Leveraging Code to Improve In-context Learning for Semantic Parsing

Ben Bogin[1*] **Shivanshu Gupta**[2*] **Peter Clark**[1] **Ashish Sabharwal**[1]

[1]Allen Institute for AI [2]University of California Irvine
{benb,peterc,ashishs}@allenai.org, shivag5@uci.edu

Abstract

In-context learning (ICL) is an appealing approach for semantic parsing due to its few-shot nature and improved generalization. However, learning to parse to rare domain-specific languages (DSLs) from just a few demonstrations is challenging, limiting the performance of even the most capable LLMs.

In this work, we show how pre-existing coding abilities of LLMs can be leveraged for semantic parsing by (1) using general-purpose programming languages such as Python instead of DSLs and (2) augmenting prompts with a structured domain description that includes, e.g., the available classes and functions. We show that both these changes significantly improve accuracy across three popular datasets; combined, they lead to dramatic improvements (e.g., 7.9% to 66.5% on SMCalFlow compositional split) and can substantially improve compositional generalization, nearly closing the performance gap between easier i.i.d. and harder compositional splits. Finally, comparisons across multiple PLs and DSL variations suggest that the similarity of a target language to general-purpose code is more important than prevalence in pretraining corpora. Our findings provide an improved methodology for building semantic parsers in the modern context of ICL with LLMs.[1]

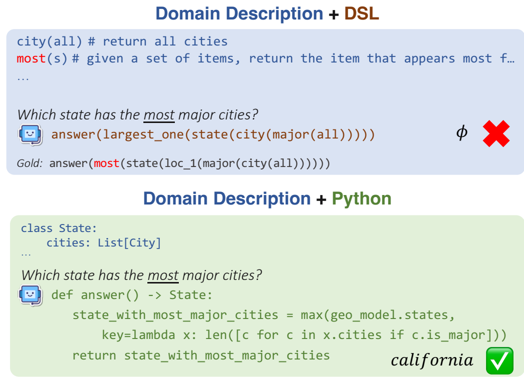

Figure 1: An example illustrating how moving the problem space from a DSL to a general-purpose programming language such as Python can improve output accuracy. When prompted with a DSL, the model doesn't use the operator `most`, resulting in an incorrect program. When prompted with Python, the model leverages its pre-existing knowledge of coding to produce the correct program and answer.

1 Introduction

Semantic parsing, the task of translating natural language utterances to structured meaning representations (Zelle and Mooney, 1996; Kate et al., 2005) is a core requirement for building task-oriented dialog systems and voice assistants. This task is primarily addressed with two approaches: fine-tuning models on labeled datasets of utterances mapped to domain-specific language (DSL) programs (Xu et al., 2020; Oren et al., 2021; Gupta et al., 2022; Yin et al., 2022) and employing in-context learning (ICL; Brown et al., 2020) to prompt a large language model (LLM) with a few demonstrations.

However, both strategies present significant limitations. Fine-tuning requires substantial pools of labeled data, which can be expensive and time-consuming to obtain. Crucially, fine-tuned models also struggle to compositionally generalize, e.g., to decode programs longer than seen during training or to emit unseen structures (Kim and Linzen, 2020; Keysers et al., 2020; Bogin et al., 2022; Yao and Koller, 2022). While ICL can improve compositional generalization in some cases (Anil et al., 2022; Qiu et al., 2022b; Drozdov et al., 2023; Hosseini et al., 2022), learning from a few demonstrations is challenging: LLMs need to not only understand the meaning of the input utterance but also learn how to correctly use a typically rare domain-specific language (DSL), given only few demonstrations. This makes ICL sensitive to demonstration selection (Zhao et al., 2021), which may not cover

[*] Equal contribution
[1] https://github.com/allenai/code-semparse

Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 4971–5012
June 16-21, 2024

all functionalities and subtleties of a DSL. While prior work has tried to alleviate this with a better selection of demonstration (Liu et al., 2022; Levy et al., 2023; Gupta et al., 2023), such approaches require access to a large pool of labeled demonstrations to select from and are not applicable in a *true* few-shot settings.

Given that LLMs show remarkable coding abilities in general-purpose programming languages (PLs; Chen et al. 2021; Xu et al. 2022), in this work, we ask two main questions: (1) How can we leverage these abilities to improve ICL-based semantic parsing? (2) Can LLMs compositionally generalize better with PLs rather than DSLs?

To investigate this, first, we replace DSLs with equivalent code written in *popular programming languages* such as Python or Javascript. This helps better align the output space with pretraining corpora, obviating the need for LLMs to learn new syntax, basic operations, or other coding practices from scratch. For example, consider Figure 1: to select a state that has the *most* major cities, an LLM prompted with a DSL needs to use the operator `most`, for which it might not be given an example. In contrast, with Python, the LLM can leverage its pre-existing knowledge of code to find such a state.

Second, we augment the ICL prompt with a structured description of the output meaning representation, which we refer to as *Domain Description (DD)*. This provides domain-specific information such as types of entities, attributes, and methods (e.g., `State` and its attributes in Figure 1). While such descriptions can also be added to DSLs, we find that domain descriptions for PLs are easier to precisely define with explicit declarations of objects, methods, their signatures, etc. Furthermore, LLMs are more likely to leverage descriptions with PLs rather than DSLs, as using previously defined objects and methods is a common coding practice.

We evaluate our approach on both ChatGPT[2] and the open-source Starcoder model (Li et al., 2023a), by implementing Python-executable environments for three complex semantic parsing benchmarks, namely GeoQuery (Zelle and Mooney, 1996), Overnight (Wang et al., 2015), and SMCalFlow (Andreas et al., 2020), and annotating them with Python programs and DDs.

In a true few-shot setting, where only a few (e.g., 10) labeled examples are available to use as demonstrations, we find that PL prompts with DDs dramatically improve execution-based accuracy across the board, e.g., 49.7 points absolute improvement (31.0% to 80.7%) on the length split of GeoQuery, compared to the standard ICL approach of a DSL-based prompt with no DD. Prompting a model with Python and domain description can often even eliminate the need for many demonstrations: with just a *single* demonstration, accuracy on a compositional split of GeoQuery reaches 80%, compared to 17% for DSL prompting with no DD. In fact, for two datasets, a single PL demonstration with DD outperforms DSL prompts with as many as 25 demonstrations and an equivalent DD. Interestingly, we find that employing Python with a DD substantially improves compositional generalization, almost entirely closing the compositionality gap, i.e., the performance difference between an i.i.d. split and harder compositional splits.

One might hypothesize that the strong performance of Python is due to its prevalence in the pretraining corpus (Cassano et al., 2023). To investigate this, we evaluate the performance of PLs whose popularity differs from that of Python. Surprisingly, we find that prevalence in pretraining corpora does not explain superiority: both Scala, a PL much rarer than Python, and Javascript, which is much more prevalent, perform roughly similarly. SQL, a common query language, performs better than DSLs, but worse than the other more general-purpose PLs. Further analyses with simplified versions of DSLs indicate that even rare DSLs, as long as they resemble general-purpose code, might perform nearly as well as PLs, provided a detailed DD is used.

In conclusion, we demonstrate that using popular PLs instead of DSLs and adding domain descriptions dramatically improves ICL for semantic parsing while nearly closing the compositionality gap. Further, we show that when LLMs are used for semantic parsing, it is better to either prompt them with PLs or design DSLs to resemble popular PLs. Overall, these findings suggest an improved way of building semantic parsing applications in the modern context of in-context learning with LLMs.

2 Related Work

Compositional Generalization. Semantic parsing has been studied extensively in recent years in the context of compositional generalization (CG), where models are evaluated on examples that contain unseen compositions of structures, rather than

[2] `https://chat.openai.com/`

Dataset	MR	# chars	Depth	Example
GeoQuery	*input*			*How high is the highest point in the largest state?*
	FunQL	49.4	4.8	`answer(elevation_1(highest(place(loc_2(largest(state(all)))))))`
	Python	115.4		`largest_state = max(geo_model.states, key=lambda x:  x.size)` `return largest_state.high_point.elevation`
Overnight	*input*			*person whose gender is male and whose birthdate is 2004*
	λ-DCS	282.0	6.8	`(call SW.listValue (call SW.filter (call SW.filter (call SW.getProperty (call SW.singleton en.person) (string !type)) (string gender) (string =) en.gender.male) (string birthdate) (string =) (date 2004 -1 -1)))`
	λ-DCS (Simp.)	164.1	6.0	`(listValue (filter (filter (getProperty en.person !type) gender = en.gender.male) birthdate = 2004))`

	Python	270.0		`people_born_in_2004 = [p for p in api.people if p.birthdate == 2004]`
				`males_born_in_2004 = [p for p in people_born_in_2004 if p.gender == Gender.male]`
				`return males_born_in_2004`

SMCalFlow	*input*			*Make an appointment in Central Park on Friday.*
	Dataflow	372.6	8.7	`(Yield :output (CreateCommitEventWrapper :event (CreatePreflightEventWrapper :constraint (Constraint[Event] :location ( ?= # (LocationKeyphrase "Central Park")) :start (Constraint[DateTime] :date ( ?= (NextDOW :dow # (DayOfWeek "FRIDAY"))))))))`
	Dataflow (Simp.)	118.7	4.2	`CreateEvent( AND( at_location( Central Park ) , starts_at( NextDOW( FRIDAY ) ) ) )`
	Python	174.4		`api.add_event(Event(subject="Appointment in Central Park", starts_at=[ DateTimeClause.get_next_dow(day_of_week="Friday")], location="Central Park"))`

Table 1: Sample input, program, average program length and average maximum depth for each dataset and meaning representation considered. Depth is computed based on parentheses.

easier i.i.d. train-test splits (Finegan-Dollak et al., 2018). Initial work on CG focused on fine-tuning-based approaches. As simply scaling the model size or amount of data CG has been shown to be insufficient for improving CG (Hosseini et al., 2022; Qiu et al., 2022b), prior work explored approaches like specialized architectures, (Herzig and Berant, 2021; Bogin et al., 2021; Yin et al., 2021; Lindemann et al., 2023), data augmentation (Andreas, 2020; Qiu et al., 2022a; Akyürek et al., 2021), training data selection (Bogin et al., 2022; Gupta et al., 2022), etc. With the increasing prevalence of in-context learning with LLMs, recent works have focused on improving its compositional generalization through better demonstration selection (Liu et al., 2022; Ye et al., 2023; An et al., 2023; Li et al., 2023b; Zhang et al., 2022; SU et al., 2023; Levy et al., 2023; Gupta et al., 2023, 2024). However, all these methods require a large pool of demonstrations or annotation efforts. In contrast, we show that by leveraging pre-existing coding abilities, LLMs do not need as many examples to generalize.

Effect of Meaning Representations. To address specific challenges with DSLs, previous work has proposed to work with simpler meaning representations (MRs) (Herzig et al., 2021; Li et al., 2022; Wu et al., 2023) or synthetic NL utterances (Shin et al., 2021), or prompting models with the grammar of the DSL (Wang et al., 2023). Recently, Jhamtani et al. (2023) used Python to satisfy virtual assistant requests. Differently from that, our work provides an extensive study exploring the advantage of using code and domain descriptions in semantic parsing, across different datasets and PLs.

Code Prompting. Numerous works have shown that code-pretrained LLMs can be leveraged to improve various tasks such as arithmetic reasoning, commonsense reasoning, and others with prompts that involve code (Gao et al., 2022; Madaan et al., 2022; Chen et al., 2022; Zhang et al., 2023; Hsieh et al., 2023). In this work, we show for the first time how to effectively use code prompts for semantic parsing, demonstrating that when the output of the task is already programmatic and structured, performance gains can be dramatically high.

3 Setup

Given a natural language request x and an environment e, our task is to "satisfy" the request as follows by executing a program z: If x is an *information-seeking question*, program z should output the correct answer y; if x is an *action request*, z should update the environment e appropri-

```
Class Person:
  name: str
  def find_team_of() -> List[Person]: …
  def find_reports_of() -> List[Person]: …
  def find_manager_of() -> Person: …

Class Event:
  attendees: List[Person] = None
  subject: Optional[str] = None
  location: Optional[str] = None
  starts_at: Optional[List[DateTimeClause]] = None
  ...

class API:
  def find_person(name: str) -> Person: …
  def get_current_user() -> Person: …
  def add_event(event: Event) -> None: …
  ...
```

Figure 2: A partial example of a domain description containing the names of all objects and operators (in green) and type signatures (in orange).

ately. For example, an information-seeking question could be *"what is the longest river?"*, where e contains a list of facts about rivers and lengths, and the answer y should be returned based on these facts. An action request could be *"set a meeting with John at 10am"* where e contains a database with a list of all calendar items, and the task is to update e such that the requested meeting event is created. The environment e can be any type of database or system that provides a way to retrieve information or update it using *a formal language program*. Each environment accepts a different formal language for z, and has its own specific list of accepted operators (see Table 1 for examples of different formalisms used in this work).

We focus on the **true few-shot setup** where only a small ($\leq$ 10) set of demonstrations is used. Specifically, we assume knowledge of the formalism and operators supported by e, and a set of training examples $\{(x_i, z_i)\}_{i=1}^{k}$ where k is small ($\leq$ 10), and no other data, labeled or otherwise.

4 Domain-Augmented PL Prompts

Semantic parsing studies have traditionally used DSLs. We posit that using general-purpose PLs with a structured description of the domain in hand could better exploit the potential of modern LLMs, which are pretrained on a mix of code and natural language.

Leveraging Existing Coding Knowledge. While DSLs tailored to specific domains can be valuable for trained domain experts, their rarity makes it challenging for LLMs to learn them from just a few demonstrations. In contrast, PLs are prevalent in pretraining corpora; by prompting LLMs to generate PLs rather than DSLs, LLMs can leverage their existing coding knowledge without the need to learn the syntax and standard operations for a new language from scratch.

For instance, consider the operator `most` in Figure 1. LLMs with no prior knowledge of the given DSL struggle to correctly apply this operator without sufficient demonstrations. However, with Python, the model can exploit its parametric knowledge to perform this operation by employing the built-in `max` and `len` operators of Python, along with list comprehension. Another example is filtering sets of items in λ-DCS (Table 1, Overnight). Using a rare DSL, models must learn how to correctly use the filter operator from just a few demonstrations. However, LLMs have likely already seen a myriad of filtering examples during pretraining, e.g. in the form of Python's conditional list comprehension.

Domain Descriptions. While using PLs allows the model to leverage its parametric knowledge of the language's generic aspects, the LLM is still tasked with understanding domain-specific functionalities from a few in-context demonstrations. This is challenging, often even impossible, in a true few-shot setup, where the few fixed demonstrations may not cover all the functionality necessary to satisfy the test input request. A line of prior work alleviated this issue by selecting the most relevant demonstrations for every test input (Levy et al., 2023; Gupta et al., 2023), but this approach typically requires a large labeled pool of demonstrations.

To address this challenge in a true few-shot setup, we propose an intuitive solution that naturally aligns with the use of PLs: providing the model with a *Domain Description* (DD) outlining the available operators. Specifically, when using PLs, we prefix the ICL prompt with definitions of the domain classes, methods, attributes, and constants exactly as they are defined in the environment, with the implementations of specific methods concealed for prompt brevity (e.g., replaced with '`...`' in Python).

Figure 2 provides a snippet of the Python DD for SMCalFlow (Andreas et al., 2020), where users can create calendar events with certain people from their organization. Perhaps most importantly, DDs include the names of all available operators (highlighted in green in the figure). Without a list of available operators and relevant demonstrations, models are unlikely to generate a correct program.

The type signatures (highlighted in orange in the figure) provide additional important information on how these operators and attributes can be used. The complete DDs are deferred to App. E.

While DDs can also be used with DSLs, there's typically no consistent and formal way to write such descriptions. In contrast, DDs for PLs are not only easier to write, they could be particularly effective as pretraining corpora contain countless examples of how previously defined classes and methods are used later in the code. As we will empirically demonstrate in Section 6, DDs are indeed utilized more effectively with PLs than with DSLs.

Prompt Construction. The prompt that we use is a concatenation of the domain description (such as the example in Figure 2) and demonstrations (such as the inputs and MRs in Table 1) for a given environment. See App. F for the exact format.

5 Experimental Setup

5.1 Datasets and Environments

Datasets. We experiment with three semantic parsing datasets, covering both information-seeking questions and action requests. See Table 1 for examples.

- **GeoQuery** (Zelle and Mooney, 1996) contains user utterances querying about geographical facts such as locations of rivers and capital cities.
- **SMCalFlow** (Andreas et al., 2020) contains user requests to a virtual assistant helping with actions such as setting up organizational calendar events.
- **Overnight** (Wang et al., 2015) contains queries about various domains; in this work, we use the 'social network' domain, with questions about people's employment, education, and friends.

DSLs. Unless mentioned otherwise, we experiment with the original DSLs of the tasks: FunQL (Kate and Mooney, 2006) for GeoQuery, Dataflow for SMCalFlow, and λ-DCS (Liang et al., 2011) for Overnight. We also experiment with a simpler version of λ-DCS for Overnight, where we reversibly remove certain redundant keyword, and Dataflow-Simple (Meron, 2022), a simpler (and less expressive) version of Dataflow, to better understand the effect of the design of DSLs (§6.2).

Dataset Splits. For each dataset, we experiment with both i.i.d. splits (random splits between training and test sets) and compositional generalization splits, as detailed in App A.2. All results are reported on development sets where available, except for Tables 2 and 5, where we use the test sets.

Executable environments. As described in § 3, an environment is capable of executing a program z and either outputting an answer y (e.g., the name of a river) or modifying its own state (e.g., creating an event). In this work, for each dataset, we use an existing executable environment for the DSL formalism and implement one for Python.

To implement the Python environments, we analyze the original DSL programs to identify the requisite classes, their properties, and their methods, and then write Python code to create an executable environment. Importantly, whenever possible, we retain the original names of properties and constants used in the DSLs, ensuring that performance improvements can be attributed to the change in MR rather than changes in naming. We refer to App. A.1 for implementation details of all of the environments we use.

5.2 Evaluation

Metrics. The executable environments we have for all datasets, for both DSL and Python, allow us to compute *execution-based* accuracy. For GeoQuery and Overnight, we compare answers returned by generated programs to those generated by gold DSL programs. For SMCalFlow, we compare the state (i.e., calendar events) of the environments after executing gold and predicted programs. For DSL experiments, we additionally provide Exact Match metric results in App. B, which are computed by comparing the generated programs to gold-annotated programs.

We run all experiments with three seeds, each with a different sample of demonstrations, and report average accuracy. For each seed, the same set of demonstrations is used across different test instances, MRs, and prompt variations. Standard deviations for main results are provided in Appendix B.2.

Conversion to Python. To generate Python programs demonstrations, we convert a subset of the DSL programs of each dataset to Python using semi-automatic methods while validating them by ensuring they execute correctly. See App. C for details.

Models. We experiment with OpenAI's ChatGPT (`gpt-3.5-turbo-0613`) and the open-source Star-

Coder (Li et al., 2023a). Since GPT's maximum context length is longer, we conduct our experiments with GPT with $k = 10$ demonstrations and provide main results for StarCoder with $k = 5$. We use a temperature of 0 (greedy decoding) for generation.

Domain Descriptions for DSLs. For a thorough comparison, we also provide DDs for each DSL, containing similar information as the PL-based DDs (§4). We manually write these DDs based on the existing environments, listing all operators and describing type signatures. We write the descriptions of operators in natural language (NL); For GeoQuery we also experiment with code descriptions, where names of operators are followed by Python-like signatures (see App. E for all DDs). Unless mentioned otherwise, Full DD for DSLs refers to the NL version.

We note that providing DDs for DSLs is often not as straightforward as for PLs; we design the DSL-based DDs to be as informative as possible but do not explore different description design choices. This highlights another advantage of using PLs—their DDs can simply comprise extracted definitions of different objects without the need to describe the language itself.

6 Results

We first compare Python-based prompts with DSL-based prompts and the effect of DDs (§6.1). We then experiment with several other PLs and variations of DSLs to better understand how the design of the output language affects performance (§6.2).

6.1 Python vs DSLs

Baselines and Ablations. We compare multiple variations of DDs. *List of operators* simply lists all available operators without typing or function signatures (i.e., we keep only green text in Figure 2). *Full DD* contains the entire domain description, while *DD w/o typing* is the same as Full DD, except that it does not contain any type information (i.e., none of the orange text in Figure 2).

Main Results. Table 2 presents the results for ChatGPT (k=10), while Table 5 in App. B.1 shows results for Starcoder (k=5). We observe that Python programs without a DD outperform not only DSLs without a DD but even surpass DSLs prompted with a full DD across all splits for GPT and on most splits for Starcoder. Python with a full DD

		GeoQuery				SMC-CS		Overnight	
		i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
DSL	No DD	37.6	34.2	43.5	31.0	42.9	7.3	34.0	23.1
	List of operators	48.6	31.8	47.3	38.0	45.9	20.1	38.1	24.0
	Full DD (code)	51.9	34.5	53.6	43.9	-	-	-	-
	Full DD (NL)	61.0	41.5	52.5	39.4	40.7	20.7	38.8	23.2
Python	No DD	72.6	54.6	67.1	57.3	58.0	28.0	62.8	69.2
	List of operators	83.5	83.0	81.9	75.3	59.6	46.9	61.7	71.5
	DD w/o typing	82.1	83.1	83.4	75.7	69.1	65.6	62.1	68.5
	Full DD	**84.4**	**84.4**	**84.9**	**80.7**	**69.2**	**66.7**	**64.5**	**72.9**

Table 2: Execution accuracy of GPT-3.5-turbo, comparing Python-based prompts with DSL-based prompts across different DD variations, when 10 in-context demonstrations are used. Python-based prompts with Full DD consistently outperform DSL-based prompts by substantial amounts. Test sets results.

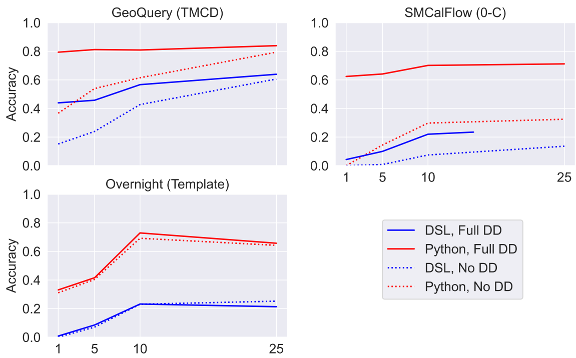

Figure 3: Execution accuracy for varying number of demonstrations. In almost all cases, Python outperforms DSL, both with a domain description and without, across different numbers of demonstrations (prompt for SMCalFlow, DSL, Full DD could not fit more than 15 examples given the model's context length limitation).

performs best in all 8 splits for GPT and on 5 splits for Starcoder. Notably, for ChatGPT, using Python with Full DD almost entirely eliminates the compositionality gap, i.e., the difference in performance between the i.i.d. split and compositional splits.

Ablating different parts of the DDs (rows "List of operators" and "DD w/o typing") reveals that in some cases, most of the performance gain for Python-based prompts is already achieved by adding the list of operators (e.g., GeoQuery i.i.d. split), while in other cases (e.g., GeoQuery length split) providing typing and signatures further improves accuracy. For DSL-based prompts, both formal DDs and natural language (rows Full DD formal/NL) underperform Python-based prompts, suggesting that Python's performance gains are not only due to descriptions being formal.

Prompt Length Trade-off. Figure 3 demonstrates that using Python consistently outperforms

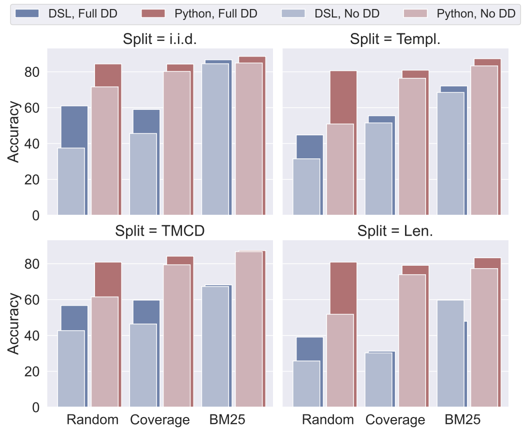

Figure 4: Python-based prompts, both with and without DD, consistently outperform DSL-based prompts, even with better demonstrations, for every split of GeoQuery.

DSLs across varying numbers (k) of demonstrations. For both GeoQuery and SMCalFlow, just a single demonstration with a DD outperforms 25 demonstrations without a DD. However, the impact of DDs depends on the dataset and the domain: DDs lead to dramatic gains for the more complex SMCalFlow, but are less impactful in Overnight where the domain is small.

Considering a real-world setup with constrained resources, where one might want to optimize performance given a maximum prompt length, we also investigate accuracy as a function of the *total number of prompt tokens* for three Python DD variations. We find that the optimal point in the trade-off between DD detail and number of demonstrations in the prompt varies per dataset (see Figure 5 in App. B.5). For Overnight, where the domain is simple, using demonstrations alone might suffice. However, for both GeoQuery and SMCalFlow, having the Full DD is preferred whenever it can fit.

Effect of Better Demonstrations Selection. Our results so far have demonstrated performance with a *random*, fixed set of demonstrations, in line with our goal of minimizing labeling workload. However, in some scenarios, the budget may allow access to larger pools of demonstrations, in turn allowing more sophisticated demonstration selection methods to be applied. To evaluate our approach in such a setting, we additionally experiment with two selection methods.

The first method optimizes for *operator coverage* (Levy et al., 2023; Gupta et al., 2023) by selecting a fixed set of demonstrations that cover as many of the operators as possible. This is achieved by greedily and iteratively selecting demonstrations to cover operators (see App. D for details). This fixed set covers 68% to 81% of the operators with $k = 10$ (coverage varies across splits). Our second selection method is similarity-based retrieval: given a test example utterance, we retrieve the training examples with the most similar utterances using BM25 (Robertson and Zaragoza, 2009).

We present the results for the different demonstration selection methods in Figure 4 for GeoQuery, for which we have annotated the entire training set with Python programs. We observe that for every selection method, both with and without DD, Python-based prompts consistently outperform DSL-based prompts.

Error Analysis. We now analyze the kinds of errors made by the LLM when prompted with Python and a DD. For SMCalFlow and ChatGPT, the development set of the compositional split (of size 250) resulted in 78 errors on one of the seeds; we include common examples of errors in App. B.3 (Table 7, with examples of correct predictions in Table 8). 42 (54%) of the errors were because the program failed to execute. The remaining 36 were due to incorrect execution. Closer analysis revealed most of these errors to be due to failure to understand the input utterance or not using the API correctly. A small fraction (11, 14%) of the error instances were found to be unsupported by the original environment or our Python re-implementation. For GeoQuery, on the other hand, among the 18 errors made by ChatGPT on the development set of the TMCD split (of size 100) on one of the seeds, only 8 were attributed to model errors, while 8 were due to discrepancies in the dataset[3] and 2 resulted from environment limitations.

The above analysis suggests that while using PLs and DDs greatly improves the performance of LLMs, there is still scope for improvement in more complex domains (like SMCalFlow). Future work can explore how to ensure LLMs remain faithful to the DD and how to design PL environments to be more amenable to LLMs.

6.2 What Makes a Good MR?

Building on the findings from Section 6.1, showing that Python prompts consistently outperform DSLs,

[3]These include incorrect FunQL annotation or discrepancies in the GeoBase database underlying GeoQuery, e.g., *Mount McKinley* is referred to as "Mount McKinley" and "mckinley" leading to ambiguity.

		GeoQuery				SMC-CS		Overnight	
		i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
DSL	No DD	38.7	32.1	42.7	25.8	42.9	7.5	34.0	23.1
	Full DD	61.9	44.8	56.7	39.1	40.8	22.0	38.8	23.2
Python	No DD	71.6	50.9	61.5	51.8	58.1	29.7	62.8	69.2
	Full DD	83.1	80.6	80.9	80.9	**69.3**	69.9	64.5	**72.9**
Javascript	No DD	80.0	72.1	75.2	73.6	64.1	46.1	61.6	51.3
	Full DD	81.1	80.0	77.3	73.6	68.1	71.6	61.1	50.1
Scala	No DD	82.5	73.3	73.9	68.5	62.3	45.9	69.7	65.1
	Full DD	**83.5**	**83.3**	**82.4**	**82.1**	69.2	**72.4**	**69.9**	61.9
SQL	No DD	65.7	56.6	61.8	45.6				
	Full DD†	73.1	68.7	75.7	62.7				
Dataflow-Simple	No DD					50.9	22.7		
	Full DD					59.7	63.9		
λ-DCS Simple	No DD							37.3	27.5
	Full DD							38.0	31.7

Table 3: Development set execution accuracies for Python, Javascript, Scala and SQL comprising 2.5%, 19.6%, 0.1% and 4.9% of Stack (Kocetkov et al., 2023), respectively, along with two DSL variations. There is no clear winner among the various PLs, suggesting that the prevalence in pretraining corpora is not a good predictor of performance. † For SQL we use the schema definition, see App. E.

we now investigate the source of these performance gains. Specifically, in this section, we ask:

1. Is the performance gain of a PL linked to its prevalence in pretraining corpora?
2. Can rare DSLs be simplified in a way that enables them to perform as well as PLs?
3. Does the ability to break down programs into intermediate steps contribute to the improved performance of PLs?

6.2.1 Effect of a PL's Prevalence

To answer the first question, we extend our experiments to include Scala and Javascript. For GeoQuery, which requires querying a database, we additionally experiment with SQL, a common query language.

According to the PL distribution provided by the Stack (Kocetkov et al., 2023), a large corpus of GitHub code, Scala is far less common than Python (0.1% vs 2.5%), while Javascript and SQL are more popular (19.6% and 4.9%).

Evaluation Procedure for Additional PLs. We evaluate the performance of these additional PLs by prompting LLMs similarly to how we evaluated Python prompts (§6.1). However, to avoid the undue engineering effort of implementing a complete executable environment for Scala and Javascript, as we did for Python, we evaluate generated programs by first automatically converting them to Python during inference time, similar to previous work (Cassano et al., 2023), while confirming that the conversion is *faithful* and does not introduce bias. For SQL, we use the original dataset queries[4] and use the schema definition instead of a Domain Description. We provide the complete procedure, prompts and analysis in App. C.2.

Results. Table 3 demonstrates that all three PLs outperform DSL-based prompts. However, the performance of the three PLs varies across datasets and splits, with Scala performing best in most splits, and SQL performing worst. This suggests that the prevalence of a PL in pretraining corpora alone does not reliably predict performance in semantic parsing tasks. This finding offers a subtle counterpoint to the results of Cassano et al. (2023), who identified a correlation between the prevalence of a PL in pretraining data and performance on other programming benchmarks.

6.2.2 Simplifying PLs

If the prevalence of PLs in pretraining corpora doesn't correlate with performance, could it be that DSL-based prompts perform worse because DSLs are overly complex, and simplifying them could improve performance (Herzig et al., 2021; Li et al., 2022)?

To investigate this, we experiment with simplified versions of SMCalFlow and Overnight's DSLs. Specifically, we use Dataflow-Simple (Meron, 2022), a version of Dataflow tailored for creating events and querying organizational charts, which uses fewer operators and an entirely different syntax, with function calls in the style of popular PLs. While Dataflow-Simple isn't equivalent to Dataflow, it can be used to satisfy all of the requests in SMCalFlow's dataset. For Overnight, we create a simplified version of λ-DCS, where we remove redundant operators in the context of the evaluation setup, reducing its length by 42% on average. Specifically, we remove the `call` operator, typing (`string`, `date`, `number`), redundant parentheses and the namespace `SW`. Examples for both MRs are provided in Table 1.

The results presented in the bottom two sections of Table 3 reveal that the surface-level simplification of λ-DCS provides only a marginal boost to performance. On the other hand, Dataflow-simple

[4]Taken from https://github.com/jkkummerfeld/text2sql-data

	GeoQuery				SMCalFlow-CS		Overnight	
	i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
Single	83.2	**80.9**	**80.3**	79.1	64.8	60.4	62.9	57.6
Multi.	**83.7**	80.6	80.0	**80.3**	**67.7**	**70.0**	**64.5**	**72.9**

Table 4: Accuracy of single-line programs against multiple-line programs with intermediate steps, in the Full DD setup. Breaking down code into intermediate steps usually contributes to performance, yet single line demonstrations still outperform DSL-based prompts.

surprisingly performs nearly as well as the other PLs. These findings suggest that designing DSLs to resemble PLs could also be effective (when DD is included), even when DSLs are rare in pretraining corpora. However, what unique elements of PLs should be adopted in DSLs to yield comparable performance gains remains an open question.

6.2.3 Effect of Intermediate Steps

A key distinction between the PLs and the DSLs evaluated in this work lies in the fact that PLs allow breaking down the programs into multiple steps and assigning intermediate results to variables. To measure the impact of this aspect, we modify PL programs such that if a program contains more than one line, we compress it into a single line, eliminating intermediate variables and vice versa. We employ GPT-4 to perform these modifications and use execution-based evaluation to ensure that the program meaning does not change (see App. C.3 for the exact prompt). We note that the only modification made is to the programs of the prompt demonstrations, however, models can still output a program of any line length.

Results presented in Table 4 suggest that breaking down code into intermediate steps indeed contributes to higher performance in most cases. However, even single line demonstrations still significantly outperform DSL-based prompts.

7 Conclusions

In this work, we have shown that leveraging PLs and DDs does not only improve the effectiveness of in-context learning for semantic parsing, leading to substantial accuracy improvements across various datasets, but also significantly narrows the performance gap between i.i.d. and compositional splits and reduces the need for large demonstration pools. Our findings carry significant implications for the development of semantic parsing applications using modern LLMs.

Limitations

We evaluate models and methods using executable environments that we have implemented in Python; however, these implementations might not always accurately replicate the original environment. Particularly in SMCalFlow, which includes many long-tail operators infrequently used in the dataset, we omit some operators in our implementation.

We use OpenAI's API to annotate most of the Python programs that are used as demonstrations. While we validate the correctness of all programs, it is possible that this method introduces some bias into the nature of the generated programs.

We present the prevalence of different PLs in the Stack, assuming it offers a rough estimate of these languages' popularity on the web. However, the actual prevalence of PLs specifically within the training data of OpenAI's models, employed in this work, remains unknown. Further, while our experiments with simplified versions of DSLs and rare PLs suggest that the improved performance of LLMs with PLs in compositional settings is not merely due to surface-level memorization, how much of these can be attributed to LLMs' ability to generalize compositionally versus memorization from pre-training corpus remains an important open question.

Finally, while this study focused on semantic parsing, the idea of using a PL for output representation and for specifying background information and task structure (as in DDs) could be applicable to any other generative tasks where the output must conform to some structure and has a step-wise nature (e.g., recipes, travel itineraries). We leave it to future work to explore these settings.

References

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. 2021. Learning to recombine and resample data for compositional generalization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Shengnan An, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Jian-Guang Lou, and Dongmei Zhang. 2023. How do in-context examples affect compositional generalization? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11027–11052, Toronto, Canada. Association for Computational Linguistics.

Jacob Andreas. 2020. Good-enough compositional data augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7556–7566, Online. Association for Computational Linguistics.

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.

Cem Anil, Yuhuai Wu, Anders Johan Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Venkatesh Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. 2022. Exploring length generalization in large language models. In *Advances in Neural Information Processing Systems*.

Ben Bogin, Shivanshu Gupta, and Jonathan Berant. 2022. Unobserved local structures make compositional generalization hard. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2731–2747, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Ben Bogin, Sanjay Subramanian, Matt Gardner, and Jonathan Berant. 2021. Latent compositional representations improve systematic generalization in grounded question answering. *Transactions of the Association for Computational Linguistics*, 9:195–210.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2023. Knowledge transfer from high-resource to low-resource programming languages for code llms.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *ArXiv preprint*, abs/2211.12588.

Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. 2023. Compositional semantic parsing with large language models. In *The Eleventh International Conference on Learning Representations*.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. Pal: Program-aided language models. *ArXiv preprint*, abs/2211.10435.

Shivanshu Gupta, Matt Gardner, and Sameer Singh. 2023. Coverage-based example selection for in-context learning.

Shivanshu Gupta, Clemens Rosenbaum, and Ethan R. Elenberg. 2024. Gistscore: Learning better representations for in-context example selection with gist bottlenecks.

Shivanshu Gupta, Sameer Singh, and Matt Gardner. 2022. Structurally diverse sampling for sample-efficient training and comprehensive evaluation. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 4966–4979, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Jonathan Herzig and Jonathan Berant. 2018. Decoupling structure and lexicon for zero-shot semantic parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1619–1629, Brussels, Belgium. Association for Computational Linguistics.

Jonathan Herzig and Jonathan Berant. 2021. Span-based semantic parsing for compositional generalization. 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 908–921, Online. Association for Computational Linguistics.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. 2021. Unlocking compositional generalization in pre-trained models using intermediate representations. *ArXiv preprint*, abs/2104.07478.

Arian Hosseini, Ankit Vani, Dzmitry Bahdanau, Alessandro Sordoni, and Aaron Courville. 2022. On the compositional generalization gap of in-context learning. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 272–280, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models.

Harsh Jhamtani, Hao Fang, Patrick Xia, Eran Levy, Jacob Andreas, and Ben Van Durme. 2023. Natural language decomposition and interpretation of complex utterances.

Rohit J. Kate and Raymond J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 913–920, Sydney, Australia. Association for Computational Linguistics.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *AAAI Conference on Artificial Intelligence*.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. 2020. Measuring compositional generalization: A comprehensive method on realistic data. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.

Itay Levy, Ben Bogin, and Jonathan Berant. 2023. Diverse demonstrations improve in-context compositional generalization. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1401–1422, Toronto, Canada. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023a. Starcoder: may the source be with you!

Xiaonan Li, Kai Lv, Hang Yan, Tianyang Lin, Wei Zhu, Yuan Ni, Guotong Xie, Xiaoling Wang, and Xipeng Qiu. 2023b. Unified demonstration retriever for in-context learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4644–4668, Toronto, Canada. Association for Computational Linguistics.

Zhenwen Li, Jiaqi Guo, Qian Liu, Jian-Guang Lou, and Tao Xie. 2022. Exploring the secrets behind the learning difficulty of meaning representations for semantic parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3616–3625, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599, Portland, Oregon, USA. Association for Computational Linguistics.

Matthias Lindemann, Alexander Koller, and Ivan Titov. 2023. Compositional generalization without trees using multiset tagging and latent permutations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14488–14506, Toronto, Canada. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Joram Meron. 2022. Simplifying semantic annotations of SMCalFlow. In *Proceedings of the 18th Joint ACL - ISO Workshop on Interoperable Semantic Annotation within LREC2022*, pages 81–85, Marseille, France. European Language Resources Association.

Inbar Oren, Jonathan Herzig, and Jonathan Berant. 2021. Finding needles in a haystack: Sampling structurally-diverse training sets from synthetic data for compositional generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10793–10809, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Pawel Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. 2022a. Improving compositional generalization with latent structure and data augmentation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4341–4362, Seattle, United States. Association for Computational Linguistics.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Tianze Shi, Jonathan Herzig, Emily Pitler, Fei Sha, and Kristina Toutanova. 2022b. Evaluating the impact of model scale for compositional generalization in semantic parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9157–9179, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? 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 922–938, Online. Association for Computational Linguistics.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7699–7715, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Hongjin SU, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Selective annotation makes language models better few-shot learners. In *The Eleventh International Conference on Learning Representations*.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Zhengxuan Wu, Christopher D. Manning, and Christopher Potts. 2023. ReCOGS: How incidental details of a logical form overshadow an evaluation of semantic interpretation. *ArXiv preprint*, abs/2303.13716.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. *ArXiv preprint*, abs/2202.13169.

Silei Xu, Sina Semnani, Giovanni Campagna, and Monica Lam. 2020. AutoQA: From databases to QA semantic parsers with only synthetic training data. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 422–434, Online. Association for Computational Linguistics.

Yuekun Yao and Alexander Koller. 2022. Structural generalization is hard for sequence-to-sequence models.

In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5048–5062, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Jiacheng Ye, Zhiyong Wu, Jiangtao Feng, Tao Yu, and Lingpeng Kong. 2023. Compositional exemplars for in-context learning.

Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. 2021. Compositional generalization for neural semantic parsing via span-level supervised attention. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2810–2823, Online. Association for Computational Linguistics.

Pengcheng Yin, John Wieting, Avirup Sil, and Graham Neubig. 2022. On the ingredients of an effective zero-shot semantic parser. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1455–1474, Dublin, Ireland. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI, Vol. 2*.

Li Zhang, Liam Dugan, Hainiu Xu, and Chris Callison-burch. 2023. Exploring the curious case of code prompts. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 9–17, Toronto, Canada. Association for Computational Linguistics.

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active example selection for in-context learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9134–9148, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12697–12706. PMLR.

A Datasets

A.1 Executable Environments

We describe the executable environments we use separately for each dataset and formalism.

A.1.1 Geoquery

FunQL To execute the FunQL queries, we use the GeoQuery[5] system, a prolog-based implementation that we execute using SWI-Prolog[6].

Python We manually write a Python environment that is functionally equivalent to the GeoQuery system. The environment includes two components: a class for parsing and loading the Geobase database and an API for executing queries against this database. We show the API in Figures 10 and 11.

SQL We use the SQLite engine to run SQL queries, with the data and schema provided in `https://github.com/jkkummerfeld/text2sql-data`.

Evaluation Running queries with the GeoQuery system using FunQL, SQL and Python programs results in either a numeric result of a set of entities. We evaluate FunQL and Python programs by comparing their denotation against the gold denotation obtained by executing the gold FunQL program for each query, with no importance to order, and similarly evaluate SQL programs by comparing their denotation of gold SQL programs.

A.1.2 SMCalFlow

Dataflow and Dataflow-Simple We use the software provided by Meron (2022)[7] to execute Dataflow-Simple. Dataflow programs are executed by 'simplifying' them, i.e. converting them to Dataflow-Simple, using the code provided in that package. The environment holds a database with people, the relationship between them in the organization, and a list of events.

Python We run Python programs by automatically converting them to Dataflow-Simple in a determinstic method, then executing them as mentioned above. Conversion is done by implementing each of the python classes and operators with a method that returns an AST that represents a relevant Dataflow-Simple sub-tree. For example, the Python method `find_manager_of('person')` returns the corresponding AST of Dataflow-Simple's method, `FindManager('person')`.

Evaluation All of the test instances in the splits we work with are requests to create events. Thus, to evaluate programs, we compare if the events created after running a generated program is exactly the same as the event create after running the gold Dataflow program. Since programs are executed using a database, which is used, for example, to find people by their names, we populate the database with a short list of people with random names. During evaluation, we extract names of people from both generated and gold programs, and arbitrarily map and replace each name in the programs to one of the people in the database. We do this for both generated and gold programs, while making sure that mapping is consistent in both of them during an evaluation for a single example.

We ignore the generated subject of the meeting, as we found that there are many inconsistencies in the way subjects were annotated: underspecified requests such as *Set up a meeting with John* are often be annotated inconsistently, having either no subject, the subject "meeting", or something else.

A.1.3 Overnight

λ-DCS and λ-DCS-Simple To execute λ-DCS programs, we use Sempre.[8] Specifically, we use the executable Java program provided by Herzig and Berant (2018).[9]

Python To create the Python environment, we first use Sempre to output all entities in the 'social-network' domain. We implement the python environment to be executed over these loaded entities.

Evaluation Running the programs returns a list of entities. For all formalisms, we consider accuracy to be correct iff the list of entities is exactly the same as the list of entities returned by running the gold λ-DCS program.

A.2 Splits

For GeoQuery, we use the splits provided by Shaw et al. (2021), comprising the original i.i.d. split and

[5] `https://www.cs.utexas.edu/users/ml/nldata/geoquery.html`

[6] `https://www.swi-prolog.org/`

[7] `https://github.com/telepathylabsai/OpenDF`

[8] `https://github.com/percyliang/sempre`

[9] `https://github.com/jonathanherzig/zero-shot-semantic-parsing/blob/master/evaluator/evaluator.jar`

the compositional generalization splits (Template, TMCD and length).

For SMCalFlow, we use the i.i.d. and compositional splits proposed by Yin et al. (2021). These compositional splits evaluate predictions for queries that combine two domains: event creation and organizational chart. Specifically, we use the hardest "0-C" split, where the training set contains examples only from each of the domains separately, with no single example that combines both domains. For experiments with 5 or more demonstrations, we make sure there are at least two demonstrations from each of the domains.

For Overnight, we take the i.i.d. split and a compositional split (specifically `template/split_0`, selected arbitrarily) from those published in Bogin et al. (2022).

We used the development sets for each of the datasets only to make sure predicted programs were executed as expected. For Overnight, where such a set was unavailable, we used 50 examples from the training set.

For GeoQuery, we use the entire tests sets (of size ranging from 279 to 331), while for SMCalFlow and Overnight, we sample 250 examples from the test sets.

We sample in-context demonstrations from the pool of training examples for which we have Python annotations. For GeoQuery, we have 824 such annotated programs, for SMCalFlow 128 and for Overnight 60.

B Additional Results

B.1 Starcoder

Main results for Starcoder are presented in Table 5. With $k = 5$ Starcoder's performance is generally lower than ChatGPT's, however the main trends remain the same: Python-based prompts with Full DD outperform DSL-based prompts in all cases, and Python-based prompts with Full DD outperform No DD in all cases but one.

B.2 Standard Deviations

All reported accuracy figures are average values obtained from three different seeds. The standard deviations corresponding to Table 2 are detailed in Table 6.

B.3 Prediction Examples

Examples for failed predictions are presented in table 7, and for correct predictions in table 8.

B.4 Exact Match Accuracy

We provide results for all DSL experiments with exact match as the metric for reference in Table 9. Note that for Geoquery, while Full DD leads to significant improvements in execution accuracy (Table 2), when measuring exact match we see less of an improvement (e.g. 37.6 to 61.0 vs 20.7 to 27.6 in the i.i.d. split). We find that this is due to correct but different usage of the DSL, e.g. the model generates `answer(count(traverse_2(stateid('colorado'))))`, which is different from the gold program `answer(count(river(loc_2(stateid('colorado'))))).`

B.5 Accuracy vs # of Tokens

We present execution-based accuracy against the number of prompt tokens in Figure 5 for three Python DD variations.

C Program Annotations

C.1 Python

To create the pool of python programs for our experiment, we start by manually convert 2-10 examples to Python programs to seed our pool of Python-annotated instances. We then iteratively sample demonstrations from the pool and prompt an LLM with the Python DD (§4) to automatically annotate the rest of the examples (we use either OpenAI's gpt-3.5-turbo or gpt-4[2]). Only predictions that are evaluated to be correct, using the same execution-based evaluation described above, are added to the pool (see App. C for further details).

We use the prompt in Figure 6 with Python DD to generate Python programs.

C.2 Scala and Javascript

We use the prompt in Figure 6 with the Scala or Javascript DD to generate programs for the corresponding language. To further convert to Python for execution-based evaluation, we use the prompt in Figure 7. Tables 10 and 11 contain example conversions from Javascript and Scala respectively to Python for GeoQuery.

To confirm that the conversion is *faithful* and does not introduce bias, such as fixing incorrect programs or breaking correct ones, we manually analyzed 100 random examples of the converted Python programs, 50 each from Javascript and Scala, finding only 1 instance each of an unfaithful conversion.

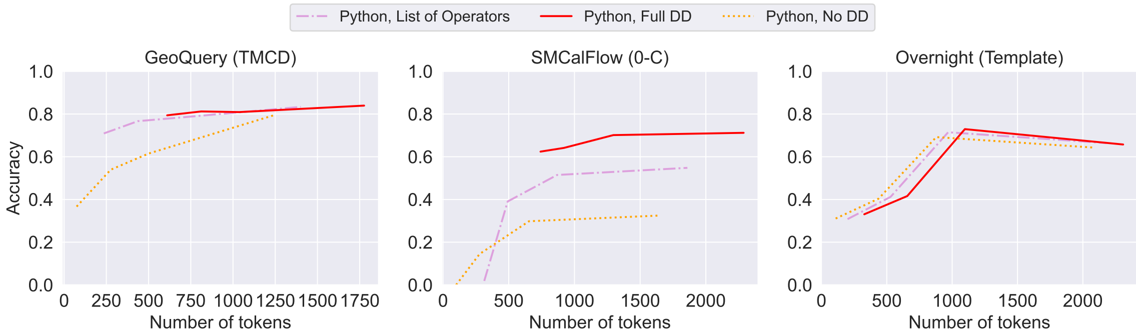

Figure 5: Execution accuracy for varying number of demonstrations, presenting the same data as Figure 3 but visualizes it against the number of prompt tokens. The effect of DDs greatly varies between the datasets. For both GeoQuery and SMCalFlow, having the Full DD is preferred whenever it can fit.

```
1 Given the following data structures and functions:
2 [DD]
3
4 Write code to solve the following queries:
5
6 query: [query-1]
7 solution: [solution-1]
8 ...
9 query: [query-test]
```

Figure 6: The prompt template we use. [DD] is replaced with the domain description for the environment being used, [query-i] and [solution-i] are replaced with utterance/output demonstrations, and [query-test] is replaced with the test utterance. Lines 1-3 are only included in experiments that contain DD.

````
Given the following python data structures and functions:

[Python DD]

and the corresponding javascript data structures and functions:

[Javascript DD]

convert the following javascript functions to python:

Javascript:
```javascript
[javascript-code-1]
```

Python:
```python
[python-code-1]
```
...

Javascript:
```javascript
[query-javascript-code]
```
Python:
```python
````

Figure 7: The prompt template we use to convert non-Python programs (Javascript in this case) to Python for evaluation. [Python DD] and [Javascript DD] are replaced with the corresponding domain descriptions, [javascript-code-i] and [python-code-i] with demonstrations of javascript to python conversion, and [query-javascript-code] is replaced with test Javascript code to be converted.
````

		GeoQuery				SMCalFlow-CS		Overnight	
		i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
DSL	No DD	24.4	19.0	28.0	14.7	23.3	0.7	18.1	7.7
	List of operators	39.8	27.2	36.2	32.3	18.4	0.3	23.2	9.9
	Full DD	46.9	45.2	45.2	38.9	22.7	2.0	22.4	12.0
Python	No DD	56.1	42.2	50.2	32.4	22.8	5.1	51.9	**39.9**
	List of operators	**73.3**	**70.1**	70.7	61.7	22.4	13.2	55.5	38.1
	DD w/o typing	73.0	70.0	69.7	62.3	35.1	25.2	56.1	36.8
	Full DD	73.2	69.7	**75.2**	**68.6**	**43.7**	**33.2**	**56.9**	36.9

Table 5: Execution accuracy of Starcoder, comparing Python-based prompts with DSL-based prompts, across different DD variations, with 5 in-context demonstrations. Similarly to ChatGPT (Table 2), Starcoder used with Python-based prompts with Full DD is consistently better than with DSL-based prompts. Test sets results.

		GeoQuery				SMCalFlow-CS		Overnight	
		i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
DSL	No DD	4.3	11.4	1.4	4.6	6.9	8.8	3.8	2.0
	List of operators	3.2	10.1	0.3	0.9	2.1	9.0	9.9	6.1
	Full DD (formal)	6.1	8.7	5.5	4.7	-	-	-	-
	Full DD (NL)	3.9	5.4	2.4	5.3	6.2	6.9	5.8	7.1
Python	No DD	13.0	11.6	5.6	3.8	4.5	8.7	2.8	0.7
	List of operators	2.2	0.9	2.5	1.9	5.0	10.7	9.3	10.2
	DD w/o typing	2.5	2.8	1.7	2.0	3.7	9.7	2.9	4.6
	Full DD	0.9	1.2	0.5	1.5	3.4	10.9	1.1	4.5

Table 6: Standard deviations for ChatGPT's accuracy in Table 2.

C.3 Single/Multi Line Conversion

As described in §6.2.3, we convert single-line programs to multiple lines programs with intermediate steps and vice-versa, using GPT-4. We make sure conversions are correct by validating the execution-based accuracy of converted programs; if programs are invalid, we regenerate programs with a a temperature of 0.4 until a correct solution is found. We use GPT-4 with the prompts provided in Fig. 8 and Fig. 9.

The following is an example for a conversion of a multi-line program given the utterance *"Which states have points higher than the highest point in Colorado?"*. The original annotation:

```
def answer():
    colorado_state = geo_model.find_state("colorado")
    highest_point_in_colorado = colorado_state.high_point.elevation
    states_with_higher_points = [s for s in geo_model.states if s.high_point.elevation > highest_point_in_colorado]
    return states_with_higher_points
```

The converted annotation:

```
def answer():
    return [s for s in geo_model.states if s.high_point.elevation > geo_model.find_state("colorado").high_point.elevation]
```

D Demonstration Selection Methods

We experiment with two demonstration selection methods.

Operator Coverage This method selects a single fixed set of demonstrations with maximal coverage of operators that are used for every test input. For this, we use a slightly modified version of the greedy set coverage algorithm of Gupta et al. (2023), shown in Algorithm 1. Here, the set of structures $\mathcal{S}$ is the set of all unigram operators in given formalism, and the measure of set-coverage is defined as $\texttt{setcov}(\mathcal{S}, Z) = \sum_{s\in\mathcal{S}} \max_{z\in Z} \mathbb{1}[s \in S_z]$ where S_z is the set of operators in the candidate demonstration z.

BM25 We use BM25 to retrieve the most similar instances and use as demonstrations for each test input. We use the `rank_bm25`[10] package's implementation of the Okapi variant of BM25.

[10] https://github.com/dorianbrown/rank_bm25

Input	Predicted Python Program	Error

Put lunch with Jane 's boss in her office down for next Wednesday at noon .	`def answer():`
	`    jane = api.find_person("Jane")`
	`    boss = jane.find_manager_of()`
	`    api.add_event(Event(subject="Lunch with Jane's Boss", attendees=[jane, boss], starts_at=[DateTimeValues.NextWeek, DateTimeClause.time_by_hm(hour=12, am_or_pm="pm")], location="Jane's Office"))`
		Execution error due to incorrect API use: `NextWeek` is not a valid `DateTimeValue`.

i need to have all of jake 's team except jennifer at my staff meeting this friday	`def answer():`
	`    jake = api.find_person("Jake")`
	`    team = jake.find_team_of()`
	`    team.remove(api.find_person("Jennifer"))`
	`    api.add_event(Event(subject="Staff Meeting", attendees=team, starts_at=[DateTimeClause.get_next_dow(day_of_week="Friday")]))`
		Execution error due to incorrect API use: The right way to exclude people is to use `attendees_to_avoid` argument of the `Event` class.

I am going to be with Abby and her Skip tomorrow	`def answer():`
	`    abby = api.find_person("Abby")`
	`    skip = abby.find_manager_of()`
	`    api.add_event(Event(attendees=[abby, skip], starts_at=[DateTimeValues.Tomorrow]))`
		Incorrect execution as the model did not understand *skip* means manager's manager.

create drinks with boss on Tuesday	`def answer():`
	`    boss = api.find_person("boss")`
	`    api.add_event(Event(subject="Drinks with Boss", starts_at=[DateTimeClause.get_next_dow(day_of_week="Tuesday")], attendees=[boss]))`
		Incorrect execution as the model did not use `find_manager_of` to find the manager.

Table 7: Examples of common errors made by GPT-Turbo-3.5 on the compositional 0-C split of SMCalFlow. Majority of errors were due to the LLM failing to understand the input utterance or incorrectly using the API.

````
Given the following request and python program:

request: [query]

```python
[program]
```
We want to decompose the program into multiple lines as much as is reasonable possible.

```python
````

Figure 8: Prompt used to convert single-line programs to multiple-line programs.

Algorithm 1 Greedy Optimization of Set Coverage

Require: Instance pool $\mathcal{T}$; Set of structures $\mathcal{S}$; desired number of demonstrations k; coverage scoring function `setcov`

1: $Z \leftarrow \emptyset$ ▷ Selected Demonstrations
2: $Z_{curr} \leftarrow \emptyset$ ▷ Current Set Cover
3: `curr_cov` $\leftarrow -$inf
4: **while** $|Z| < k$ **do**
5: $\quad z^*$, `next_cov` $= \arg\max_{z \in \mathcal{T}-Z}$ `setcov` $(\mathcal{S}, Z_{curr} \cup z)$
6: $\quad$ **if** `next_cov` $>$ `curr_cov` **then** ▷ Pick z^*
7: $\quad\quad$ `curr_cov` $\leftarrow$ `next_cov`
8: $\quad\quad Z \leftarrow Z \cup z^*$
9: $\quad\quad Z_{curr} \leftarrow Z_{curr} \cup z^*$
10: $\quad$ **else** ▷ Or start new cover
11: $\quad\quad Z_{curr} \leftarrow \emptyset$, `curr_cov` $\leftarrow -$inf
12: $\quad$ **end if**
13: **end while**
14: **return** Z

E Domain Descriptions

We provide the domain descriptions that we use for each environment in the following figures:

- Geoquery: Python (10, 11), FunQL (NL: 12, formal: 13), Javascript (14, 15), Scala (16). For SQL, instead of domain descriptions, we use the schema definition taken verbatim from https://raw.githubusercontent.com/jkkummerfeld/text2sql-data/master/data/geography-schema.csv.
- SMCalFlow: Python (17), Dataflow (18, 19), Dataflow-Simple (20, 21), Javascript (22, 23), Scala (24).
- Overnight: Python (25) λ-DCS (26, 27), λ-
````

<table>
<tr><th>Dataset</th><th>Input</th><th>Predicted Python Program</th></tr>
<tr><td>GeoQuery</td><td>Which states border states through which the Mississippi traverses ?</td><td><pre>def answer() -> List[State]:
    mississippi_river = geo_model.find_river("mississippi")
    bordering_states = set()
    for state in mississippi_river.traverses:
        bordering_states.update(state.next_to)
    return list(bordering_states)</pre></td></tr>
<tr><td>SMCalFlow</td><td>Schedule a meeting with my team on November 3rd at 11 am as well .</td><td><pre>def answer():
    team = api.get_current_user().find_team_of()
    api.add_event(Event(subject="Meeting with Team", starts_at=[DateTimeClause.date_by_mdy(month=11, day=3), DateTimeClause.time_by_hm(hour=11, am_or_pm="am")], attendees=team))</pre></td></tr>
<tr><td>Overnight</td><td>student whose start date is end date of employee alice</td><td><pre>def answer():
    alice = api.find_person_by_id("en.person.alice")
    students_with_same_start_date = [person for person in api.people if person.education and any(e.start_date == alice_employment.end_date for e in person.education for alice_employment in alice.employment)]
    return students_with_same_start_date</pre></td></tr>
</table>

Table 8: Examples of correct Python predictions made by GPT-Turbo-3.5 on the compositional TMCD split of GeoQuery, 0-C split of SMCalFlow, and Template split of Overnight.

		GeoQuery				**SMCalFlow-CS**		**Overnight**	
		i.i.d.	Templ.	TMCD	Len.	i.i.d.	0-C	i.i.d.	Templ.
DSL	No DD	20.7	16.5	26.1	14.8	16.7	3.1	26.4	0.3
	List of operators	28.7	13.5	29.4	18.3	17.3	4.3	29.2	0.5
	Full DD	27.6	17.8	31.0	16.1	15.7	4.5	27.3	0.3

Table 9: Exact match accuracy of GPT-3.5-turbo for DSL-based prompts. Test set results.

DCS Simple (28), Javascript (29), Scala (30).

F Prompt Construction

We provide the prompt template that we use in Fig. 6.

```
Given the following request and python program:

request: [query]

```python
[program]
```
We want to make the python program a single line program that returns the same output.
If a single line program is not possible, use a minimal number of lines.

```python
```

Figure 9: Prompt used to convert multiple-line programs to single-line programs.

Input	Directly Annotated Python Program	Javascript Program	Converted Python Program

In which state does the highest point in USA exist ?	`def answer() -> State:`
	`    highest_point = max(geo_model.places, key=lambda x: x.elevation)`
	`    return highest_point.state`
		`function answer() {`
		`    let highest_point_state = geo_model.states.reduce((maxState, currentState) => {`
		`        return currentState.high_point > maxState.high_point ? currentState : maxState;`
		`    });`
		`    return highest_point_state.name;`
		`}`
			`def answer() -> str:`
			`    highest_point_state = max(geo_model.states, key=lambda x: x.high_point.elevation)`
			`    return highest_point_state.name`

What rivers flow through the largest state ?	`def answer() -> List[River]:`
	`    largest_state = max(geo_model.states, key=lambda x: x.size)`
	`    return largest_state.rivers`
		`function answer() {`
		`    let largest_state = geo_model.states.reduce((maxState, currentState) => {`
		`        return currentState.area > maxState.area ? currentState : maxState;`
		`    });`
		`    return largest_state.rivers;`
		`}`
			`def answer() -> List[River]:`
			`    largest_state = max(geo_model.states, key=lambda x: x.area)`
			`    return largest_state.rivers`

What is the highest point in the state with the capital Des Moines ?	`def answer() -> Place:`
	`    state = [s for s in geo_model.states if s.capital and s.capital.name == "des moines"][0]`
	`    return state.high_point`
		`function answer() {`
		`    let des_moines_state = geo_model.find_city("des moines").state;`
		`    return des_moines_state.high_point;`
		`}`
			`def answer() -> Place:`
			`    des_moines_state = geo_model.find_city("des moines").state`
			`    return des_moines_state.high_point`

Table 10: Example conversions of predicted Javascript programs for GeoQuery to Python along with original (direct) Python annotation. Converted programs can differ from the directly annotated Python programs as the conversions were performed without the original query and hence tended to be faithful to given Javascript programs.
```

Input	Directly Annotated Python Program	Scala Program	Converted Python Program

Which rivers run through states bordering New Mexico ?	`def answer() -> List[River]:`
	`new_mexico_state = geo_model.find_state("new mexico")`
	`bordering_states = new_mexico_state.next_to`
	`rivers = []`
	`for state in bordering_states:`
	`rivers.extend(state.rivers)`
	`return rivers`
	`}`
		`def answer(): Option[List[River]] = {`
		`val newMexicoState = geoModel.findState("new mexico")`
		`val borderingStates = newMexicoState.map(_.nextTo)`
		`val rivers = borderingStates.map(_.flatMap(_.rivers))`
		`rivers`
		`}`
			`def answer() -> List[River]:`
			`new_mexico_state = geo_model.find_state("new mexico")`
			`bordering_states = new_mexico_state.next_to`
			`rivers = [r for state in bordering_states for r in state.rivers]`
			`return rivers`

What is the most populated state bordering Oklahoma ?	`def answer() -> State:`
	`oklahoma_state = geo_model.find_state("oklahoma")`
	`return max(oklahoma_state.next_to, key=lambda x: x.population)`
		`def answer(): Option[State] = {`
		`val oklahomaState = geoModel.findState("oklahoma")`
		`val borderingStates = oklahomaState.map(_.nextTo)`
		`borderingStates.flatMap(_.maxByOption(_.population))`
		`}`
			`def answer() -> State:`
			`oklahoma_state = geo_model.find_state("oklahoma")`
			`bordering_states = oklahoma_state.next_to`
			`return max(bordering_states, key=lambda x: x.population)`

Which river goes through the most states ?	`def answer() -> River:`
	`return max(geo_model.rivers, key=lambda x: len(x.traverses))`
		`def answer() -> def answer(): Option[River] = {`
		`val riverWithMostStates = geoModel.rivers.maxBy(_.traverses.length)`
		`riverWithMostStates`
		`}`
			`def answer() -> River:`
			`return max(geo_model.rivers, key=lambda x: len(x.traverses))`

Table 11: Example conversions of predicted Scala programs for GeoQuery to Python along with original (direct) Python annotation. Converted programs can differ from the directly annotated Python programs as the conversions were performed without the original query and hence tended to be faithful to given Scala programs.

````
```python
@dataclass
class State:
    name: str
    abbreviation: str
    country: Country
    area: int
    size: int
    population: int
    density: float
    capital: Optional[City]
    high_point: Place
    low_point: Place
    next_to: List[State]
    cities: List[City]
    places: List[Place]
    mountains: List[Mountain]
    lakes: List[Lake]
    rivers: List[River]

@dataclass
class City:
    name: str
    state: State
    country: Country
    is_capital: bool
    population: int
    size: int
    is_major: bool
    density: float

@dataclass
class Country:
    name: str
    area: int
    population: int
    density: float
    high_point: Place
    low_point: Place
    cities: List[City]
    states: List[State]
    places: List[Place]
    mountains: List[Mountain]
    lakes: List[Lake]
    rivers: List[River]

@dataclass
class River:
    name: str
    traverses: List[State]
    length: int
    size: int
    is_major: bool

@dataclass
class Place:
    name: str
    state: State
    elevation: int
    size: int

@dataclass
class Mountain:
    name: str
    state: State
    elevation: int

@dataclass
class Lake:
    name: str
    area: int
    states: List[State]
````

Figure 10: Domain description for Geoquery, using Python. Continued in Fig. 11.
````

```
@dataclass
class GeoModel:
    countries: List[Country]
    states: List[State]
    cities: List[City]
    rivers: List[River]
    mountains: List[Mountain]
    lakes: List[Lake]
    places: List[Place]

    def find_country(self, name: str) -> Country:
        ...

    def find_state(self, name: str) -> State:
        ...

    def find_city(self, name: str, state_abbreviation: str = None) -> City:
        ...

    def find_river(self, name: str) -> River:
        ...

    def find_mountain(self, name: str) -> Mountain:
        ...

    def find_lake(self, name: str) -> Lake:
        ...

    def find_place(self, name: str) -> Place:
        ...

geo_model = GeoModel()
```
````

Figure 11: Domain description for Geoquery, using Python. Continued from Fig. 10.
````

````
```
cityid(CityName,StateAbbrev)  # given a city name and state, return the city id
countryid(CountryName)  # given a country name, return the country id
placeid(PlaceName)  # given a place (lakes, mountains, etc.) name, return the place id
riverid(RiverName)  # given a river name, return the river id
stateid(StateName)  # given a state name, return the state id

capital(all)  # return all cities that are capitals
city(all)  # return all cities
lake(all)  # return all lakes
mountain(all)  # return all mountains
place(all)  # return all places
river(all)  # return all rivers
state(all)  # return all states

capital(items)  # given a set of cities, return those that are capitals
city(p)  # given a set of items, return those that are cities
lake(p)  # given a set of items, return those that are lakes
major(p)  # given a set of items, return those that are considered of major size
mountain(p)  # given a set of items, return those that are mountains
place(p)  # given a set of items, return those that are places
river(p)  # given a set of items, return those that are rivers
state(p)  # given a set of items, return those that are states
area_1(p)  # given a set of items, return their areas' sizes
capital_1(p)  # given a set of states, return their capitals
capital_2(p)  # given a set of cities, return their states
elevation_1(p)  # given a set of places, return their elevations
elevation_2(E)  # given a set of elevations, return the places with those elevations
high_point_1(p)  # given a set of items, return their highest points
high_point_2(p)  # given a set of places, return the items with those places as their highest points
higher_2(p)  # given a set of places, return the places that are higher than them
loc_1(p)  #  given a set of items, return where each item is located
loc_2(p)  #  given a set of items, return the items located there
longer(p)  # given a set of rivers, return those that are longer than them
lower_2(p)  # given a set of places, return the places that are lower than them
len(p)  # given a set of rivers, return their lengths
next_to_1(p)  # given a set of states, return the states that are next to them
next_to_2(p)  # given a set of states, return the states that this state is next to
population_1(p)  # given a set of cities or states, return their populations
size(p)  # given a set of items, return their sizes (area for state, population for city, length for river)
traverse_1(p)  # given a set of rivers, return the states they traverse
traverse_2(p)  # given a set of states, return the rivers that traverse them

answer(p)  # return as answer (always needed)
largest(p)  # given a set of items, return the item with the largest size
largest_one(area_1(p))  # given a set of items, return the item with the largest area
largest_one(density_1(p))  # given a set of items, return the item with the largest density
largest_one(population_1(p))  # given a set of items, return the item with the largest population
smallest(p)  # given a set of items, return the item with the smallest size
smallest_one(area_1(p))  # given a set of items, return the item with the smallest area
smallest_one(density_1(p))  # given a set of items, return the item with the smallest density
smallest_one(population_1(p))  # given a set of items, return the item with the smallest population
highest(p)  # given a set of items, return the item that is highest
lowest(p)  # given a set of items, return the item that is lowest
longest(p)  # given a set of items, return the item that is longest
shortest(p)  # given a set of items, return the item that is shortest
count(p)  # given a set of items, return the number of items in the set
most(pD)  # given a set of items, return the item that appears most frequently in the set
fewest(pD)  # given a set of items, return the item that appears fewest times in the set

exclude(p1, p2)  # given a set of items, return the items that are in p1 but not in p2
intersect(p1, p2)  # given a set of items, return the items that are in both p1 and p2
```
````

Figure 12: Domain description for Geoquery, using FunQL (NL).

````
```
def cityid(CityName: str, StateAbbrev: str) -> City: ...
def countryid(CountryName: str) -> Country: ...
def placeid(PlaceName: str) -> Place: ...
def riverid(RiverName: str) -> River: ...
def stateid(StateName: str) -> State: ...
def capital(places: List[Place]) -> List[City]: ...
def city(places: List[Place]) -> List[City]: ...
def lake(places: List[Place]) -> List[Lake]: ...
def mountain(places: List[Place]) -> List[Mountain]: ...
def place(places: List[Place]) -> List[Place]: ...
def river(places: List[Place]) -> List[River]: ...
def state(places: List[Place]) -> List[State]: ...
def major(places: List[Place]) -> List[Place]: ...
def area_1(state: State | List[State]) -> List[float]: ...
def capital_1(state: State | List[State]) -> List[City]: ...
def capital_2(city: City | List[City]) -> List[State]: ...
def density_1(state: State | List[State]) -> List[float]: ...
def elevation_1(place: List[Place]) -> List[floats]: ...
def elevation_2(elevation: float) -> List[Place]: ...
def high_point_1(state: State | List[State]) -> List[Place]: ...
def high_point_2(place: Place) -> List[State]: ...
def higher_2(place: Place) -> List[Place]: ...
def loc_1(place: Place | List[Place]) -> List[State]: ...
def loc_2(state: State | List[State]) -> List[Place]: ...
def longer(river: River) -> List[River]: ...
def lower_2(place: Place) -> List[Place]: ...
def len(river: River | List[River]) -> List[float]: ...
def next_to_1(state: State | List[State]) -> List[State]: ...
def next_to_2(state: State | List[State]) -> List[State]: ...
def population_1(state: State | List[State]) -> List[float]: ...
def size(place: List[State] | List[City]) -> List[float]: ...
def traverse_1(river: River | List[River]) -> List[State]: ...
def traverse_2(state: State | List[State] | Country | List[Country]) -> List[River]: ...
def largest(place: List[Place]) -> List[Place]: ...
def largest_one(lst: List[Place]) -> Place: ...
def smallest(place: List[Place]) -> List[Place]: ...
def smallest_one(lst: List[Place]) -> Place: ...

def highest(place: List[Place]) -> List[Place]: ...
def lowest(place: List[Place]) -> List[Place]: ...
def longest(place: List[Place]) -> List[River]: ...
def shortest(place: List[Place]) -> List[River]: ...
def count(place: List[Place]) -> List[int]: ...
def most(place: List[Place]) -> Place: ...
def fewest(place: List[Place]) -> Place: ...

def exclude(lst1: List[Place], lst2: List[Place]) -> List[Place]: ...
def intersect(lst1: List[Place], lst2: List[Place]) -> List[Place]: ...
```
````

Figure 13: Domain description for Geoquery, using FunQL (formal).

```
class State {
    constructor(name, abbreviation, country, area, population, density, capital, high_point, low_point,
    ↪  next_to, cities, places, mountains, lakes, rivers) {
        this.name = name;
        this.abbreviation = abbreviation;
        this.country = country;
        this.area = area;
        this.population = population;
        this.density = density;
        this.capital = capital;
        this.high_point = high_point;
        this.low_point = low_point;
        this.next_to = next_to;
        this.cities = cities;
        this.places = places;
        this.mountains = mountains;
        this.lakes = lakes;
        this.rivers = rivers;
    }
}

class City {
    constructor(name, state, country, is_capital, population, size, is_major) {
        this.name = name;
        this.state = state;
        this.country = country;
        this.is_capital = is_capital;
        this.population = population;
        this.size = size;
        this.is_major = is_major;
    }
}

class Country {
    constructor(name) {
        this.name = name;
    }
}

class River {
    constructor(name, traverses, length, size, is_major) {
        this.name = name;
        this.traverses = traverses;
        this.length = length;
        this.size = size;
        this.is_major = is_major;
    }
}

class Place {
    constructor(name, state, elevation, size) {
        this.name = name;
        this.state = state;
        this.elevation = elevation;
        this.size = size;
    }
}

class Mountain {
    constructor(name, state, elevation) {
        this.name = name;
        this.state = state;
        this.elevation = elevation;
    }
}

class Lake {
    constructor(name, area, states) {
        this.name = name;
        this.area = area;
        this.states = states;
    }
}
```

Figure 14: Domain description for Geoquery, using Javascript. Continued in Fig. 15.

````
class GeoModel {
    constructor(countries, states, cities, rivers, mountains, lakes, places) {
        this.countries = countries;
        this.states = states;
        this.cities = cities;
        this.rivers = rivers;
        this.mountains = mountains;
        this.lakes = lakes;
        this.places = places;
    }

    find_country(name) {
        // ...
    }

    find_state(name) {
        // ...
    }

    find_city(name, state_abbreviation = null) {
        // ...
    }

    find_river(name) {
        // ...
    }

    find_mountain(name) {
        // ...
    }

    find_lake(name) {
        // ...
    }

    find_place(name) {
        // ...
    }
}

let geo_model = new GeoModel();
```
````

Figure 15: Domain description for Geoquery, using Javascript. Continued from Fig. 14.
````

````
```scala
case class Country(name: String)

case class State(name: String, abbreviation: String, country: Country, area: Int, population: Int, density: Float, capital: Option[City], highPoint: Place, lowPoint: Place, nextTo: List[State], cities: List[City], places: List[Place], mountains: List[Mountain], lakes: List[Lake], rivers: List[River])

case class City(name: String, state: State, country: Country, isCapital: Boolean, population: Int, size: Int, isMajor: Boolean)

case class River(name: String, traverses: List[State], length: Int, size: Int, isMajor: Boolean)

case class Place(name: String, state: State, elevation: Int, size: Int)

case class Mountain(name: String, state: State, elevation: Int)

case class Lake(name: String, area: Int, states: List[State])

class GeoModel {
  var countries: List[Country] = List()
  var states: List[State] = List()
  var cities: List[City] = List()
  var rivers: List[River] = List()
  var mountains: List[Mountain] = List()
  var lakes: List[Lake] = List()
  var places: List[Place] = List()

  def findCountry(name: String): Option[Country] = ???

  def findState(name: String): Option[State] = ???

  def findCity(name: String, stateAbbreviation: Option[String] = None): Option[City] = ???

  def findRiver(name: String): Option[River] = ???

  def findMountain(name: String): Option[Mountain] = mountains.find(_.name == name)

  def findLake(name: String): Option[Lake] = lakes.find(_.name == name)

  def findPlace(name: String): Option[Place] = places.find(_.name == name)
}

val geoModel = new GeoModel()
```
````

Figure 16: Domain description for Geoquery, using Scala.

```python
```python
@dataclass
class Person:
    name: str

    def find_team_of() -> List["Person"]:
        ...

    def find_reports_of() -> List["Person"]:
        ...

    def find_manager_of() -> "Person":
        ...

@dataclass
class Event:
    attendees: List[Person] = None
    attendees_to_avoid: List[Person] = None
    subject: Optional[str] = None
    location: Optional[str] = None
    starts_at: Optional[List[DateTimeClause]] = None
    ends_at: Optional[List[DateTimeClause]] = None
    duration: Optional["TimeUnit"] = None
    show_as_status: Optional["ShowAsStatus"] = None

DateTimeValues = Enum("DateTimeValues", ["Afternoon", "Breakfast", "Brunch", "Dinner", "Early",
↪  "EndOfWorkDay", "Evening",
    "FullMonthofMonth", "FullYearofYear", "LastWeekNew", "Late", "LateAfternoon", "LateMorning",
    ↪  "Lunch", "Morning",
    "NextMonth", "NextWeekend", "NextWeekList", "NextYear", "Night", "Noon", "Now", "SeasonFall",
    ↪  "SeasonSpring",
    "SeasonSummer", "SeasonWinter", "ThisWeek", "ThisWeekend", "Today", "Tomorrow", "Yesterday"])

class DateTimeClause:
    def get_by_value(date_time_value: DateTimeValues) -> "DateTimeClause": ...
    def get_next_dow(day_of_week: str) -> "DateTimeClause": ...
    def date_by_mdy(month: int = None, day: int = None, year: int = None) -> "DateTimeClause": ...
    def time_by_hm(hour: int = None, minute: int = None, am_or_pm: str = None) -> "DateTimeClause":
    ↪  ...
    def on_date_before_date_time(date: "DateTimeClause", time: "DateTimeClause") -> "DateTimeClause":
    ↪  ...
    def on_date_after_date_time(date: "DateTimeClause", time: "DateTimeClause") -> "DateTimeClause":
    ↪  ...
    def around_date_time(date_time: "DateTimeClause") -> "DateTimeClause": ...


TimeUnits = Enum("TimeUnits", ["Hours", "Minutes", "Days"])
TimeUnitsModifiers = Enum("TimeUnitsModifiers", ["Acouple", "Afew"])

@dataclass
class TimeUnit:
    number: Optional[Union[int,float]] = None
    unit: Optional[TimeUnits] = None
    modifier: Optional[TimeUnitsModifiers] = None

ShowAsStatusType = Enum("ShowAsStatusType", ["Busy", "OutOfOffice"])


class API:
    def find_person(name: str) -> Person:
        ...

    def get_current_user() -> Person:
        ...

    def add_event(event: Event) -> None:
        ...

    def find_event(attendees: Optional[List[Person]] = None, subject: Optional[str] = None) -> Event:
        ...

api = API()
```
```

Figure 17: Domain description for SMCalFlow, using Python.

````
```
Yield  # Arguments: (1) :output, the function to be executed. Returns: The result of the function.
CreateCommitEventWrapper   # Arguments: (1) :event, containing event details. Returns: The created event.
CreatePreflightEventWrapper  # Arguments: (1) :constraint, containing event details. Returns: The event that satisfies the constraint.
FindEventWrapperWithDefaults  # Arguments: (1) :constraint, the constraint to be satisfied by the event. Returns: The event that satisfies the constraint.

extensionConstraint  # Arguments: (1) the type of constraint (e.g., Constraint[Recipient], Constraint[Date], RecipientWithNameLike), Returns: A constraint that needs to be satisfied by the entity.
Constraint[Event]  # Arguments: (1) :attendees, :start, :subject or :location. Returns: Constraints to create or find an event.
Constraint[DateTime]  # Arguments: (1) :date, the date constraint. Returns: A constraint that needs to be satisfied by the date and time.
andConstraint  # Arguments: Any number of constraints. Returns: A constraint that is satisfied when all the input constraints are satisfied.
RecipientWithNameLike  # Arguments: (1) :constraint, the type of constraint (e.g., Constraint[Recipient]), (2) :name, the name of the recipient. Returns: A constraint that needs to be satisfied by the recipient.
PersonName  # Arguments: (1) the name of the person. Returns: The name of the person. e.g. `PersonName " Dan "`
AttendeeListHasRecipient  # Arguments: (1) :recipient, the recipient to be included. Returns: A constraint for the event.
AttendeeListHasPeople  # Arguments: (1) :people, the group of people to be included. Returns: A constraint for the event.
AttendeeListHasRecipientConstraint  # Arguments: (1) :recipientConstraint, the recipient constrained to be included. Returns: A constraint for the event.
DateTimeConstraint  # Arguments: (1) :constraint, the time constraint, (2) :date, the date. Returns: A constraint that needs to be satisfied by the date and time.
AttendeeListExcludesRecipient  # Arguments: (1) :recipient, the recipient to be excluded. Returns: A constraint for the event.

Execute  # Arguments: (1) :intension, the intension to be executed. Returns: The entity referred to by the intension.
refer  # Arguments: (1) extensionConstraint, the constraint to be satisfied by the entity. Returns: A reference to the entity that satisfies the constraint.
singleton  # Arguments: (1) an element or a list with an element. Returns: The single element.
do  # Arguments: Any number of functions. Returns: The results of the functions.
String  # Arguments: (1) a literal string. Returns: A string representation.

FindManager  # Arguments: (1) :recipient, the recipient whose manager is to be found. Returns: The manager of the recipient.
FindReports  # Arguments: (1) :recipient, the recipient whose reports are to be found. Returns: The reports of the recipient.
FindTeamOf  # Arguments: (1) :recipient, the recipient whose team is to be found. Returns: The group of people who make up the recipient's team.
toRecipient  # Arguments: (1) A user. Returns: The given user as a recipient.
CurrentUser  # Arguments: None. Returns: The current user.

LocationKeyphrase  # Arguments: (1) the location. Returns: The location. e.g. `LocationKeyphrase " office "`
roomRequest  # Arguments: None. Returns: A request for a room.

# These operators represent specific times or dates. They have no arguments and return the specified time or date.
Today
Tomorrow
NextWeekList
NextDOW
Noon
Afternoon
Morning
Night
EndOfWorkDay
Evening
Weekend
ThisWeekend
ThisWeek
Early
Now
NextYear
Lunch

# These operators represent specific numbers or convert values to numbers. Arguments: (1) the number or value to be converted. Returns: The specific number or the converted value.
Number
NumberAM
NumberPM
````

Figure 18: Domain description for SMCalFlow, using DataFlow. Continued in Fig. 19.
````

````
toDays
toHours
toMinutes
DateAndConstraint  # Arguments: (1) :date1, the first date, (2) :date2, the second date. Returns: The date and constraint.
DateAtTimeWithDefaults  # Arguments: (1) :date, the date, (2) :time, the time. Returns: The date and time.
nextDayOfMonth  # Arguments: (1) the day of the month. Returns: The next occurrence of the day of the month.
DayOfWeek  # Arguments: (1) the day of the week. Returns: The day of the week.
DowOfWeekNew  # Arguments: (1) :dow, the day of the week, (2) :week, the week. Returns: The day of the week in the week.
previousDayOfWeek  # Arguments: (1) :dayOfWeek, the day of the week. Returns: The previous occurrence of the day of the week.
NextDOW  # Arguments: (1) :dow, the day of the week. Returns: The next occurrence of the day of the week.
EventAllDayStartingDateForPeriod  # Arguments: (1) :event, the event, (2) :period, the duration of the event, (3) :startDate, the start date of the event. Returns: The event with the specified start date and duration.
PeriodDuration  # Arguments: (1) :duration, the duration. Returns: The period duration.

MD  # Arguments: (1) :day, the day, (2) :month, the month. Returns: The date.
MDY  # Arguments: (1) :day, the day, (2) :month, the month, (3) :year, the year. Returns: The date.
Month  # Arguments: (1) the month. Returns: The month.
NextTime  # Arguments: (1) :time, the time. Returns: The next occurrence of the time.
HourMinuteAm  # Arguments: (1) :hours, the hours, (2) :minutes, the minutes. Returns: The time.
HourMinutePm  # Arguments: (1) :hours, the hours, (2) :minutes, the minutes. Returns: The time.
FullMonthofMonth  # Arguments: (1) :month, the month. Returns: The full month.

TimeAfterDateTime  # Arguments: (1) :dateTime, the date and time, (2) :time, the time after the date and time. Returns: The time after the date and time.
OnDateAfterTime  # Arguments: (1) :date, the date, (2) :time, the time after the date. Returns: The date after the time.
AroundDateTime  # Arguments: (1) :dateTime, the date and time. Returns: The time around the date and time.
```
````

Figure 19: Domain description for SMCalFlow, using DataFlow. Continued from Fig. 18.
````

````
```
FindTeamOf  # given a person name or id, returns a pseudo-person representing the team of that person
FindReports  # given a person name or id, returns a pseudo-person representing the reports of that
↪  person
FindManager  # given a person name or id, returns the manager of that person

with_attendee  # given a person name or id, returns a clause to match or create an event with that
↪  person as an attendee
avoid_attendee  # given a person name or id, returns an event clause to avoid that attendee when
↪  creating an event
has_subject  # given a string, returns an event to match or create an event with that subject
at_location  # given a string, returns an event clause to match or create an event at that location
starts_at  # given a datetime clause, returns an event clause to match or create an event starting at
↪  that time
ends_at  # given a datetime clause, returns an event clause to match or create an event ending at that
↪  time
has_duration  # given a time unit value, returns an event clause to match or create an event with that
↪  duration
has_status  # given a ShowAsStatus value, returns an event clause to match or create an event with
↪  that status

# the following operators return datetime clauses and accept no arguments
Afternoon
Breakfast
Brunch
Dinner
Early
EndOfWorkDay
Evening
FullMonthofMonth
FullYearofYear
LastWeekNew
Late
LateAfternoon
LateMorning
Lunch
Morning
NextMonth
NextWeekend
NextWeekList
NextYear
Night
Noon
Now
SeasonFall
SeasonSpring
SeasonSummer
SeasonWinter
ThisWeek
ThisWeekend
Today
Tomorrow
Yesterday

# general date time clauses
DateTime  # given either a datetime clause representing a date and/or a time operator representing a
↪  time, returns a datetime clause
Date  # given a date or dayofweek, returns a date
DayOfWeek  # given a day of week string, returns a time clause
NextDOW  # given a day of week string, returns a time clause for the next occurrence of that day of
↪  week
MD  # given a month and day as arguments, returns a date clause
MDY  # given a month, day, and year as arguments, returns a date clause

# given a value, the following operators return datetime clauses according to the given value
toMonth
toFourDigitYear
HourMinuteAm
HourMinutePm
NumberAM
NumberPM

# given a datetime clause, the following operators modify the clause and return a datetime clause
↪  according to the modification
OnDateAfterTime
OnDateBeforeTime
AroundDateTime
````

Figure 20: Domain description for SMCalFlow, using DataFlow Simple. Continued in Fig. 21.
````

```
# given either a number or the operators Acouple/Afew, all the following operators return time unit
↪  values according to the given unit
toDays
toHours
toMinutes

# these operators can be used to create time unit values instead of using integer values
Acouple
Afew

ShowAsStatus  # enumeration of possible event statuses (Busy, OutOfOffice)

CreateEvent  # given multiple event clauses (such as with_attendee, has_subject, combined together
↪  with `AND`), creates an event complying with those clauses
FindEvents  # given multiple event clauses (such as with_attendee, has_subject, combined together with
↪  `AND`), returns a list of events complying with those clauses
CurrentUser  # returns the current user (person)

do  # allows the execution of multiple commands in a single prompt (each command is an argument).
↪  Often used in conjunction with `Let` to define variables
Let  # defines a variable (first argument) with a value (second argument)

AND  # combines multiple event clauses together
```
```
```

Figure 21: Domain description for SMCalFlow, using DataFlow Simple. Continued from Fig. 20.

````
```javascript
class Person {
    constructor(name) {
        this.name = name;
    }

    find_team_of() {
        // ...
    }

    find_reports_of() {
        // ...
    }

    find_manager_of() {
        // ...
    }
}

class Event {
    constructor(attendees = null, attendees_to_avoid = null, subject = null, location = null, starts_at = null, ends_at = null, duration = null, show_as_status = null) {
        this.attendees = attendees;
        this.attendees_to_avoid = attendees_to_avoid;
        this.subject = subject;
        this.location = location;
        this.starts_at = starts_at;
        this.ends_at = ends_at;
        this.duration = duration;
        this.show_as_status = show_as_status;
    }
}

const DateTimeValues = ["Afternoon", "Breakfast", "Brunch", "Dinner", "Early", "EndOfWorkDay", "Evening",
    "FullMonthofMonth", "FullYearofYear", "LastWeekNew", "Late", "LateAfternoon", "LateMorning", "Lunch", "Morning",
    "NextMonth", "NextWeekend", "NextWeekList", "NextYear", "Night", "Noon", "Now", "SeasonFall", "SeasonSpring",
    "SeasonSummer", "SeasonWinter", "ThisWeek", "ThisWeekend", "Today", "Tomorrow", "Yesterday"];

class DateTimeClause {
    get_by_value(date_time_value) {
        // ...
    }

    get_next_dow(day_of_week) {
        // ...
    }

    date_by_mdy(month = null, day = null, year = null) {
        // ...
    }

    time_by_hm(hour = null, minute = null, am_or_pm = null) {
        // ...
    }

    on_date_before_date_time(date, time) {
        // ...
    }

    on_date_after_date_time(date, time) {
        // ...
    }

    around_date_time(date_time) {
        // ...
    }
}

const TimeUnits = ["Hours", "Minutes", "Days"];
const TimeUnitsModifiers = ["Acouple", "Afew"];
````

Figure 22: Domain description for SMCalFlow, using Javascript. Continued in Fig. 23.
````

````javascript
class TimeUnit {
    constructor(number = null, unit = null, modifier = null) {
        this.number = number;
        this.unit = unit;
        this.modifier = modifier;
    }
}

const ShowAsStatusType = ["Busy", "OutOfOffice"];

class API {
    find_person(name) {
        // ...
    }

    get_current_user() {
        // ...
    }

    add_event(event) {
        // ...
    }

    find_event(attendees = null, subject = null) {
        // ...
    }
}

const api = new API();
```
````

Figure 23: Domain description for SMCalFlow, using Javascript. Continued from Fig. 22.
````

````scala
```scala
case class Person(name: String) {
  def findTeamOf(): List[Person] = ???
  def findReportsOf(): List[Person] = ???
  def findManagerOf(): Person = ???
}

case class Event(var attendees: Option[List[Person]] = None,
                 var attendeesToAvoid: Option[List[Person]] = None,
                 var subject: Option[String] = None,
                 var location: Option[String] = None,
                 var startsAt: Option[List[DateTimeClause]] = None,
                 var endsAt: Option[List[DateTimeClause]] = None,
                 var duration: Option[TimeUnit] = None,
                 var showAsStatus: Option[ShowAsStatusType.Value] = None)

object DateTimeValues extends Enumeration {
  val Afternoon, Breakfast, Brunch, Dinner, Early, EndOfWorkDay, Evening,
  FullMonthofMonth, FullYearofYear, LastWeekNew, Late, LateAfternoon, LateMorning, Lunch, Morning,
  NextMonth, NextWeekend, NextWeekList, NextYear, Night, Noon, Now, SeasonFall, SeasonSpring,
  SeasonSummer, SeasonWinter, ThisWeek, ThisWeekend, Today, Tomorrow, Yesterday = Value
}

class DateTimeClause {
  def getByValue(dateTimeValue: DateTimeValues.Value): DateTimeClause = ???
  def getNextDow(dayOfWeek: String): DateTimeClause = ???
  def dateByMdy(month: Option[Int] = None, day: Option[Int] = None, year: Option[Int] = None): DateTimeClause = ???
  def timeByHm(hour: Option[Int] = None, minute: Option[Int] = None, amOrPm: Option[String] = None): DateTimeClause = ???
  def onDateBeforeDateTime(date: DateTimeClause, time: DateTimeClause): DateTimeClause = ???
  def onDateAfterDateTime(date: DateTimeClause, time: DateTimeClause): DateTimeClause = ???
  def aroundDateTime(dateTime: DateTimeClause): DateTimeClause = ???
}

object TimeUnits extends Enumeration {
  val Hours, Minutes, Days = Value
}

object TimeUnitsModifiers extends Enumeration {
  val Acouple, Afew = Value
}

case class TimeUnit(var number: Option[Either[Int, Double]] = None,
                    var unit: Option[TimeUnits.Value] = None,
                    var modifier: Option[TimeUnitsModifiers.Value] = None)

object ShowAsStatusType extends Enumeration {
  val Busy, OutOfOffice = Value
}

class API {
  def findPerson(name: String): Person = ???
  def getCurrentUser(): Person = ???
  def addEvent(event: Event): Unit = ???
  def findEvent(attendees: Option[List[Person]] = None, subject: Option[String] = None): Event = ???
}

val api = new API
```
````

Figure 24: Domain description for SMCalFlow, using Scala.

````
```python
Gender = Enum('Gender', 'male,female')
RelationshipStatus = Enum('RelationshipStatus', 'single,married')
Education = NamedTuple('Education', [('university', str), ('field_of_study', str), ('start_date', int), ('end_date', int)])
Employment = NamedTuple('Employment', [('employer', str), ('job_title', str), ('start_date', int), ('end_date', int)])

@dataclass
class Person:
    name: str
    gender: Gender
    relationship_status: RelationshipStatus
    height: int
    birthdate: int
    birthplace: str
    friends: List['Person'] = None
    logged_in: bool = False

    education: List[Education] = None
    employment: List[Employment] = None


@dataclass
class API:
    people: List[Block]

    def find_person_by_id(self, block_id: str) -> Person:
        ...

api = API()
```
````

Figure 25: Domain description for Overnight, using Python.

````
```
call  # invoke a function. Arguments: (1) function to be invoked, (2 and subsequent) parameters to be passed to that function or method. Returns: the result of the function call.
SW.listValue  # extract values from an object. Arguments: (1) An object of any type. Returns: A list of values.
SW.filter  # applies a filter to a list of objects. Arguments: (1) A list of objects, (2) A property to filter on, (3) A comparison operator, (4) A value to compare against. If property is boolean (unary), arguments: (1) A list of objects, (2) Unary property to filter on. Returns: A list of objects that pass the filter.
SW.getProperty  # retrieves a property from an object. Arguments: (1) An object, (2) A property name. Returns: The value of the property.
SW.reverse  # reverses the direction of a property. Arguments: (1) A property name. Returns: The reversed property name.
SW.singleton  # creates a singleton set containing a single object. Arguments: (1) An object. Returns: A singleton set containing the object.
SW.domain  # retrieves the domain of a property, which is the set of entities or objects that the property can be applied to. Arguments: (1) A property name. Returns: The set of entities that can have the property.
SW.countSuperlative  # finds the object(s) with the minimum or maximum count of a certain property. Arguments: (1) A list of objects, (2) A superlative operator (min or max), (3) A property to count, (4) A list of objects to count from. Returns: The object(s) with the minimum or maximum count of the property.
SW.ensureNumericProperty  # ensures that a property is treated as numeric for comparison purposes. Arguments: (1) A property name. Returns: The property name, treated as numeric.
SW.ensureNumericEntity  # ensures that an entity is treated as numeric for comparison purposes. Arguments: (1) An entity. Returns: The entity, treated as numeric.
SW.size  # retrieves the size of a collection. Arguments: (1) A collection of objects. Returns: The size of the collection as a numeric value.
SW.aggregate  # applies an aggregate function to a property over a set of objects. Arguments: (1) An aggregate function (e.g., sum, avg, min, max), (2) A property to aggregate over, (3) A set of objects. Returns: The result of the aggregate function.
SW.concat  # concatenates two or more strings or lists. Arguments: (1 and subsequent) Strings or lists to concatenate. Returns: The concatenated result.
SW.countComparative  # compares the count of a property over a set of objects with a given number. Arguments: (1) A set of objects, (2) A property to count, (3) A comparison operator, (4) A number to compare against, (5) A set of objects to count from. Returns: The objects for which the count of the property satisfies the comparison.
SW.superlative  # finds the object(s) with the minimum or maximum value of a certain property. Arguments: (1) A set of objects, (2) A superlative operator (min or max), (3) A property to compare. Returns: The object(s) with the minimum or maximum value of the property.

lambda  # creates a function. Arguments: (1) A variable name, (2) A function body. Returns: A function.
var  # references a variable. Arguments: (1) A variable name. Returns: The value of the variable.
string  # creates a string. Arguments: (1) A string value. Returns: The string.
number  # creates a number. Arguments: (1) A numeric value, (2) A unit (optional). Returns: The number.
date  # creates a date. Arguments: (1) Year, (2) Month, (3) Day. Returns: The date.

# The following are namespaces for different types of entities.
en.person
en.company
en.university
en.relationship_status
en.employee
en.student
en.field
en.city
en.gender

# specific entities under these namespaces:
en.gender.female
en.gender.male
en.relationship_status
en.relationship_status.single
en.relationship_status.married

# en.person properties:
height  # property of type (number with unit en.cm)
birthdate  # property of type date
birthplace  # property of type en.city
logged_in  # property of type bool
friend  # property of type en.person
relationship_status  # property of type en.relationship_status
```
````

Figure 26: Domain description for Overnight, using λ-DCS. Continued in Fig. 27.

```
# education properties:
student  # property of type en.person
university  # property of type en.university
field_of_study  # property of type en.field
education_start_date  # property of type date
education_end_date  # property of type date

# employment properties:
employee  # property of type en.person
employer  # property of type en.company
job_title  # property of type string
employment_start_date  # property of type date
employment_end_date  # property of type date
```

Figure 27: Domain description for Overnight, using λ-DCS. Continued from Fig. 26.

```
listValue  # extract values from an object. Arguments: (1) An object of any type. Returns: A list of values.
filter  # applies a filter to a list of objects. Arguments: (1) A list of objects, (2) A property to filter on, (3) A comparison operator, (4) A value to compare against. If property is boolean (unary), arguments: (1) A list of objects, (2) Unary property to filter on. Returns: A list of objects that pass the filter.
getProperty  # retrieves a property from an object. Arguments: (1) An object, (2) A property name. Returns: The value of the property.
reverse  # reverses the direction of a property. Arguments: (1) A property name. Returns: The reversed property name.
singleton  # creates a singleton set containing a single object. Arguments: (1) An object. Returns: A singleton set containing the object.
domain  # retrieves the domain of a property, which is the set of entities or objects that the property can be applied to. Arguments: (1) A property name. Returns: The set of entities that can have the property.
countSuperlative  # finds the object(s) with the minimum or maximum count of a certain property. Arguments: (1) A list of objects, (2) A superlative operator (min or max), (3) A property to count, (4) A list of objects to count from. Returns: The object(s) with the minimum or maximum count of the property.
ensureNumericProperty  # ensures that a property is treated as numeric for comparison purposes. Arguments: (1) A property name. Returns: The property name, treated as numeric.
ensureNumericEntity  # ensures that an entity is treated as numeric for comparison purposes. Arguments: (1) An entity. Returns: The entity, treated as numeric.
size  # retrieves the size of a collection. Arguments: (1) A collection of objects. Returns: The size of the collection as a numeric value.
aggregate  # applies an aggregate function to a property over a set of objects. Arguments: (1) An aggregate function (e.g., sum, avg, min, max), (2) A property to aggregate over, (3) A set of objects. Returns: The result of the aggregate function.
concat  # concatenates two or more strings or lists. Arguments: (1 and subsequent) Strings or lists to concatenate. Returns: The concatenated result.
countComparative  # compares the count of a property over a set of objects with a given number. Arguments: (1) A set of objects, (2) A property to count, (3) A comparison operator, (4) A number to compare against, (5) A set of objects to count from. Returns: The objects for which the count of the property satisfies the comparison.
superlative  # finds the object(s) with the minimum or maximum value of a certain property. Arguments: (1) A set of objects, (2) A superlative operator (min or max), (3) A property to compare. Returns: The object(s) with the minimum or maximum value of the property.

lambda  # creates a function. Arguments: (1) A variable name, (2) A function body. Returns: A function.
var  # references a variable. Arguments: (1) A variable name. Returns: The value of the variable.

# The following are namespaces for different types of entities.
en.person
en.company
en.university
en.relationship_status
en.employee
en.student
en.field
en.city
en.gender

# specific entities under these namespaces:
en.gender.female
en.gender.male
en.relationship_status
en.relationship_status.single
en.relationship_status.married

# en.person properties:
height  # property of type (number with unit en.cm)
birthdate  # property of type date
birthplace  # property of type en.city
logged_in  # property of type bool
friend  # property of type en.person
relationship_status  # property of type en.relationship_status

# education properties:
student  # property of type en.person
university  # property of type en.university
field_of_study  # property of type en.field
education_start_date  # property of type date
education_end_date  # property of type date

# employment properties:
employee  # property of type en.person
employer  # property of type en.company
job_title  # property of type string
employment_start_date  # property of type date
employment_end_date  # property of type date
```

Figure 28: Domain description for Overnight, using λ-DCS Simple..

````
```javascript
const Gender = Object.freeze({"male":1, "female":2});
const RelationshipStatus = Object.freeze({"single":1, "married":2});

class Education {
    constructor(university, field_of_study, start_date, end_date) {
        this.university = university;
        this.field_of_study = field_of_study;
        this.start_date = start_date;
        this.end_date = end_date;
    }
}

class Employment {
    constructor(employer, job_title, start_date, end_date) {
        this.employer = employer;
        this.job_title = job_title;
        this.start_date = start_date;
        this.end_date = end_date;
    }
}

class Person {
    constructor(name, gender, relationship_status, height, birthdate, birthplace, friends = [],
    ↪  logged_in = false, education = [], employment = []) {
        this.name = name;
        this.gender = gender;
        this.relationship_status = relationship_status;
        this.height = height;
        this.birthdate = birthdate;
        this.birthplace = birthplace;
        this.friends = friends;
        this.logged_in = logged_in;
        this.education = education;
        this.employment = employment;
    }
}

class API {
    constructor(people = []) {
        this.people = people;
    }

    find_person_by_id(block_id) { ... }
}

let api = new API();
```
````

Figure 29: Domain description for Overnight, using Javascript. Continued from Fig. 22.

````scala
```scala
object Gender extends Enumeration {
  type Gender = Value
  val Male, Female = Value
}

object RelationshipStatus extends Enumeration {
  type RelationshipStatus = Value
  val Single, Married = Value
}

case class Education(university: String, fieldOfStudy: String, startDate: LocalDate, endDate: LocalDate)
case class Employment(employer: String, jobTitle: String, startDate: LocalDate, endDate: LocalDate)

case class Person(
  name: String,
  gender: Gender.Gender,
  relationshipStatus: RelationshipStatus.RelationshipStatus,
  height: Int,
  birthdate: LocalDate,
  birthplace: String,
  friends: Option[List[Person]] = None,
  loggedIn: Boolean = false,
  education: Option[List[Education]] = None,
  employment: Option[List[Employment]] = None
)

class API {
  var people: List[Person] = List()

  def findPersonById(personId: String): Person = ???
}

val api = new API
```
````

Figure 30: Domain description for Overnight, using Scala.