Online Semantic Parsing for Latency Reduction in Task-Oriented Dialogue

Jiawei Zhou[♠][*] **Jason Eisner**[◇] **Michael Newman**[◇]
Emmanouil Antonios Platanios[◇] **Sam Thomson**[◇]
[♠]Harvard University [◇]Microsoft Semantic Machines
[♠]`jzhou02@g.harvard.edu`
[◇]`{jason.eisner,mike.newman,`
`anthony.platanios,samuel.thomson}@microsoft.com`

Abstract

Standard conversational semantic parsing maps a complete user utterance into an executable program, after which the program is executed to respond to the user. This could be slow when the program contains expensive function calls. We investigate the opportunity to reduce latency by predicting and executing function calls while the user is still speaking. We introduce the task of *online semantic parsing* for this purpose, with a formal latency reduction metric inspired by simultaneous machine translation. We propose a general framework with first a learned prefix-to-program prediction module, and then a simple yet effective thresholding heuristic for subprogram selection for early execution. Experiments on the SMCalFlow and TreeDST datasets show our approach achieves large latency reduction with good parsing quality, with a 30%–63% latency reduction depending on function execution time and allowed cost.

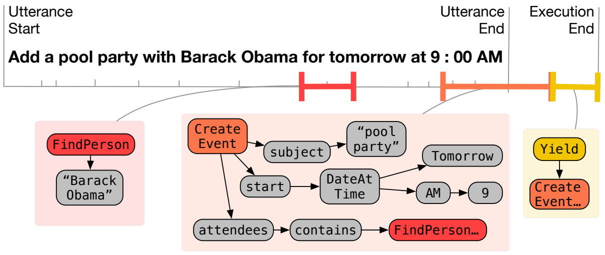


Figure 1: A possible execution timeline for *"Add a pool party with Barack Obama for tomorrow at 9:00 AM"*. `FindPerson` was predicted and executed after the token *"Obama."* `CreateEvent` was predicted and executed after the token *"9"* (`AM` was guessed). `Yield` was also predicted after the token *"9"*, but could not begin until `CreateEvent` finished. Because `FindPerson` and `CreateEvent` started execution before the end of the utterance, the top-level node `Yield` was able to finish early, and overall latency was reduced. Gray nodes have near-instantaneous execution, and do not contribute to the timeline. Edge labels were omitted to save space.

1 Introduction

In task-oriented dialogue systems, a software agent typically translates a user's intent into a program, executes it to query information sources (e.g., find a person in the user's contact list) or effect external actions (e.g., schedule a meeting or send an email), then communicates the results back to the user. If an agent waits to begin this process until the user finishes speaking, there is a noticeable lag before the user receives a response. The complex intents in the datasets SMCalFlow (Semantic Machines et al., 2020) and TreeDST (Cheng et al., 2020), for example, can be slow to execute, nesting up to 7 slow function calls that cannot be parallelized. Inspired by simultaneous machine translation, we ask: How much can latency be reduced by interpreting and executing early, before the user finishes speaking?

[*] Work performed during a research internship at Microsoft Semantic Machines.

In general, an agent could begin speculatively executing any subprogram at any instant while a user is speaking, based on partial results from automatic speech recognition (ASR) and the current state of execution. Take Figure 1 for a hypothetical example. If partial programs can be identified while the user is still speaking, they can be pre-executed and the final response to the user could be expedited.

This is an online decision problem: decisions to invoke particular functions on particular arguments can be made before all information has arrived. Thus, we refer to it as **online semantic parsing**. This requires spotting user intents that have already been expressed (without the help of aligned training data) and—even harder—anticipating user intents that have not been expressed yet. To assess an online semantic parser, we propose reporting the reduction in latency of the agent's final response (relative to an offline agent), as measured by real or simulated execution of the function calls.

We propose two approaches. Our first system is built on a neural graph-based semantic parser, which is specially trained to parse an incomplete utterance into a full program. Our second system is a pipeline that uses a language model (LM) to predict how a user will finish the incomplete utterance, and then parses the predicted completed utterance. In either case, a subprogram is selected for execution as soon as the semantic parser predicts that it has a high probability of being in the correct parse. Experiments on both SMCalFlow and TreeDST datasets show that both approaches achieve high latency reduction with a small number of excess function calls. We make three main contributions: First, we propose a new task for online semantic parsing and a realistic evaluation metric for latency reduction. Second, we present a neural graph-based semantic parser that matches or surpasses the state-of-the-art on SMCalFlow and TreeDST, and extend it to support two novel approaches to map utterance prefixes to programs. Third, we show our approaches achieve estimated latency reductions of 30%–63%, setting up a good benchmark for future explorations.

2 Background

Simultaneous Translation Our task is inspired by the online version of machine translation (MT), known as **simultaneous MT**, which aims to translate a source sentence in real time into a target language (Wahlster, 1993). The latency of such a system is assessed by counting how many source tokens it has observed before it produces the first, second, third, etc. target token. These counts are aggregated into an overall latency metric—a measure either of "waiting," such as Average Proportion (AP) (Cho and Esipova, 2016) and Consecutive Wait (CW) (Gu et al., 2017), or of "lagging" (in comparison with an ideally paced system), such as Average Lagging (AL) (Ma et al., 2019) and Differentiable Average Lagging (DAL) (Cherry and Foster, 2019; Arivazhagan et al., 2019). We discuss the relationship of our proposed metric to DAL and other existing metrics in Section 4.3.

Approaches to simultaneous MT include explicit source word prediction (Grissom II et al., 2014), discrete decision sequence modeling with reinforcement learning (Satija and Pineau, 2016; Gu et al., 2017), latency-controllable wait-k systems with fixed scheduling (Ma et al., 2019), learned adaptive scheduling (Arivazhagan et al., 2019), and retranslation (Arivazhagan et al., 2020a,b).

Executable Programs as Semantic Graphs Semantic parsing maps natural language to structured meaning representations (MRs) that can be executed or reasoned about. These include general-purpose MRs (Clark and Curran, 2007; Banarescu et al., 2013), database queries (Tang and Mooney, 2001; Zettlemoyer and Collins, 2005; Yu et al., 2018), and source code in general-purpose programming languages (Yin and Neubig, 2017), etc. Despite formal differences, these representations can generally be represented as graphs. We will focus on the dataflow graph (Semantic Machines et al., 2020), which represents an executable program in response to a user's utterance in a task-oriented dialogue system (Zettlemoyer and Collins, 2009). Each function invocation is represented by a node, whose label specifies the function, and whose outgoing[1] edges indicate its arguments, which may be constants or other function invocations.

Preliminaries Formally, we represent a program as a labeled directed acyclic graph $G = (V, E)$, where each node $v \in V$ represents a function invocation or a constant value, and each directed edge $u \xrightarrow{\ell} v \in E$ represents that v fills the ℓ argument of the function u. Positional arguments are given edge labels `arg0`, `arg1`, etc. We use "graph" and "program" interchangeably hereon.

In task-oriented dialogue systems, an executable program G is generated in response to a user utterance $\mathbf{u}$ with possible context $\mathbf{c}$ from the dialogue history. The utterance is a token sequence $\mathbf{u} = (u_1, u_2, \ldots, u_{|\mathbf{u}|})$ and the context is also encoded as a sequence $\mathbf{c} = (c_1, c_2, \ldots, c_{|\mathbf{c}|})$.

We use $\mathbf{u}_{[m]}$ to denote the m^{th} prefix presented to the online system, and t_m to denote the time at which it is presented. t denotes the time at which the complete utterance $\mathbf{u}$ is presented. In our experiments, each $\mathbf{u}_{[m]}$ is some prefix of the gold utterance $\mathbf{u}$. A real system could use the noisy partial outputs returned by an ASR system from successively longer speech prefixes. Each partial ASR output $\mathbf{u}_{[m]}$ is returned at some time $t_m \in \mathbb{R}$. It may append one or more words to the previous output $\mathbf{u}_{[m-1]}$, and may also revise some words.

An *offline* system models $p(G \mid \mathbf{c}, \mathbf{u})$, predicting the program only after the user utterance $\mathbf{u}$ has been fully received. But our online system aims

[1]Our description has reversed the edge directions from Semantic Machines et al. (2020).

to *simultaneously* parse $\mathbf{u}$ as the user utters it, so as to pre-execute subprograms to reduce the final response time. Our setting differs from simultaneous MT in an important way: we currently do not show the user any output until their utterance is complete. So speculatively executing a predicted subprogram, silently, does not commit to using it in the final result. Our parse of $\mathbf{u}_{[m-1]}$ therefore does not constrain our parse of $\mathbf{u}_{[m]}$.[2] Indeed, in this work, we re-parse each prefix from scratch.

We distinguish between the time or times at which a function invocation is *selected* by the system for execution, the time it is actually *executed*, and the time it *returns*. A selected function invocation is not actually executed until its arguments have returned from execution. But by that point, the system may have deselected it (and so will not execute it), since the system's predictions may have changed based on additional input.

3 Methods

After each successive utterance prefix $\mathbf{u}_{[m]}$, we perform the following two steps (see Figure 2):

1. **propose**: Predict the complete graph G from only the current prefix $\mathbf{u}_{[m]}$ and context $\mathbf{c}$.
2. **select**: Select the graph nodes (function invocations) that are worth executing at this time. This is an update that replaces the former list of selected nodes; so any formerly selected nodes that were still waiting for their arguments have lost their chance to execute until they are selected again.[3]

In the first step, we currently search for the single most probable G. More generally, one could construct an estimate of the distribution $p(G \mid \mathbf{c}, \mathbf{u}_{[m]})$. In the second step, we select nodes that are probably correct, using a heuristic approximation to their marginal probability. In future work, selecting a node should also consider the predicted execution cost and the predicted effect on overall latency.

An alternative design would collapse propose and select into a single step that directly predicts some graph fragments to execute. But as gold fragments are not defined, this would require a more complicated training objective. Predicting complete programs may also yield more accurate fragments, by making latent structure explicit.

[2]The corresponding setup in simultaneous MT is re-translation (Arivazhagan et al., 2020b; Han et al., 2021).

[3]Ongoing executions of formerly selected nodes will be allowed to finish, however.

We first describe our general approach for graph prediction (Section 3.1), followed by two different approaches for propose (Section 3.2–3.3), and finally our heuristic for select (Section 3.4).

3.1 Graph Generation Model

We encode any graph G as a sequence $\mathbf{a} = (v_1, \mathbf{e}_1, v_2, \mathbf{e}_2, \ldots, v_{|V|}, \mathbf{e}_{|E|})$. Each element of $\mathbf{a}$ can be regarded as an *action* that adds a node or a sequence of edges to the graph. Note that the subgraphs selected in Section 3.4 below will not necessarily correspond to contiguous substrings of $\mathbf{a}$.

This representation is borrowed from the action-pointer mechanism in Zhou et al. (2021a), but they are operating with graph-utterance alignments in a transition-based model, whereas we develop a more general alignment-free model. Each v_k is a node, representing a constant value or function invocation, while each $\mathbf{e}_k$ is a subsequence that lists all edges between v_k and earlier nodes. At training time, graphs are converted to action sequences to enable us to train a sequence-to-sequence model. At inference time, the model outputs the action sequence, from which the graph can be constructed.

In our action sequence representation, each node and each edge in G corresponds to one token in $\mathbf{a}$, with the only exception being string literals, which can span multiple action tokens. A token of a string literal can appear directly as an action or can be copied from the j^{th} token of the source via a special $\textsc{CopyInput}(j)$ action. The details of the formulation of the action sequence and the model parametrization can be found in Appendix A.

For an offline parser, the model learns $p(G \mid \mathbf{c}, \mathbf{u}) = \prod_{n=1}^{|\mathbf{a}|} p(a_n \mid \mathbf{c}, \mathbf{u}, \mathbf{a}_{1:n-1})$, where the input to the encoder is the concatenation of the context and the full utterance. We call this standard setup FULLTOGRAPH.

3.2 Approach 1: PREFIXTOGRAPH

The FULLTOGRAPH model achieves very strong performance when trained and tested on the standard offline benchmark (see Table 1). We could simply run this trained model on utterance prefixes for our propose step, but that would suffer from a train-test mismatch. Thus, we replace it with a PREFIXTOGRAPH model $p(G \mid \mathbf{c}, \mathbf{u}_{[m]})$ that we explicitly train to map from each prefix of $\mathbf{u}$ to the complete graph. Every $((\mathbf{c}, \mathbf{u}), G)$ pair in the original training data is multiplied into many training pairs $((\mathbf{c}, \mathbf{u}_{[m]}), G)$. Notice that we always use

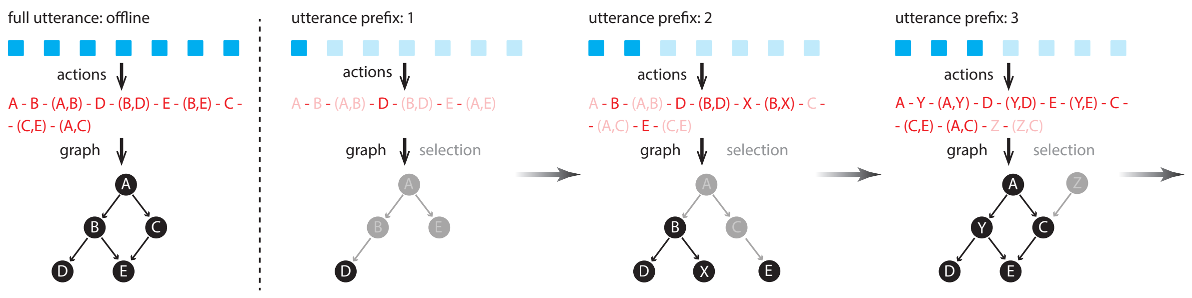

full utterance: offline
actions
A - B - (A,B) - D - (B,D) - E - (B,E) - C - - (C,E) - (A,C)
graph
utterance prefix: 1
actions
A - B - (A,B) - D - (B,D) - E - (A,E)
graph
selection
utterance prefix: 2
actions
A - B - (A,B) - D - (B,D) - X - (B,X) - C - - (A,C) - E - (C,E)
graph
selection
utterance prefix: 3
actions
A - Y - (A,Y) - D - (Y,D) - E - (Y,E) - C - - (C,E) - (A,C) - Z - (Z,C)
graph
selection
A
B
C
D
E
X
Y
Z

Figure 2: Our framework for simultaneous semantic parsing. Graph nodes and edges are represented as actions in the target (following a depth-first traversal order). Left is the baseline FULLTOGRAPH scenario. For the online scenario, at each prefix position, our model first proposes a full graph which is then pruned based on predicted probabilities. The surviving (selected) nodes, in black, can be executed once their children have returned.

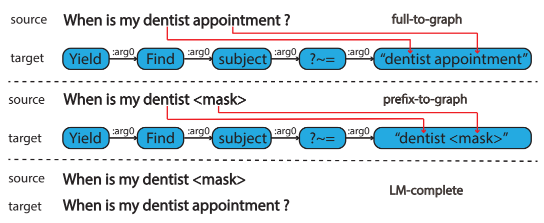


Figure 3: Example of source utterance/prefix and target training data for different models. The red link marks copying from source.

the full graph as the target, rather than attempting to predict only the part of the graph that aligns to the prefix. Hence our method requires no alignment. It tries to predict any function calls that are likely given the prefix, even if they have not been explicitly mentioned yet.

A problem with this setup is that the target graph is often unreachable because it contains string literals that have not been seen yet. This happens when the gold action sequence includes COPYINPUT(j) and j is a position beyond the current prefix. To handle such cases, we modify the target action sequence to instead copy the final position of the prefix, where we have appended a special MASK token as a placeholder for all future tokens. Such a modified training example is shown in the second row of Figure 3. In this way, we disable hallucination of free text by the model, while keeping the graph structure intact with the MASK placeholder.

3.3 Approach 2: LMCOMPLETE then FULLTOGRAPH

Alternatively, propose can first predict the full utterance from the prefix, and use FULLTOGRAPH to parse this completed utterance.[4] Specifically, we fine-tune a pretrained BART model (Lewis et al., 2020) so that it can map an utterance prefix (terminated with the MASK symbol, just as in BART's pre-training recipe) to the full utterance (freely hallucinating content words). As before, the training data includes one example for each prefix of each utterance, so the fine-tuning objective is to maximize the sum of $\log p(\mathbf{u} \mid \mathbf{c}, \mathbf{u}_{[m]})$ over all prefixes $\mathbf{u}_{[m]}$ of all utterances $\mathbf{u}$.

3.4 Subgraph Selection

Let $\hat{G}_m$ be the graph proposed from $\mathbf{u}_{[m]}$. We wish to execute only its probable subgraphs. Recall that we predicted $\hat{G}_m$ by attempting to maximize $\prod_{n=1}^{|\mathbf{a}|} p(a_n \mid \mathbf{c}, \mathbf{u}_{[m]}, \mathbf{a}_{1:n-1})$ (approach 1) or $p(\mathbf{u} \mid \mathbf{c}, \mathbf{u}_{[m]}) \cdot \prod_{n=1}^{|\mathbf{a}|} p(a_n \mid \mathbf{c}, \mathbf{u}, \mathbf{a}_{1:n-1})$ (approach 2). The probability of a subgraph could be obtained by marginalizing over all possible action sequences (and also all completions $\mathbf{u}$ in approach 2), which could be approximated by sampling from the models. For simplicity and efficiency, we instead approximate the probability of a subgraph of $\hat{G}_m$ by the product of the conditional probabilities of the predicted actions that actually built that subgraph[5]—that is, each subgraph of the predicted $\hat{G}_m$ was built by a subset of the predicted actions $\mathbf{a}$. This essentially approximates the marginal probability of the relevant action subsequence by its conditional probability given preceding actions. In practice we found that this simplified heuristic works relatively well, with action-level likelihoods

[4]To avoid training-test mismatch, we could have retrained FULLTOGRAPH to predict the gold graphs from these noisily completed utterances, instead of from the gold utterances. However, this learning problem might be too difficult. Instead, we will consider the uncertainty of completion during select.

[5]In approach 2, this includes the probabilities of the predicted unseen tokens of $\mathbf{u}$. We cannot limit to the tokens that contributed to the subgraph because *all* tokens potentially did so: we do not have an alignment. Thus, when $p(\mathbf{u} \mid \mathbf{c}, \mathbf{u}_{[m]})$ is small, *all* subgraphs will be regarded as uncertain.

being decently calibrated (Section 6.4).

We then select the nodes $v \in \hat{G}_m$ such that the subgraph rooted at v has probability above a constant threshold τ.[6] There are three exceptions: (1) Of course we do not select any node whose subgraph we have previously executed (after predicting and selecting it from a previous prefix). That is unnecessary: we already know the result or are waiting for it. (2) Until the utterance is complete, we do not select any nodes whose function invocations have side effects, as they are unsafe to pre-execute. (In particular, we do not show final results to the user.) (3) But once the utterance is complete, we select *all* unexecuted nodes of the final predicted graph, $\hat{G}$, since now they are both safe and necessary to execute.

4 Evaluation

To quantify the latency improvements for online semantic parsing methods, we propose a new metric, final latency reduction.

4.1 Program Execution Process

We assume that functions can be executed, in parallel, as soon as their arguments are available. Given a graph G, any node $v \in G$ is the root of an executable subgraph. Let $g(v)$ be the time that this subgraph is selected.[7] Let $e(v) \geq 0$ be the time it takes to execute just the function at v on its given arguments.[8] The return time $r(v)$ of node v is

$$r(v) = \max[g(v), \max_{w \in \text{children}(v)} r(w)] + e(v) \quad (1)$$

where children(v) is the set of nodes that return the arguments of v. This is a recursive definition—a node can only be executed after it is selected and all its children (if any) have finished executing—and so $r(v) \geq r(w)$ for $w \in$ children(v). The program G finishes executing at time[9]

$$r(G) = \max_{v \in G} r(v) \quad (2)$$

We assume that our own system's computation time is negligible, so $g(v) = t_m$ if the subgraph rooted at v was predicted and selected from $\mathbf{u}_{[m]}$. In our fully simulated experiments, we set $t_m = m$, which measures time in units of input tokens. These practices follow the simultaneous machine translation literature (Cho and Esipova, 2016; Gu et al., 2017; Ma et al., 2019; Cherry and Foster, 2019). In Section 5, we will also explore using real-time measurements to define t_m.

4.2 Final Latency Reduction

We compute the time at which the system completes executing the *gold* graph G^*, namely $r(G^*)$. Thus, the system cannot achieve a good completion time simply by predicting a small graph. The system's **final latency** is $r(G^*) - t$. Note that $r(G^*) \geq t$, since at least the root node that shows final results to the user has to wait until the utterance is complete (section 3.4).

If the system's final prediction $\hat{G} \neq G^*$, then there may be nodes $v \in G^*$ whose subgraph was never executed. Then $r(G^*) \geq r(v) = \infty$, properly speaking—but we keep it finite by defining $g(v) = t$ for these nodes v. That is, for purposes of latency evaluation, we generously consider the worst case for $v \in G^*$ to be that v is selected for execution when the utterance is complete (rather than that v is *never* executed).

We also compute a baseline: $r_o(G^*)$ is the completion time $r(G^*)$ achieved by the offline parser, which is a *batch* system that sees no prefixes before seeing the full utterance at time t. It is found by setting $g(v) = t$ for all $v \in G^*$ in equations (1)–(2).

We now define our **final latency reduction**

$$\text{FLR} = r_o(G^*) - r(G^*) \geq 0 \quad (3)$$

An *oracle* system would have $g(v) = 0$ for all $v \in G^*$, achieving the best possible final latency of $\max(r_o(G^*) - t, t)$ and the best possible FLR of $\min(t, r_o(G^*) - t)$. This is the FLR upper bound.

4.3 Relationship to Existing Metrics

FLR focuses on how much sooner the user can see results from the target program *after* the user

[6]As Section 3 noted, this strategy is not optimal. All subgraphs with the same probability do have the same risk of being useless work, but they do not all represent the *same amount* of useless work: some incorrect subgraphs require more computation. And they have the same probability of being useful work, but they are not all *equally useful*: some correct subgraphs can be postponed for longer without affecting the overall latency, because they will run more quickly or will not be needed as soon. In both cases, it would be appropriate to raise the subgraph's threshold and wait for more evidence that the subgraph is actually correct.

[7]More precisely, the final time that this happens; it may have previously been selected but not executed (section 3).

[8]$e(v)$ could be modeled as a random variable with some distribution learned from data, so that FLR becomes a random variable whose expectation we would report. In our simulated experiments we model it by a constant Δ for all "slow" function calls, and 0 otherwise.

[9]In the literature on job-shop scheduling (Applegate and Cook, 1991), the quantity $r(G)$ is known as the **makespan**.

has finished speaking. This is different from simultaneous MT, whose focus is how far the target is lagging behind *while* the user is speaking. Therefore, instead of measuring the average over different subprograms, our metric attends to the final completion of the whole program. This allows flexibility in execution order, compared to the translation scenario, where target generation always follows a linear order.

We share with other simultaneous generation applications the assumption that the model inference time is negligible, compared to slower spoken input and program execution (which may involve system and database interactions).

Separate from the final form of our FLR metric, our latency measurement of subprogram return time $r(v)$ can be seen as a generalization of the target time measurement in DAL (Cherry and Foster, 2019) for simultaneous MT. Our program execution time is analogous to the target speaking time in DAL, but DAL operates in a narrower spectrum with a linear chain structured target, and a fixed constant estimate for the target speaking rate.

5 Experimental Setup

Data We make use of two recently released large-scale conversational semantic parsing datasets, SMCalFlow v2.0 (Semantic Machines et al., 2020) and the version of TreeDST (Cheng et al., 2020) released by Platanios et al. (2021). Table 1 and Appendix B provide statistics about the datasets.

Model Training We use the training splits of these datasets to train our FULLTOGRAPH, PREFIXTOGRAPH, and LMCOMPLETE models, and evaluate them on the corresponding validation data. From each training example $(\mathbf{u}, G)$, we extract prefixes of different relative lengths, obtaining $(\mathbf{u}_{0\%}, G), (\mathbf{u}_{10\%}, G), \ldots, (\mathbf{u}_{90\%}, G), (\mathbf{u}_{100\%}, G)$.[10] The prefix-graph pairs of the same percentage length are then stacked to form different training sets, denoted as {prefix0%, prefix10%, ..., prefix90%, prefix100%}. The FULLTOGRAPH parser is trained only using the prefix100% data. For our PREFIXTOGRAPH parser, we experiment with training on different mixtures of the prefix datasets, to quantify the effect on parsing accuracy. For LMCOMPLETE we train on all pairs $(\mathbf{u}', G)$ where $\mathbf{u}'$ is a prefix of $\mathbf{u}$ of *any* length (not limited to the above percentages).

[10]We omit the context $\mathbf{c}$ here as it remains the same.

Model Details All of our parsers are based on the Transformer architecture (Vaswani et al., 2017), adapted to the graph action sequence (see Appendix A). The LMCOMPLETE is based on fine-tuning the pre-trained BART large model (Lewis et al., 2020). One turn of dialogue history is included as the context $\mathbf{c}$. We use greedy decoding for all models. See more details in Appendix E.[11]

Model Evaluation We directly evaluate the parsers FULLTOGRAPH and PREFIXTOGRAPH using exact match accuracy (Semantic Machines et al., 2020; Cheng et al., 2020; Platanios et al., 2021). We also report a finer-grained metric, graph tuple match (Anderson et al., 2016): the F1 score of the set of labeled nodes and labeled edges in the predicted parse. We evaluate LMCOMPLETE using BLEU score (Papineni et al., 2002).

Online Parsing Evaluation For online parsing, we simulate the program execution procedure described in Section 4.1, presenting the system with all prefixes of $\mathbf{u}$ in order: that is, $\mathbf{u}_{[m]} = (u_1, \ldots, u_m)$. We experiment with different probability thresholds τ. For each τ, we report the benefit of our approach as FLR, versus the cost as the number of excess function calls (on top of gold).

When computing FLR, we consider two definitions of t_m: an **intrinsic** one with everything measured by the number of source tokens ($t_m = |\mathbf{u}_{[m]}|$), and an **extrinsic** one with real utterance speaking times in milliseconds. For the latter we recorded human speech data and timing information of the ASR output for 300 randomly sampled examples from SMCalFlow data.[12] When computing FLR, we also assume $e(v) = \Delta$ for all slow function nodes v,[13] and sweep over the constant Δ to see its effects, where Δ is measured either in number of source tokens or in milliseconds.

6 Results and Analysis

6.1 FULLTOGRAPH and LMCOMPLETE

We evaluated our offline parser FULLTOGRAPH and utterance completion model LMCOMPLETE on all prefixes of all utterances in validation data.

[11]Our code is available at `http://aka.ms/simulsp`.

[12]More details in Appendix C. We also extended this evaluation to the full validation data (Appendix D), by using a linear model of t_m fit on recorded speech.

[13]We follow descriptions in `https://github.com/microsoft/task_oriented_dialogue_as_dataflow_synthesis` to identify the set of slow functions. For fast functions, we assume $e(v) = 0$.

Dataset	**SMCalFlow**	**TreeDST**
# utterances in training	121,024	121,652
# utterances in validation	13,496	22,910
Best reported accuracy†	80.4	88.3
FULLTOGRAPH accuracy	80.7	90.8
Prefix BLEU (no completion)	38.04	37.54
LMCOMPLETE BLEU	53.51	55.93

Table 1: Dataset statistics (more in Table 3), offline parser exact-match accuracy, and language model prefix completion performance on corresponding validation data. † both from Platanios et al. (2021).

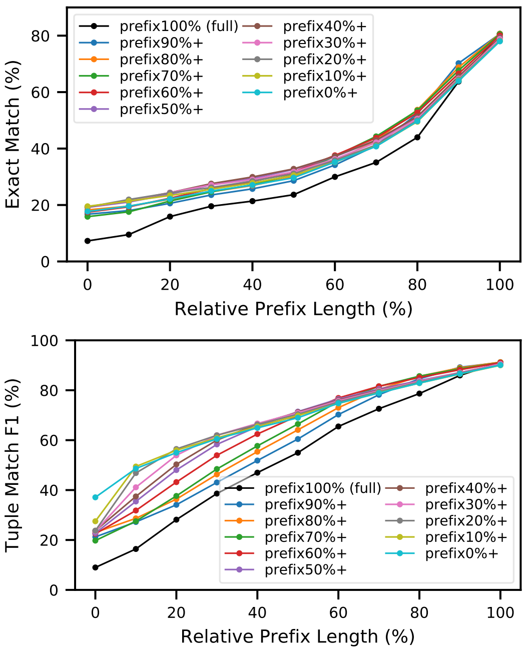


Figure 4: PREFIXTOGRAPH performance on SMCalFlow validation data of varying prefix lengths, by models trained with varying prefix data. E.g., the model with "prefix80%+" is trained on the prefix data with 80%+ relative lengths. We show exact match accuracy on top and graph tuple match F1 scores at the bottom. The larger the area under the curve, the better.

The parser achieves state-of-the-art accuracy on both validation sets. Completing the sentences using our fine-tuned BART model achieves a rather high corpus BLEU score, much higher than if we do not complete them. These models provide a strong foundation for our online parsing methods.

6.2 PREFIXTOGRAPH Quality

In Figure 4 we plot the PREFIXTOGRAPH parser performance when tested on different prefix lengths,[14] with models trained with different mixtures of the prefix training sets. Parsing performance of course degrades for shorter prefixes, but degrades most rapidly for the offline parser (the prefix100% curve). Gradually mixing in shorter prefix data does not affect offline parsing results much (the scores at prefix length 100%, on the top-right), but significantly lifts the curve for earlier prefixes, making the parser better at *anticipating*. The trend is more obvious under the graph tuple match metric, suggesting that PREFIXTOGRAPH succeeds at predicting useful *subgraphs* from short prefixes.

6.3 Final Latency Reduction and Cost

We obtain FLR vs. cost tradeoff curves by varying the threshold τ in our method. Results on the two datasets are shown in Figure 5 under the intrinsic source-timing setup, and results with extrinsic source-timing are shown in Figure 6. The offline approach, FULLTOGRAPH, operates with no latency reduction and no extra calls. The ideal system would have high latency reduction with few excessive function calls, thus the upper left region is desired. We compare our proposed methods with the baseline that directly applies the offline parser on utterance prefixes, which under-performs our methods across all evaluation setups. Between the PREFIXTOGRAPH[15] and LMCOMPLETE + FULLTOGRAPH approaches, we observe that: 1) on SMCalFlow the latter performs better in most cost regions, but on TreeDST they are much closer; 2) when the function execution time Δ is longer, PREFIXTOGRAPH tends to show more advantages in low-cost regions, which is perhaps due to the fact that its early prediction is better when the execution time dominates the source speaking time. Results are similar with the real utterance timing information. Overall, we reduce the final latency by 30%–63%. In fast execution regimes, we obtain 50%–65% of the "best possible" reduction (achieved by the oracle), and 30%–50% in slow execution regimes. Although the FLR metric does not consider model inference time, the LMCOMPLETE + FULLTOGRAPH approach does have higher inference time, since it requires two steps per prefix.

6.4 Analysis and Discussion

PREFIXTOGRAPH Parsing Example In Figure 7, we show the model log-probabilities of individual actions. In (a), the model guesses a complete program structure, but one that finds the next event

[14]This is similar to the latency-BLEU curve in Grissom II et al. (2014) for simultaneous machine translation.

[15]We show results of the model trained with prefix30%+ data, as different training setups result in similar curves.

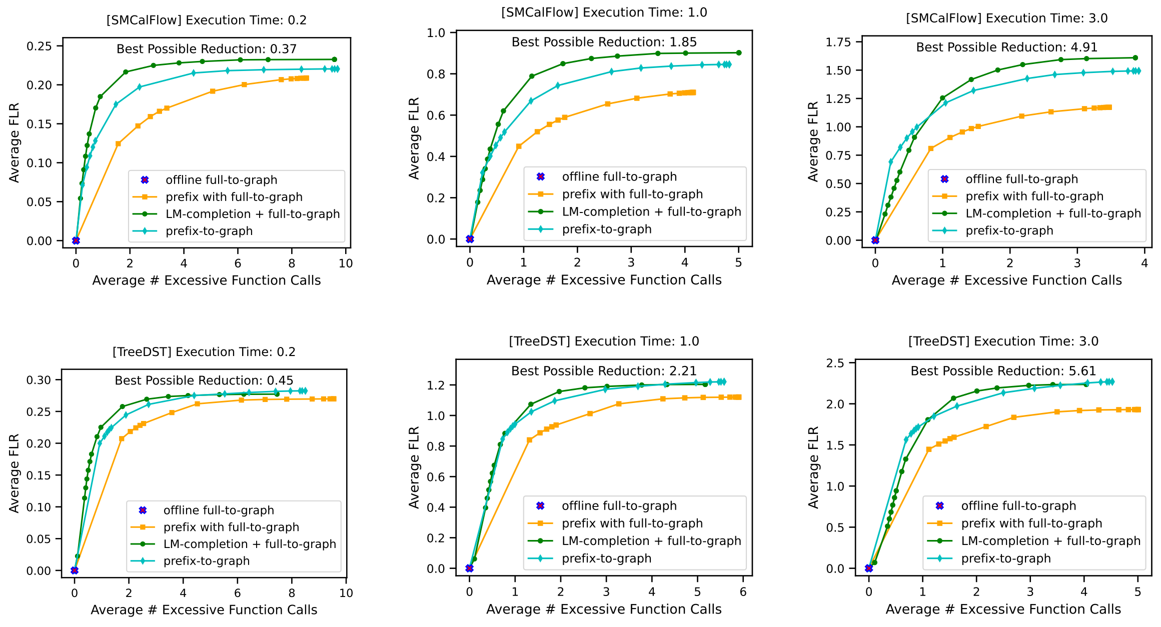


Figure 5: Intrinsic latency reduction vs. cost as we vary the selection threshold τ, with timing measured by number of soure tokens. Columns show different function execution times Δ. The upper left region is desired for better tradeoff. The top row shows results on SMCalFlow dataset, and bottom row is on TreeDST dataset, both on the whole validation data. The average number of gold "slow" function calls (footnote 8) is 1.93 and 2.28, respectively.

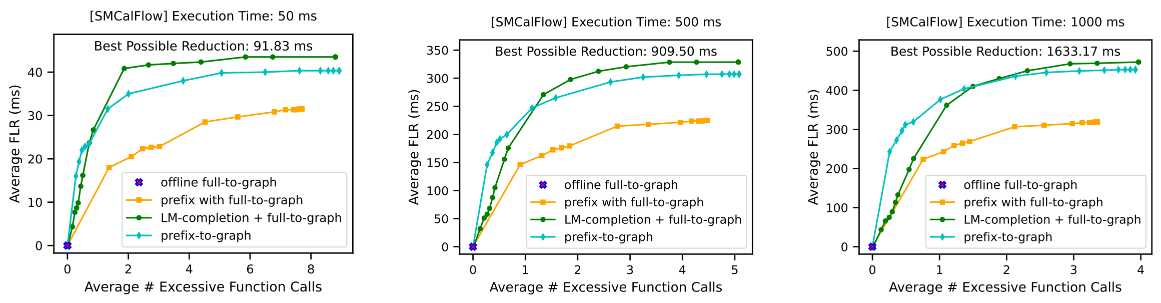


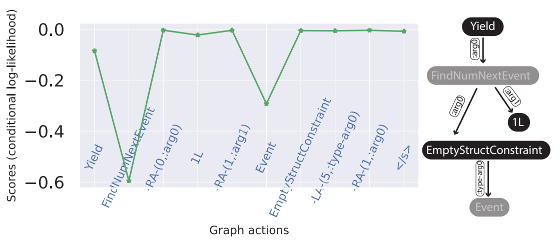
Figure 6: Extrinsic latency reduction vs. cost as we vary the selection threshold τ, with real utterance timing from ASR outputs for 300 examples from SMCalFlow validation data. Columns show different function execution times Δ. The upper left region is desired for better tradeoff. There are on average 1.84 "slow" gold function calls.

instead of finding the supervisor's name. The uncertainty of this guess is reflected in the low probabilities of the actions, and our simple thresholding heuristic can filter out the incorrect subgraphs. But once the new word *"supervisor"* arrives in (b), the model anticipates the correct program even before seeing the final tokens, and all actions have higher scores. Appendix F traces a complete example.

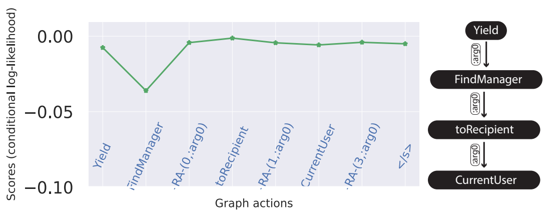
Action-level Probability Calibration In Figure 8 we plot the actual probability of a node's being in the true graph against the (binned) model probability of the action that predicted it. Perfectly calibrated model probabilities would fit the dotted diagonal. Ours are slightly overconfident, likely because they are conditional (on action history), whereas we are treating them as marginal. But they roughly follow the true likelihoods, which empirically justifies our use of action-level probabilities to assess subgraph probabilities.[16]

Latency Reduction per Function We inspect the absolute latency reduction (allowing an earlier finish time than the user utterance) for each function type in Figure 9. The largest gains are obtained for `RecipientWithNameLike` and `FindManager`, likely because invocations of these functions tend to have less structure, often having a string literal as their only argument.

[16]Section 3.4 proposed a *product* of action probabilities. We found that min worked equally well, and used min throughout our experiments (Sections 6.3–6.4).



(a) Prefix: *What is my*



(b) Prefix: *What is my supervisor*

Figure 7: PrefixToGraph's action-level scores and parsed programs for two prefixes of *What is my supervisor 's name ?* Gray nodes have lower action-level scores (whereas in Figure 2, gray nodes were the roots of subgraphs with lower scores).

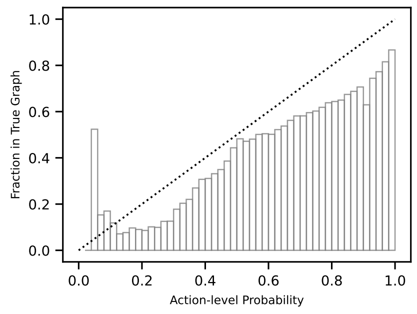


Figure 8: Action-level probability calibration plot from the PrefixToGraph model, over all prefix data.

7 Related Work

An **incremental** algorithm for computing the function f updates $f(x)$ efficiently each time x grows. In this spirit, incremental parsing updates a partial parse or parse chart each time a new word arrives (e.g. Earley, 1970; Huang and Sagae, 2010; Ambati et al., 2015; Damonte et al., 2017). An **online** algorithm *may commit to possibly suboptimal decisions* before it has seen all the input, as in simultaneous MT or online sequence-to-sequence transduction (Jaitly et al., 2016; Yu et al., 2016). By analogy, an online parser might be expected to start *printing the parse* early. However, when we speak of online semantic parsing in this paper, we really mean online semantic *interpretation*—parsing into a program *and* executing that program—and our algorithm starts *executing* early. It commits early to incurring execution costs, but not to any parse (we rapidly reparse each prefix from scratch) nor to any output

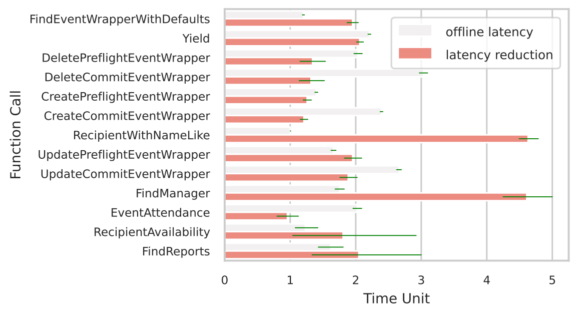


Figure 9: Average latency reduction in SMCalFlow from PrefixToGraph parsing, when $\Delta = 1$ token and τ allows 3 excessive calls. The gray bar shows how long the offline system takes to return a value for this type of function call once the utterance is complete. The pink bar shows the reduction in this latency.

(only side-effect-free functions execute early).

Ma et al. (2019) directly trained a model to generate from source prefixes for simultaneous MT. However, they used a prefix-to-prefix paradigm whereas we trained a prefix-to-full model, in which more aggressive anticipation is not blocked by target reordering. Also, we allow updating the target history by reparsing at each prefix. We masked the unseen source with copying to avoid excessive hallucination in program prediction. Arivazhagan et al. (2020b) adopted a similar idea but only used a crude heuristic to mask the last k target tokens.

More recently, Deng et al. (2021) also explored parsing an utterance prefix into a full program (in their case an SQL query). They focus on saving user effort in formulating questions, while we focus on reducing latency. Accordingly, our task does not stop at predicting the full program; we also decide which subprograms to execute and when.

8 Conclusion

We propose a new task, online semantic parsing, with an accompanying formal evaluation metric, final latency reduction. We show that it is possible to reduce latency by 30%–63% using a strong graph-based semantic parser—either trained to parse prefixes directly or combined with a pre-trained language model for utterance completion—followed by a simple heuristic for subgraph selection. Our general framework can work with different types of parsers and executable semantic representations. In future work, the subgraph selection decisions could be made by a learned model that considers the cost and benefit of each call, instead of using a fixed threshold. The parser could also condition on the execution status, instead of operating separately.

Ethical Considerations

Our paper describes an enabling technology that can expedite a dialogue system's response for a better user experience. It could also assist people who have trouble interacting with the system by reducing their effort in completing the query utterance.

Caution must be taken when pre-executing program calls before the user intent is fully revealed, as there may be an unacceptable cost to mistakenly executing state-changing programs (for example, sending emails or scheduling meetings) without user confirmation. In this work, we only pre-execute "safe" function calls, which retrieve or compute information *without* changing the state of the environment.

Another concern, if training on real user data, is leaking private information to other users. This is especially pressing when predicting with incomplete intent, as the model is encouraged to hallucinate, and may hallucinate information that it has memorized from other users' data. For PREFIXTOGRAPH, we use an explicit MASK token for unrevealed future tokens, and force the model to copy MASK to the predicted program instead of freely generating text. We could easily completely remove the model's ability to hallucinate free text. LMCOMPLETE, on the other hand, can and will leak text from the training data directly into an utterance completion, which can then be copied into a string literal in the predicted program. Thus PREFIXTOGRAPH may be closer to suitable for production use.

References

Bharat Ram Ambati, Tejaswini Deoskar, Mark Johnson, and Mark Steedman. 2015. An incremental algorithm for transition-based CCG parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 53–63, Denver, Colorado. Association for Computational Linguistics.

Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. 2016. SPICE: Semantic propositional image caption evaluation. In *European conference on computer vision*, pages 382–398. Springer.

David Applegate and William Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *INFORMS Journal on Computing*, 3(2):149–156.

Naveen Arivazhagan, Colin Cherry, Wolfgang Macherey, Chung-Cheng Chiu, Semih Yavuz, Ruoming Pang, Wei Li, and Colin Raffel. 2019. Monotonic infinite lookback attention for simultaneous machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1313–1323, Florence, Italy. Association for Computational Linguistics.

Naveen Arivazhagan, Colin Cherry, Wolfgang Macherey, and George Foster. 2020a. Re-translation versus streaming for simultaneous translation. In *Proceedings of the 17th International Conference on Spoken Language Translation*, pages 220–227, Online. Association for Computational Linguistics.

Naveen Arivazhagan, Colin Cherry, Isabelle Te, Wolfgang Macherey, Pallavi Baljekar, and George Foster. 2020b. Re-translation strategies for long form, simultaneous, spoken language translation. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7919–7923. IEEE.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Jianpeng Cheng, Devang Agrawal, Héctor Martínez Alonso, Shruti Bhargava, Joris Driesen, Federico Flego, Dain Kaplan, Dimitri Kartsaklis, Lin Li, Dhivya Piraviperumal, Jason D. Williams, Hong Yu, Diarmuid Ó Séaghdha, and Anders Johannsen. 2020. Conversational semantic parsing for dialog state tracking. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8107–8117, Online. Association for Computational Linguistics.

Colin Cherry and George Foster. 2019. Thinking Slow about Latency Evaluation for Simultaneous Machine Translation. *arXiv:1906.00048 [cs]*. ArXiv: 1906.00048.

Kyunghyun Cho and Masha Esipova. 2016. Can neural machine translation do simultaneous translation? *arXiv preprint arXiv:1606.02012*.

Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

Marco Damonte, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for Abstract Meaning Representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 536–546, Valencia, Spain. Association for Computational Linguistics.

Naihao Deng, Shuaichen Chang, Peng Shi, Tao Yu, and Rui Zhang. 2021. Prefix-to-SQL: Text-to-SQL Generation from Incomplete User Questions. *arXiv:2109.13066 [cs]*. ArXiv: 2109.13066.

J. Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Alvin Grissom II, He He, Jordan Boyd-Graber, John Morgan, and Hal Daumé III. 2014. Don't until the final verb wait: Reinforcement learning for simultaneous machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1342–1352, Doha, Qatar. Association for Computational Linguistics.

Jiatao Gu, Graham Neubig, Kyunghyun Cho, and Victor O.K. Li. 2017. Learning to translate in real-time with neural machine translation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1053–1062, Valencia, Spain. Association for Computational Linguistics.

Hyojung Han, Sathish Indurthi, Mohd Abbas Zaidi, Nikhil Kumar Lakumarapu, Beomseok Lee, Sangha Kim, Chanwoo Kim, and Inchul Hwang. 2021. Faster Re-translation Using Non-Autoregressive Model For Simultaneous Neural Machine Translation. *arXiv:2012.14681 [cs]*. ArXiv: 2012.14681 version: 2.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086, Uppsala, Sweden. Association for Computational Linguistics.

Navdeep Jaitly, Quoc V Le, Oriol Vinyals, Ilya Sutskever, David Sussillo, and Samy Bengio. 2016. An online sequence-to-sequence model using partial conditioning. *Advances in Neural Information Processing Systems*, 29.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692 [cs]*. ArXiv: 1907.11692.

Mingbo Ma, Liang Huang, Hao Xiong, Renjie Zheng, Kaibo Liu, Baigong Zheng, Chuanqiang Zhang, Zhongjun He, Hairong Liu, Xing Li, Hua Wu, and Haifeng Wang. 2019. STACL: Simultaneous Translation with Implicit Anticipation and Controllable Latency using Prefix-to-Prefix Framework. *arXiv:1810.08398 [cs]*. ArXiv: 1810.08398 version: 2.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the Association for Computational Linguistics*.

Emmanouil Antonios Platanios, Adam Pauls, Subhro Roy, Yuchen Zhang, Alexander Kyte, Alan Guo, Sam Thomson, Jayant Krishnamurthy, Jason Wolfe, Jacob Andreas, and Dan Klein. 2021. Value-agnostic conversational semantic parsing. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3666–3681, Online. Association for Computational Linguistics.

Harsh Satija and Joelle Pineau. 2016. Simultaneous machine translation using deep reinforcement learning. *Abstraction in Reinforcement Learning Workshop, ICML 2016*.

Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Lappoon R. Tang and Raymond J. Mooney. 2001. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *Machine Learning: ECML 2001*, pages 466–477, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International*

Conference on Neural Information Processing Systems, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Wolfgang Wahlster. 1993. Verbmobil. In *Grundlagen und anwendungen der künstlichen intelligenz*, pages 393–402. Springer.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Lei Yu, Jan Buys, and Phil Blunsom. 2016. Online segment to segment neural transduction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1307–1316, Austin, Texas. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Luke Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 976–984, Suntec, Singapore. Association for Computational Linguistics.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, page 658–666, Arlington, Virginia, USA. AUAI Press.

Jiawei Zhou, Tahira Naseem, Ramón Fernandez Astudillo, and Radu Florian. 2021a. AMR parsing with action-pointer transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5585–5598, Online. Association for Computational Linguistics.

Jiawei Zhou, Tahira Naseem, Ramón Fernandez Astudillo, Young-Suk Lee, Radu Florian, and Salim Roukos. 2021b. Structure-aware fine-tuning of sequence-to-sequence transformers for transition-based AMR parsing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6279–6290, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

A Graph Generation Model

We encode the graph G as a sequence $\mathbf{a} = (v_1, \mathbf{e}_1, v_2, \mathbf{e}_2, \ldots, v_{|V|}, \mathbf{e}_{|E|})$. This enables us to train a sequence-to-sequence model to predict G. Each element of $\mathbf{a}$ can be regarded as an *action* that adds a vertex or edge to the graph. Each v_k is a vertex, representing a constant value or function invocation, while each $\mathbf{e}_k$ is a subsequence that lists all edges between v_k and earlier vertices.

The vertices $v_1, \ldots, v_{|V|}$ are enumerated in the same order that they appear in the dataset, which is a top-down DFS order.[17] The edges in each $\mathbf{e}_k$ are sorted such that edges from/to more recent vertices come first, i.e., an edge $v_j \xrightarrow{\ell} v_k$ (or $v_j \xleftarrow{\ell} v_k$) precedes an edge $v_i \xrightarrow{\ell'} v_k$ (or $v_i \xleftarrow{\ell'} v_k$) if $i < j$.

Borrowing from the action-pointer mechanism (Zhou et al., 2021a,b), an edge $v_i \xrightarrow{\ell} v_k$ in the subsequence $\mathbf{e}_k$ is represented in the form $\textsc{RightArc}(n, \ell)$ where n is the position in $\mathbf{a}$ such that $\mathbf{a}_n = v_i$. Similarly $v_i \xleftarrow{\ell} v_k$ in the subsequence $\mathbf{e}_k$ is represented in the form $\textsc{LeftArc}(n, \ell)$.

Thus, each node and each edge in G corresponds to one token in $\mathbf{a}$. As an exception, a vertex v_i that is labeled with a string literal is encoded as multiple tokens, e.g., `<str> lunch meeting </str>` (and the vertex's position n is taken to be the position of the initial `<str>` token). Within such a string literal, we include tokens of the form $\textsc{CopyInput}(j)$ action wherever possible, meaning to copy token j of the source sequence. Constructing the program graph from the actions is straightforward—read off the nodes and edges, and append them to the graph. This provides an efficient and compact sequential representation of the graph, with structural well-formedness maintained. An example of the program graph (derived from the original Lispress format[18]) and the action sequence is shown in Table 2.

We model the action sequence generation with a Transformer encoder-decoder network (Vaswani et al., 2017), augmented with two pointer networks—target-side pointing for the edges and source-side pointing for the copied node values. See Appendix E for details. Similar to Zhou et al. (2021a,b), we do not introduce new modules for the pointer networks but directly re-purpose the decoder self-attention head and source-attention head respectively. The actions, edge pointers and copy pointers are supervised together during training and are decoded and combined during inference to reconstruct the complete actions and thus graphs. For each parser, we model $p(G \mid \mathbf{u}, \mathbf{c}) = \prod_{n=1}^{|\mathbf{a}|} p(a_n \mid \mathbf{c}, \mathbf{u}, \mathbf{a}_{1:n-1})$, where the source to the encoder is the concatenation of the context and the full utterance. We decode greedily, incorporating both edge pointers (as in Zhou et al. (2021a)) and source-copy pointers into the action space.

B Prefix Data Length Distribution

We provide basic dataset statistics in Table 3. We further display the length distribution of the complete utterances and their prefixes in Figure 10, for both SMCalFlow and TreeDST training data.

C Real Utterance Timing from ASR

To obtain real utterance speaking timing information for Figure 6, we randomly sampled 300 utterances from the SMCalFlow validation data and recruited two volunteers to each read a portion of the utterances with their normal voice and pace. An ASR system was run to process the 300 recordings and output word segmentations with timing information. The audio was processed using off-the-shelf models in the Microsoft real-time ASR system.[19] Running with the appropriate flags exposes word-level time markings, which allows us to factor recognition latency into the overall computation.

As the ASR outputs are associated with tokens recognized from the human voice, they are not fully consistent with the original utterance text. We map the ASR output tokens back to the original utterance tokens with the dynamic time warping (DTW) algorithm using edit distance as the distance metric. This gives a sequence of segmentation boundaries of the ASR output tokens, where each segmentation contains zero, one, or a few consecutive words with recorded ASR timing information, that align with one token in the original utterance. The times are then mapped back to the original utterance based on the DTW alignments. On average, each word of the original utterance takes 0.386 seconds to

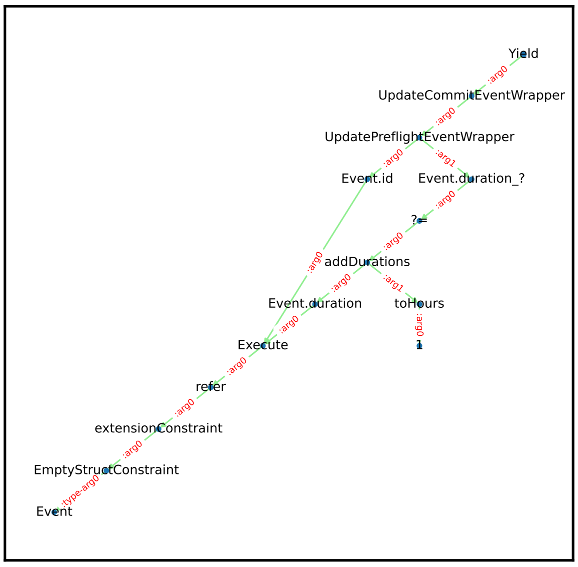
[17] We experimented with other vertex orders, including bottom-up, but found that this made little difference.

[18] `https://github.com/microsoft/task_oriented_dialogue_as_dataflow_synthesis/blob/master/README-LISPRESS.md`.

[19] See documentation at `https://docs.microsoft.com/en-us/azure/cognitive-services/speech-service/spx-overview`.

[Lispress]

```
(let
  (x0
    (Execute
      (refer
        (extensionConstraint
          (^(Event)
            EmptyStructConstraint)))))
  (Yield
    (UpdateCommitEventWrapper
      (UpdatePreflightEventWrapper
        (Event.id x0)
        (Event.duration_?
          (?= (addDurations
                (Event.duration x0)
                (toHours 1))))))))
```



[Actions] (top-down generation order)

```
Yield UpdateCommitEventWrapper -RA-(0,:arg0) UpdatePreflightEventWrapper
-RA-(1,:arg0) Event.duration_? -RA-(3,:arg1) ?= -RA-(5,:arg0) addDurations
-RA-(7,:arg0) toHours -RA-(9,:arg1) 1 -RA-(11,:arg0) Event.duration -RA-(9,:arg0)
Event.id -RA-(3,:arg0) Execute -RA-(17,:arg0) -RA-(15,:arg0) refer -RA-(19,:arg0)
extensionConstraint -RA-(22,:arg0) Event EmptyStructConstraint -LA-(26,:type-arg0)
-RA-(24,:arg0)
```

Table 2: Graph representation and action sequence of the program formulated by our model. `-RA-` represents RIGHTARC and `-LA-` represents LEFTARC. For example, `-RA-(1,:arg0)` constructs an edge between `UpdatePreflightEventWrapper` (the most recent node) and `UpdateCommitEventWrapper` (the node previously generated at action index `1`), with the edge direction being rightward (from previous node to current node) and the label being `:arg0`. The original Lispress format[18] is shown in the upper left. The action sequence is obtained by converting the Lispress into a graph (upper right) and traversing the graph.

Dataset	**SMCalFlow**	**TreeDST**
# utterances in training	121,024	121,652
# utterances in validation	13,496	22,910
Avg. length of full utterance **u**	8.5	8.6
Avg. length of target sequence **a**	22.5	39.1

Table 3: Dataset statistics.

speak. We show the distribution of the voice duration of words in Figure 11. For the corresponding experiments, the real utterance timing of the 300 utterances is used in the execution process for each prefix (whereas the function execution times Δ are still simulated).

D FLR Evaluation with Extrinsic Timings

To simulate realistic speaking rates on the full SMCalFlow and TreeDST corpora, we took a private corpus of 1000 spoken utterances with ASR output, and fit a linear model predicting word duration based on the number of characters in the word. The fit model had the form `len(word) * 0.05502014s + 0.11375083s`. Using this model, we were then able to simulate speaking rates on our full text-only data.

For slow function calls (see footnote 8), we swept over various execution times Δ in milliseconds. Under the extrinsic setting, the FLR vs. cost tradeoff curves for various Δ values are shown in Figure 12.[20] Our proposed methods consistently outperform the baseline with the FULLTOGRAPH directly applied on the utterance prefix. Overall we achieved 30%–63% final latency reduction relative to the offline parser, depending on Δ and τ.

[20]Its top row is similar to Figure 6, but it evaluates on the full SMCalFlow validation dataset, by using simulated utterance timings. Also, it shows a different range of Δ values. The bottom row evaluates on the full TreeDST validation dataset. Thus, Figure 12 evaluates on the same data as Figure 5.

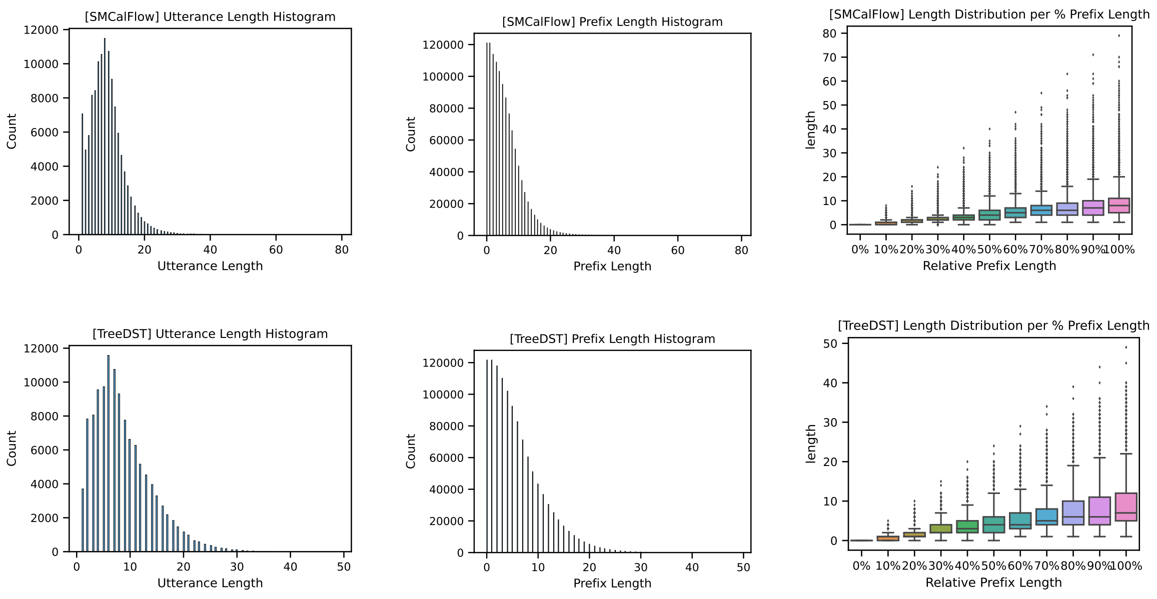


Figure 10: Utterance and prefix length distribution for SMCalFlow (top row) and TreeDST (bottom row) training data. The left column shows the length distribution of the complete utterances, the middle column shows the length distribution of the extracted all possible prefixes, and the right column shows the length distribution for each relative prefix length group (corresponding to our different prefix training subsets {prefix0%, prefix10%, . . ., prefix90%, prefix100%} mentioned in Section 5). Tokenization is considered in all lengths.

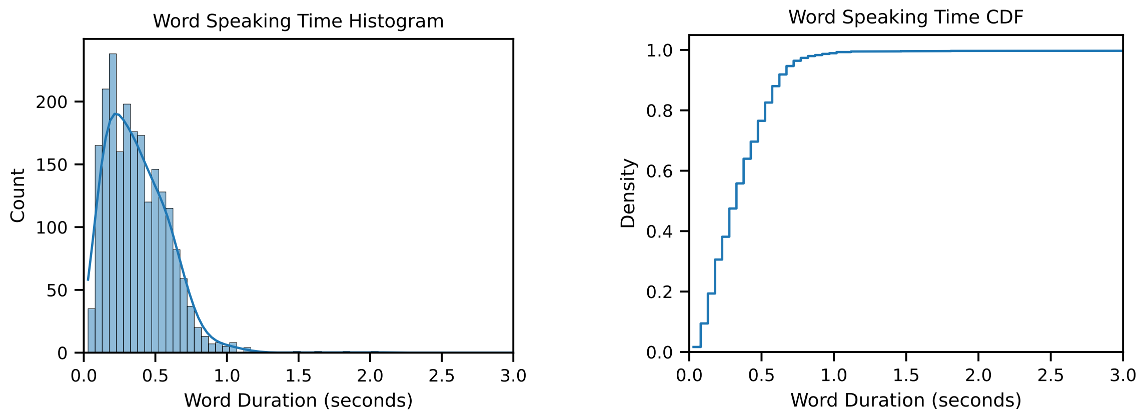


Figure 11: Utterance word voice duration distribution from real ASR outputs, collected from normal human reading voices on 300 randomly sampled utterances in SMCalFlow validation data (2118 words in total, with 0 duration tokens such as punctuation marks removed). The left shows the histogram with the smooth kernel density estimation curve, and the right shows the CDF curve (both with zoomed x-axis to ignore outliers for better visualization). The average voice duration from ASR for a word is 0.386 seconds, and the 0%, 25%, 50%, 75%, 100% quantiles are 0.03, 0.20, 0.34, 0.52, 6.16 seconds, respectively.

E Implementation Details

All of our parsers are based on a 6-layer-4-head Transformer architecture (Vaswani et al., 2017), with 256 hidden dimensions and 512 dimensions for fully connected layers, without additional modules. Pointers for edges are modeled by a self-attention head on the decoder's top layer, and the source copy mechanism is modeled by a cross-attention head of the penultimate decoder layer. The LMComplete model is based on fine-tuning the pre-trained BART large model (Lewis et al., 2020). The context **c** includes 1 previous turn of dialogue history, consisting of the user utterance and agent response. We encode the source (**c** followed by **u**) with the fixed RoBERTa (Liu et al.,

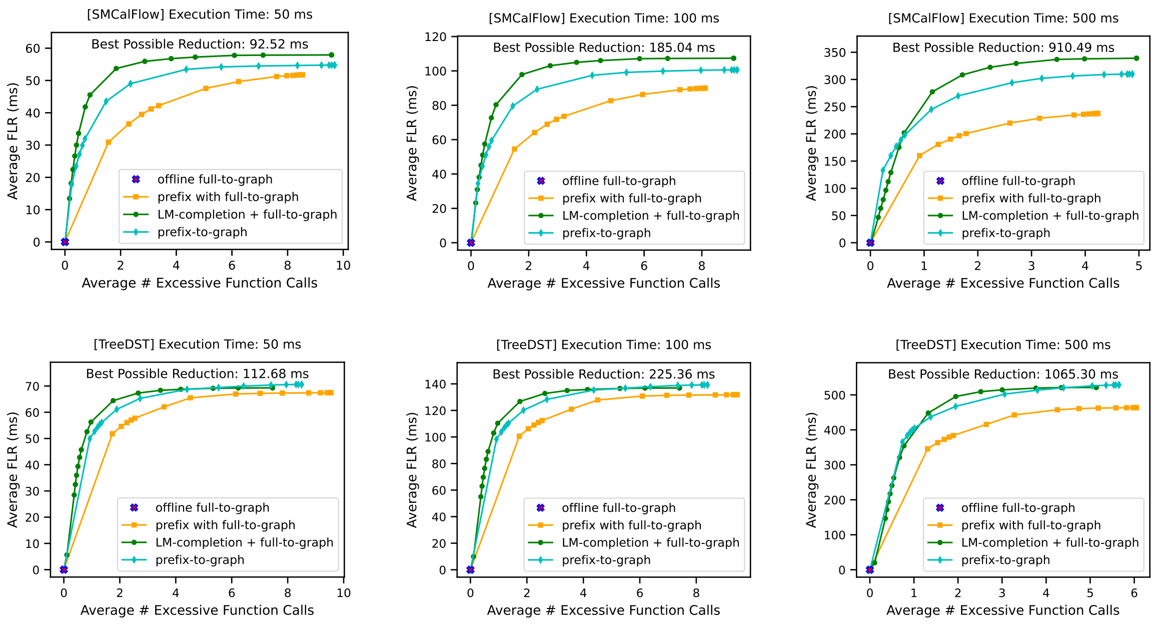


Figure 12: Extrinsic latency reduction vs. cost as we vary the selection threshold τ, with utterance timing estimated as a linear function of length in characters. Columns show different execution times Δ. The upper left region is desired for better tradeoff. The top row shows results on SMCalFlow dataset, and bottom row is on TreeDST dataset, both on the whole validation data. There are on average 1.93 and 2.28 "slow" gold function calls, respectively.

2019) large model for contextualized embeddings (averaged over all RoBERTa layers) to be input to our encoder. For both parsers, FULLTOGRAPH and PREFIXTOGRAPH, we train with the Adam optimizer with batch size 2048 tokens and gradient accumulation of 4 steps. Learning rate is 5e$-$4 for FULLTOGRAPH and 1e$-$4 for PREFIXTOGRAPH as there is more data, both with 4000 warm-up steps using the inverse-sqrt scheduling scheme (Vaswani et al., 2017). They are trained for 50 epochs and we use the last checkpoint for all evaluations without model averaging and ensemble decoding. Training takes about 3 hours on a single Nvidia Titan RTX GPU with 24 GB memory with floating-point 16 mixed precision training for the FULLTOGRAPH parser, and the time increases proportionally for PREFIXTOGRAPH with the prefix data size when combining different prefix lengths. For the LMCOMPLETE model, we fine-tune BART-large for 12 epochs and take the best checkpoint, following the standard recipe from FAIRSEQ. For all the models, we use greedy decoding at inference time. All the models are implemented and trained with the FAIRSEQ toolkit (Ott et al., 2019).

F Example of Parsing and Execution

Table 4 illustrates the behavior of our PREFIXTOGRAPH system on an utterance from the SMCalFlow dataset, showing the detailed parsing and execution process for each step. The utterance timings in milliseconds were obtained from real ASR output on a human-spoken version of the utterance. The program graphs are printed in the Lispress format.[18]

We implicitly add an extra root node to the top of every graph G, representing the function that shows the result to the user. Since this function has side effects, it can only be selected for execution after time t, when the utterance is complete (see Section 3.4). As a result, the graph's completion time $r(G) \geq t$. However, this root node is not shown in our graphs—and in this dataset, the remaining nodes are safe since any other side effects (such as adding the requested event to the calendar) are actually deferred until the root node executes. As a result, all nodes that are shown in Table 4 can be selected at any time.

At each time when a new prefix token arrives, our model first proposes a full graph and then uses a thresholding heuristic to select more probable subgraphs (marked as blue). The threshold is $\tau = \exp(-1.0) \approx 0.368$ in this example. In our experiments, the score of a subgraph is computed as the minimum action-level probability among all actions that construct the subgraph (footnote 16). Hence if a larger subgraph is selected, all its subgraphs must have also been selected.

F.1 Simulated Execution Process

The program execution process (simulated) happens in the background as the user speaks. At the end of the execution the benefit (final latency reduction) is known, along with its cost (count of excessive function calls). We review here some details about the execution process that were already explained in the main paper:

- We assume a constant execution time Δ for all non-trivial ("slow") function calls. All other graph nodes are assumed to have 0 execution time.
- The execution of a selected function must wait until all its dependent functions have finished executing.[21]
- Identical subgraphs that are predicted at different times are treated as the same: there is no need to execute both of them.[22] (E.g., the predicted graphs after seeing the token *friday* and after seeing the token *for* are considered to be the same graph.)
- The selected set is refreshed every time when a new prefix $\mathbf{u}_{[m]}$ arrives, as the new prefix provides new information that may change the predicted subgraphs or their probabilities. Therefore, unexecuted calls in the old selection will be discarded: e.g., after the token *night* arrives, the previously selected functions are no longer selected in the new predicted graph.
- Despite this, ongoing executions are never interrupted or canceled.
- Our method does not affect the accuracy of the semantic parser, but only its latency. This is because our final result is derived from the root node of our parse of the complete utterance $\mathbf{u}$, just as it was for the offline parser.[23] The earlier parses based on prefixes are only used for speculative pre-execution of subprograms, whose results will only be used if they appear in the final parse.
- However, the FLR metric does reflect accuracy to some degree. Recall that it is computed on the gold program calls. In the example, our parser predicts the gold graph G^* successfully—but when it does not, our FLR metric will still require the system to execute any remaining nodes of G^* once the utterance completes. A consequence is that FLR generally suffers somewhat when the parser is inaccurate, since subgraphs of G^* that are never predicted cannot be pre-executed to reduce final latency; indeed, they must be correctly predicted *before* t in order to be pre-executed.

Following the data descriptions in `https://github.com/microsoft/task_oriented_dialogue_as_dataflow_synthesis`, we consider the following function calls to be non-trivial ("slow") function calls. They are marked with colored background in the graphs when they are selected.

SMCalFlow: `Yield`, `RecipientAvailability`, `FindReports`, `FindManager`, `UpdatePreflightEventWrapper`, `CreatePreflightEventWrapper`, `DeletePreflightEventWrapper`, `FindEventWrapperWithDefaults`, `RecipientWithNameLike`, `DeleteCommitEventWrapper`, `UpdateCommitEventWrapper`, `CreateCommitEventWrapper`, `EventAttendance`.

[21] In future work, however, we should relax this requirement to allow short-circuit evaluation. A node can sometimes be executed when some but not all of its children have returned. Examples include if-then-else once the boolean condition has returned, a multiplication when one argument has returned with a zero value, or any function when one argument has returned with an exception that is not caught by the function. (Exceptions can be regarded as special return values, as described by Semantic Machines et al. (2020).)

[22] A predicted node $v \in \hat{G}_m$ has a well-defined return value provided that it is the root of a sub-DAG. Any other predicted node that is the root of an identical sub-DAG can be rapidly identified, for example by the technique of hash consing, allowing it to share this return value. (Ordinarily v is indeed the root of a sub-DAG; indeed, the decoder can optionally ensure this by disallowing edge actions that would create a cycle in $\hat{G}_m$, on the grounds that G^* is acyclic.)

[23] It is true that our first approach (Section 3.2) parses $\mathbf{u}$ using a different parser than the offline parser—it uses PREFIXTOGRAPH rather than FULLTOGRAPH—which could have a slight effect on accuracy. If it had harmed accuracy, then we would have had our online parser switch to using FULLTOGRAPH at the final step when it finally sees the complete utterance $\mathbf{u}$. We did not do this because Figure 4 shows that in practice, the various PREFIXTOGRAPH parsers are as accurate as FULLTOGRAPH on complete utterances.

TreeDST: `plan`, `Create`, `Find`, `Update`, `Delete`, `Book`, `CheckExistence`, `reference`, `revise`, `refer`, `someSalient`.

F.2 Discussion of the Example

In the example, the function execution time (500 ms) spans 1 to 2 utterance token times. We could expect faster execution time to result in an earlier finish time.

Our approach achieves an FLR of 880 ms (see the very end of Table 4). This is largely due to the fact that we predict the correct program before the utterance ends, at prefix steps *for*, *2*, and *hours*. The PREFIXTOGRAPH model learns to anticipate the correct information even before seeing it. For example, *2 hours* is predicted in the program as the duration even before seeing the last two tokens *2 hours*, and *friday* and *7 pm* are predicted after seeing only *Add date night for next _*. Therefore, these function calls can be successfully pre-executed in the background, reducing the final latency of the response.

Not all of our speculative executions were useful. We executed 4 excess functions that are not in the gold program, which induces a computational cost. These were due to overconfident early predictions, when we lacked most of the information in the utterance. Future work could attempt to improve selection by better assessing the probability, expected cost, and expected future benefit of each speculative execution (see footnotes 6 and 21).

An interesting phenomenon is that as new words arrive, subgraphs are often corrected to incorporate the new information. In our example, consider how the graph is updated after the token *for*: the call to `CreatePreflightEventWrapper` is almost identical, but a (guessed) duration has been added. Our present system would execute a whole new call to `CreatePreflightEventWrapper`, discarding the results of the previous call. However, as the change to the call was small, a future opportunity would be to keep and modify the result of the previous call, which might be faster than executing a new call from scratch.

The execution ① invokes the `PersonName.apply` function on an argument that is the special token MASK (see Section 3.2). As MASK denotes a value that is not yet known from the prefix, we define such function invocations to return an exception value (see footnote 21).

Indeed, *any* subgraph that contains MASK—such as all of the selected subgraphs at the time of ①—is guaranteed to return an exception value. These subgraphs are also guaranteed to be wrong, as G^* never contains MASK, so we could have assigned them probability 0, overriding the heuristic of Section 3.4. This would have reduced our execution cost—that is, the number of excessive function calls.

Table 4: An example of simultaneous parsing and pre-execution from our PREFIXTOGRAPH system, for the utterance *Add date night for next friday at 7 for 2 hours*. For this particular example, we achieve FLR of 880 ms with 4 excess function calls. For subgraph selection a constant threshold of $\tau = \exp(-1.0) \approx 0.368$ is used. The source prefix timing is from real ASR outputs of human speaking (Appendix C): the "DUR" column shows token duration, and the "Time" column shows the total time elapsed so far. We assume that non-trivial function calls execute in $\Delta = 500$ time. We use circled numbers (e.g. ①, ②, $\cdots$) to denote unique function calls being executed (`in exec.` means ongoing execution, `done` means finished execution). The selected set of subgraphs for execution is refreshed every time a new prefix token arrives, but ongoing executions are allowed to finish.

Prefix token	DUR (ms)	Time (ms)	Proposed Graph, Subgraph Selection, and Executable Function Calls (non-zero exec. time)	Execution Status
∅	0	0	`(GenericPleasantry)`	`Selection Refresh` ∅
Add	390	390	`(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper (& (Event.subject_? (?= "<mask>")) (Event.start_? (DateTime.date_? (?= (NextDOW (Friday)))))))))`	`Selection Refresh` ∅
date	320	710	`(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper (& (& (Event.subject_? (?= "date")) (Event.start_? (DateTime.date_? (?= (NextDOW (Friday)))))) (Event.attendees_? (AttendeeListHasRecipient (Execute (refer (extensionConstraint (RecipientWithNameLike ^(Recipient) EmptyStructConstraint) (PersonName.apply "<mask>")))))))))))`	`Selection Refresh` `(RecipientWithNameLike...)` `(CreatePreflightEventWrapper...)` `(CreateCommitEventWrapper...)` `(Yield...)` `Start executing ①` `(RecipientWithNameLike...)` `Selected Set` `(CreatePreflightEventWrapper...)` `(CreateCommitEventWrapper...)` `(Yield...)`
night	370	1080	`(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper (& (& (Event.subject_? (?= "date night")) (Event.start_? (?= (DateAtTimeWithDefaults (NextDOW (Friday)) (NumberPM 7L))))) (Event.attendees_? (AttendeeListHasRecipient (Execute (refer (extensionConstraint (RecipientWithNameLike` `[in exec.①]` `(^(Recipient) EmptyStructConstraint) (PersonName.apply "<mask>")))))))))))`	`Selection Refresh` ∅
		1210		`Finish executing ①` `(RecipientWithNameLike...)`

(To be continued next page)

Prefix token	DUR (ms)	Time (ms)	Proposed Graph, Subgraph Selection, and Executable Function Calls (non-zero exec. time)	Execution Status
for	290	1370	(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper (& (Event.subject_? (?= "date night")) (Event.start_? (?= (DateAtTimeWithDefaults (NextDOW (Friday)) (NumberPM 7L)))))))	Selection Refresh (CreatePreflightEventWrapper...) (CreateCommitEventWrapper...) (Yield...) Start executing ② (CreatePreflightEventWrapper...) Selected Set (CreateCommitEventWrapper...) (Yield...)
next	330	1700	(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper (& (Event.subject_? (?= "date night")) (Event.start_? (DateTime.date_? (?= (NextDOW (Friday))))))))	Selection Refresh ∅
		1870		Finish executing ② (CreatePreflightEventWrapper...)
friday	630	2330	(Yield (CreateCommitEventWrapper (CreatePreflightEventWrapper [done ②] (& (Event.subject_? (?= "date night")) (Event.start_? (?= (DateAtTimeWithDefaults (NextDOW (Friday)) (NumberPM 7L)))))))	Selection Refresh (CreateCommitEventWrapper...) (Yield...) Start executing ③ (CreateCommitEventWrapper...) Selected Set (Yield...)
at	200	2530	(Yield (CreateCommitEventWrapper [in exec.③] (CreatePreflightEventWrapper [done ②] (& (Event.subject_? (?= "date night")) (Event.start_? (?= (DateAtTimeWithDefaults (NextDOW (Friday)) (NumberPM 7L)))))))	Selection Refresh (Yield...)
		2830		Finish executing ③ (CreateCommitEventWrapper...) Start executing ④ (Yield...) Selected Set ∅

(To be continued next page)

Prefix token	DUR (ms)	Time (ms)	Proposed Graph, Subgraph Selection, and Executable Function Calls (non-zero exec. time)	Execution Status

7	590	3120	`(Yield` [in exec.④]
			`(CreateCommitEventWrapper` [done ③]
			`(CreatePreflightEventWrapper` [done ②]
			`(&`
			`(Event.subject_? (?= "date night"))`
			`(Event.start_?`
			`(?=`
			`(DateAtTimeWithDefaults`
			`(NextDOW (Friday))`
			`(NumberPM 7L))))))))`
				`Selection Refresh`
				∅

for	210	3330	`(Yield`
			`(CreateCommitEventWrapper`
			`(CreatePreflightEventWrapper`
			`(& (&`
			`(Event.subject_? (?= "date night"))`
			`(Event.start_?`
			`(?=`
			`(DateAtTimeWithDefaults`
			`(NextDOW (Friday))`
			`(NumberPM 7L)))))`
			`(Event.duration_? (?= (toHours 2)))))))`
				`Finish executing ④`
				`(Yield...)`
				`Selection Refresh`
				`(CreatePreflightEventWrapper...)`
				`(CreateCommitEventWrapper...)`
				`(Yield...)`
				`Start executing ⑤`
				`(CreatePreflightEventWrapper...)`
				`Selected Set`
				`(CreateCommitEventWrapper...)`
				`(Yield...)`

2	320	3650	`(Yield`
			`(CreateCommitEventWrapper`
			`(CreatePreflightEventWrapper`[in exec.⑤]
			`(& (&`
			`(Event.subject_? (?= "date night"))`
			`(Event.start_?`
			`(?=`
			`(DateAtTimeWithDefaults`
			`(NextDOW (Friday))`
			`(NumberPM 7L)))))`
			`(Event.duration_? (?= (toHours 2)))))))`
				`Selection Refresh`
				`(CreateCommitEventWrapper...)`
				`(Yield...)`

		3830		`Finish executing ⑤`
				`(CreatePreflightEventWrapper...)`
				`Start executing ⑥`
				`(CreateCommitEventWrapper...)`
				`Selected Set`
				`(Yield...)`

hours	560	4210	`(Yield`
			`(CreateCommitEventWrapper` [in exec.⑥]
			`(CreatePreflightEventWrapper` [done ⑤]
			`(& (&`
			`(Event.subject_? (?= "date night"))`
			`(Event.start_?`
			`(?=`
			`(DateAtTimeWithDefaults`
			`(NextDOW (Friday))`
			`(NumberPM 7L)))))`
			`(Event.duration_? (?= (toHours 2)))))))`
				`Selection Refresh`
				`(Yield...)`

(To be continued next page)

Prefix token	DUR (ms)	Time (ms)	Proposed Graph, Subgraph Selection, and Executable Function Calls (non-zero exec. time)	Execution Status

		4330		`Finish executing` ⑥
				`(CreateCommitEventWrapper...)`
				`Start executing` ⑦
				`(Yield...)`
				`Selected Set`
				∅

		4830		`Finish executing` ⑦
				`(Yield...)`
				`Finish All`

Offline Base System

Prefix token	DUR (ms)	Time (ms)	Gold Graph, and Executable Function Calls (non-zero exec. time)	Execution Status
∅	0	0		
Add	390	390		
⋮	⋮	⋮		
for	210	3330		
2	320	3650		

hours	560	4210	`(Yield`
			`(CreateCommitEventWrapper`
			`(CreatePreflightEventWrapper`
			`(& (&`
			`(Event.subject_? (?= "date night"))`
			`(Event.start_?`
			`(?=`
			`(DateAtTimeWithDefaults`
			`(NextDOW (Friday))`
			`(NumberPM 7L)))))`
			`(Event.duration_? (?= (toHours 2)))))))`
				`Selection Refresh`
				`(CreatePreflightEventWrapper...)`
				`(CreateCommitEventWrapper...)`
				`(Yield...)`
				`Start executing` ①
				`(CreatePreflightEventWrapper...)`
				`Selected Set`
				`(CreateCommitEventWrapper...)`
				`(Yield...)`

		4710		`Finish executing` ①
				`(CreatePreflightEventWrapper...)`
				`Start executing` ②
				`(CreateCommitEventWrapper...)`
				`Selected Set`
				`(Yield...)`

		5210		`Finish executing` ②
				`(CreateCommitEventWrapper...)`
				`Start executing` ③
				`(Yield...)`
				`Selected Set`
				∅

		5710		`Finish executing` ③
				`(Yield...)`
				`Finish All`

(To be continued next page)

Final Latency Reduction (FLR)

Utterance Finish (ms)		Gold Graph & Finish time (ms) of Executable Function Calls (non-zero execution time)	Execution Finish (ms)	Final Latency (ms)	# Calls

4210	Online System	`(Yield [⑦ Finished @ 4830]`
		`  (CreateCommitEventWrapper [⑥ Finished @ 4330]`
		`    (CreatePreflightEventWrapper [⑤ Finished @ 3830]`
		`      (& (&`
		`        (Event.subject_? (?= "date night"))`
		`        (Event.start_?`
		`          (?=`
		`            (DateAtTimeWithDefaults (NextDOW (Friday))`
		`              (NumberPM 7L)))))`
		`      (Event.duration_? (?= (toHours 2)))))))`
			4830	620	7

	Offline System	`(Yield [③ Finished @ 5710]`
		`  (CreateCommitEventWrapper [② Finished @ 5210]`
		`    (CreatePreflightEventWrapper [① Finished @ 4710]`
		`      (& (&`
		`        (Event.subject_? (?= "date night"))`
		`        (Event.start_?`
		`          (?=`
		`            (DateAtTimeWithDefaults (NextDOW (Friday))`
		`              (NumberPM 7L)))))`
		`      (Event.duration_? (?= (toHours 2)))))))`
			5710	1500	3

FLR		1500 - 620 = 880 ms (59% reduction)			