intermediate NL steps [Roy et al., 2022]. We compare to this approach, which we denote DIRECT-PRED, in our experiments. There are a few key differences compared to the DECINT method. Complex utterance examples are presented without the intermediate NL steps (i.e., each utterance is paired with a multi-line program). The output generation is a single-step process since there are no intermediate NL steps that need to be generated. As with DECINT, examples of elementary utterances are also included in the prompt. We also consider a COPT [Wei et al., 2022] baseline, wherein the model first predicts all intermediate NL steps and then predicts the program. Accordingly, the complex utterance examples in the prompt are annotated with intermediate steps. This baseline resembles the method proposed in Jiang et al [2023]. Note that compared to COPT, DECINT interleaves step generation and parsing, and dynamically updates the subset of exemplars from elementary utterances to be relevant to the step being parsed.

We also report results using a variant of DECINT that relies only on $K$ decomposition exemplars but without access to elementary utterances ($M=0$ instead of 25). We refer to such a baseline as FEW-SHOT. We also consider a variant of DECINT that uses only a single decomposition exemplar ($K=1$ instead of 10), and thus relies almost entirely on the elementary utterances from the underlying domain. We refer to the variant as ELEMENTARY-ONLY. Finally, we also report results on a variant of DECINT that employs a fixed set (randomly selected) of elementary utterance exemplars. We refer to this variant as REACT, as similar to ReAct [Yao et al., 2023], it does not employ any dynamic exemplar selection. This is in contrast to DECINT that dynamically identifies elementary utterances most similar to the generated step being parsed.

5 Experiments

5.1 Evaluation

Overlap with Reference Programs: We report Exact Match (EM) and character-based edit distance (CER) metrics\(^6\) against the gold program. Before computing these metrics, we normalize the programs by loweringcase the entire program and removing extra spaces. Since there can be multiple possible ways to express the target multi-line program, Exact Match can only be viewed as a lower-bound metric for this task. These metrics are reported only for the subset of the data that consists of annotated reference programs.

Well-formed Evaluation: Additionally, we report the fraction of predictions that are valid (WellForm) under the domain library, i.e., the full program follows correct syntax and only uses functions available in the library. Note that WellForm does not necessarily represent correctness with respect to the user goal. We report the metric for the entire test set.

Program Correctness: Finally, we report the overall correctness of the generated programs. We define a program to be correct overall if: it is well-formed, and correctly represents the user request. We use GPT-4 (gpt-4-32k) [OpenAI, 2023] to rate the correctness of the generated programs (Correct). The prompt consists of an instruction and four manually labeled exemplars (two “correct” and two “incorrect”) followed by the test example. Each example is a user utterance followed by the associated program. The label is a natural language caption/explanation of the generated program, followed by a final verdict on whether the generated program is “correct” or “incorrect” for the given user utterance – following a chain-of-thought style prediction\(^7\). Since we have an automatic static analysis to infer exactly which programs are well-formed (WellForm), outputs that are not well-formed are automatically considered to be incorrect as per the definition above (but are included in the denominator for all evaluations). Note that the Correct metric is reference-less, is easier to scale than human evaluations, and correlates well with human ratings (Section 5.3).

5.2 Setup

We consider the task of parsing complex utterances in DeCU given only ten complex utterances (annotated with decompositions) to be used as training data (exemplars for in-context learning). We report results on the test set consisting of the remaining 200 complex utterances. We use a maximum of $M \leq 25$ additional elementary utterances (as many as permitted by the LM’s context window) selected according to BM25 similarity with the step being parsed. We use OpenAI’s text-davinci-003 model as the LLM for generating each NL step as well as for parsing it into a program.

\(^6\)https://huggingface.co/spaces/evaluate-metric/cer

\(^7\)The exact prompts used in Correct are presented in Appendix D
5.3 Evaluation of Generated Program

Table 1 reports various automated metrics. DECI NT outperforms all the baselines, sometimes by a wide margin. As can be seen in the table, DECI NT outputs receive an overall correctness score (Correct) of 41% for complex utterances compared to 34% and 25% for the baselines DIRECT-PRED and CoT respectively.\(^8\) We posit that DECI NT is able to make more effective use of pretraining by breaking down a complex command into NL steps and retrieving relevant exemplars for each step. Further, FEW-SHOT, that is equivalent to DECI NT with M=0, fares badly, suggesting that DECI NT relies on information from elementary utterances in addition to supervised decompositions. Finally, ELEMENTARY-ONLY, which is equivalent to DECI NT with K=1, also does worse than DECI NT, suggesting the usefulness of a handful of supervised decompositions. Note, however, that a 54% of the predictions from DECI NT are not well-formed, indicating that even structural generalization in DECU remains a major challenge. Nonetheless, DECI NT fares better compared to other methods on WellForm metric.

**Human Evaluation for Program Correctness:** We also obtained the overall program correctness rating (“correct” vs “incorrect” for a user utterance) from human evaluators familiar with the domain library. Just as was the case with Correct metric, outputs that are not well-formed are automatically considered incorrect. The aggregate scores for DECI NT, DIRECT-PRED and CoT (our method and the two top performing baselines as per automated Correct metric) under human evaluation are 41%, 33% and 26% respectively, which are very close to the scores for these methods under the automated Correct metric. Additionally, we observe a high correlation between human annotator-provided judgment and Correct judgments (a more detailed correlation analysis is provided in the Appendix D).

**Results with other LLMs:** We also report results using GPT-4 (gpt-4-32k) and LLAMA-2-70B [Touvron et al., 2023] as the underlying LLM. Due to cost considerations, we report results only for the top three methods from Table 1. We observe that DECI NT outperforms the baselines, demonstrating our approach is effective across underlying LLMs (Table 2).

5.4 Evaluation of NL Decomposition

We measure whether the NL decomposition steps altogether are sufficient and correct to complete the user request.\(^9\) For

<table>
<thead>
<tr>
<th>System</th>
<th>Correct</th>
<th>WellForm</th>
<th>EM</th>
<th>CER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT-PRED</td>
<td>0.34</td>
<td>0.36</td>
<td>0.04</td>
<td>0.44</td>
</tr>
<tr>
<td>CoT</td>
<td>0.25</td>
<td>0.20</td>
<td>0.05</td>
<td>0.46</td>
</tr>
<tr>
<td>FEW-SHOT</td>
<td>0.13</td>
<td>0.19</td>
<td>0.00</td>
<td>0.50</td>
</tr>
<tr>
<td>REACT</td>
<td>0.21</td>
<td>0.23</td>
<td>0.02</td>
<td>0.49</td>
</tr>
<tr>
<td>ELEMENTARY-ONLY</td>
<td>0.23</td>
<td>0.31</td>
<td>0.04</td>
<td>0.54</td>
</tr>
<tr>
<td>DECI NT</td>
<td>0.41</td>
<td>0.46</td>
<td>0.05</td>
<td>0.40</td>
</tr>
</tbody>
</table>

\(^8\) Differences are significant (p < 5%) using bootstrap resampling.

\(^9\) Unless stated otherwise, all analysis uses outputs with text-davinci-003 as the underlying LLM.
example, the output from DECINT for the second utterance in Figure 4 is not sufficient and correct because the fourth step fails to specify the duration of the meeting, which is supposed to be 15 minutes as per user request. A random subset of 40 of DECINT NL predictions and corresponding expert annotations were manually labeled by one of the authors as correct or incorrect. The expert annotations and DECINT predictions were rated as 98% and 85% correct, respectively. Future work can explore ways to further improve the accuracy of the predicted NL steps. We also conducted a step-level evaluation, which we discuss in Appendix D.

5.5 Qualitative Analysis

We provide example predictions in Figure 4, with additional examples provided in the Appendix. Additionally, we perform an error analysis of the NL-to-program step of DECINT. We restrict the study to the predictions that were labeled as incorrect in Table 1. The most common issues are those that make the program not well-formed, as summarized in Table 1. Many errors are due to nonexistent APIs / API arguments (21% of the incorrect programs have at least this problem) and nonexistent type attribute (43%). A smaller number result from even more basic syntax errors and type mismatches (17%). Future work could constrain the outputs of the parser [Shin et al., 2021] to only use allowed functions and follow correct syntax, though such approaches can substantially increase the cost of decoding.

A few errors result from predictions that capture only partial user intent (6%). For example, for utterance 2 in Figure 4, the prediction does not capture the user intent of creating the second event for 15 minutes. Many of the remaining errors involve more fundamental semantic mismatches between user intents and model outputs. For example, for “Loop around all my 1/1 meetings this week so that they also happen next week”, the prediction updates the meetings this week instead of creating another set of meetings next week.

6 Related Work

Past work has explored using command decomposition to break down complex tasks or requests into smaller subtasks that are easier to manage. The LaMDA model [Thoppilan et al., 2022], for example, is capable of breaking down “How to” type queries into steps. However, generated steps are not tied to any actions or APIs, and are more in the form of a narrative rather than executable steps.

Khot et al. [2021] decompose a question into sub-questions that can be answered by a neural factoid single-span QA model and a symbolic calculator. Drozdov et al. [2022] decompose an utterance using a syntactic parse. However, not all utterances in our dataset would lend to such a style of decomposing, since all required actions might not align to a part of the parse. Recent work [Jiang et al., 2023] has also explored first generating an entire plan in NL and then generating a program. Paranjape et al. [2023] focus on using tools and python scripts to complete a given task such ‘Translate into Pig Latin’. Compared to such past work, the complex utterances in our case are decomposed into intermediate steps that are parsed into a sub-program in the target representation as opposed to generating Python programs. Additionally, these sub-programs are a part of the final program output and thus we care about the accuracy of intermediate steps as well.

A related area of research involves grounding high-level tasks, expressed in natural language, to a chosen set of actionable steps that a robot could take [Sharma et al., 2022; Singh et al., 2022; Ahn et al., 2022; Huang et al., 2022]. Huang et al [2022] propose a method to ground high-level tasks such as ‘make breakfast’ to a set of actionable steps such as ‘open fridge’. Such work typically assumes a fixed inventory of low-level actions. For example, ‘Semantic Translation’ discussed in Huang et al [2022] translates the predicted step into an admissible action by calculating the semantic distance of the predicted action phrase against all possible actions. The APIs in our case can be composed and chained together, and have optional arguments. So identifying an exhaustive set of allowed actions under the DSL (domain-specific-language) in question is intractable.

7 Conclusion

We have presented DECINT, an approach for interpreting complex user utterances by decomposing them into elementary natural language steps. To evaluate methods for generating programs from user requests, we have introduced the DeCU dataset, featuring a diverse set of utterances requiring substantial generalization from a small training set. Experiments on DeCU show that DECINT outperforms a standard few-shot prompting approach to program generation, with additional analysis revealing opportunities for improvement in both natural language decomposition and program generation phases.

Ethical Statement

We leverage pre-trained neural language models such as GPT-3, and systems built using our approach might inherit some biases present in these pre-trained models. We build a system for NL-to-program, that users can leverage to command various NL interfaces. Such systems are not perfectly accurate and should be carefully deployed since they may lead to unintended side effects.

Acknowledgements

We would like to thank Jason Eisner, Subhro Roy, Jason Wolfe, Matt Gardner, and Val Ramirez for valuable discussions. We thank anonymous reviewers for feedback on the paper.

References

[Ahn et al., 2022] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman,


