NoviCode: Generating Programs from Natural Language Utterances by Novices

Asaf Achi Mordechai **Yoav Goldberg** **Reut Tsarfaty**
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel
{asaf.achimordechai, yoav.goldberg, reut.tsarfaty}@gmail.com

Abstract

Current Text-to-Code models demonstrate impressive capabilities in generating executable code from natural language snippets. However, current studies focus on technical instructions and programmer-oriented language, and it is an open question whether these models can effectively translate natural language descriptions given by non-technical users and express complex goals, to an executable program that contains an intricate flow—composed of API access and control structures as loops, conditions, and sequences. To unlock the challenge of generating a complete program from a plain non-technical description we present NoviCode, a novel NL Programming task, which takes as input an API and a natural language description by a novice non-programmer, and provides an executable program as output. To assess the efficacy of models on this task, we provide a novel benchmark accompanied by test suites wherein the generated program code is assessed not according to their form, but according to their functional execution. Our experiments show that, first, NoviCode is indeed a challenging task in the code synthesis domain, and that generating complex code from non-technical instructions goes beyond the current Text-to-Code paradigm. Second, we show that a novel approach wherein we align the NL utterances with the compositional hierarchical structure of the code, greatly enhances the performance of LLMs on this task, compared with the end-to-end Text-to-Code counterparts.

1 Introduction

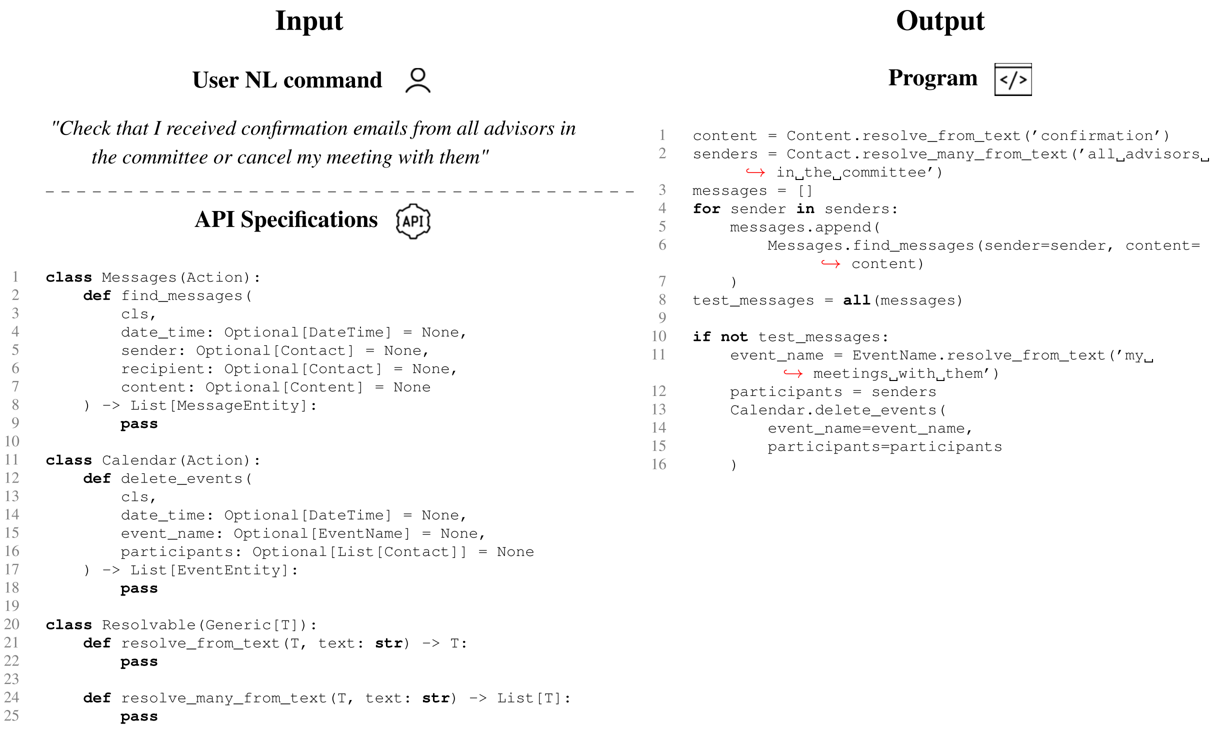
The current Text-to-Code paradigm focuses on generating *code-lines* from technical descriptions produced by trained programmers. In this work, we move from this Text-to-Code paradigm towards generating *programs* with intricate structures from intuitive, *day-to-day language descriptions* produced by *non-technical* individuals. This novel reconstruction of the task is challenging on two levels: (a) the generated programs have non-trivial control-flow structures—with API calls to a novel API that the model was not necessarily trained on—rather than generic one-liners; and (b) we are interested in instructions provided by laypeople in everyday natural language, without using technical concepts or jargon. We term this task *Natural Language Programming* (NLProg).

NLProg differs from standard code generation models, tasked to convert natural language instructions or descriptions into executable code, in several ways. In the Text-to-Code paradigm, models require the user to use technical terms, like flow structures (e.g., conditions or loops), data types, variables, and programming concepts (e.g., sorting algorithms, object-oriented concepts, recursions, etc.). In contrast, in NLProg we aim to convert plain, everyday language descriptions, devoid of any technical jargon, into functional code using independent and unseen API specifications.

NLProg is particularly challenging when the descriptions implicitly allude to code constructs as control flow elements such as sequences, loops, or conditional statements, *without* explicitly mentioning them. For example, given a standard API of email and calendar applications and a user request ‘‘*Check that I received confirmation emails from all advisors in the committee or cancel my meeting with them*’’, we expect the model to interpret the conjunction ‘‘*or*’’ as indicating a condition, and to recognize ‘‘*all advisors in the committee*’’ as an iteration over a particular set. The model should then generate a proper executable program based on the API (Figure 1).

Recent work on natural language synthesis to code (Chen et al., 2021; Wang et al., 2023a;

Transactions of the Association for Computational Linguistics, vol. 12, pp. 1330–1345, 2024. https://doi.org/10.1162/tacl_a_00694
Action Editor: Shay Cohen. Submission batch: 1/2024; Revision batch: 4/2024; Published 10/2024.



"Check that I received confirmation emails from all advisors in the committee or cancel my meeting with them"

API Specifications

```
class Messages(Action):
    def find_messages(
        cls,
        date_time: Optional[DateTime] = None,
        sender: Optional[Contact] = None,
        recipient: Optional[Contact] = None,
        content: Optional[Content] = None
    ) -> List[MessageEntity]:
        pass

class Calendar(Action):
    def delete_events(
        cls,
        date_time: Optional[DateTime] = None,
        event_name: Optional[EventName] = None,
        participants: Optional[List[Contact]] = None
    ) -> List[EventEntity]:
        pass

class Resolvable(Generic[T]):
    def resolve_from_text(T, text: str) -> T:
        pass

    def resolve_many_from_text(T, text: str) -> List[T]:
        pass
```

```
content = Content.resolve_from_text(’confirmation’)
senders = Contact.resolve_many_from_text(’all␣advisors␣
      ↪ in␣the␣committee’)
messages = []
for sender in senders:
    messages.append(
        Messages.find_messages(sender=sender, content=
             ↪ content)
    )
test_messages = all(messages)

if not test_messages:
    event_name = EventName.resolve_from_text(’my␣
         ↪ meetings␣with␣them’)
    participants = senders
    Calendar.delete_events(
        event_name=event_name,
        participants=participants
    )
```

Figure 1: The NLProg Task. Left: The input, consisting of an utterance and an API specification for implementing the target code. Right: The output, reflecting the procedural instructions in the utterance, is implemented with the correct usage of the API specifications.

Nijkamp et al., 2023) have shown remarkable success in code generation tasks using LLMs. Trained on a large fraction of GitHub repositories (Iyer et al., 2018; Husain et al., 2019; Lu et al., 2021; Wang et al., 2021; Wang et al., 2023a) or StackOverflow question-answer pairs (Yin et al., 2018), these code-oriented LLMs learn rich contextual representations that can be transferred to various code-related downstream tasks.

However, these works share several limitations. They predominantly focus on technically precise descriptions *given by programmers*, using jargon-laden, code-centric discourse geared towards seasoned programmers. Other datasets in intent recognition and slot fillings (Gupta et al., 2018; Chen et al., 2020) focus on plain language description from non-programmers, or novices, but these are translated into *simple code* that exhibits no complexity in its flow execution and is inherently derived from the NL descriptions in its datasets.

To bridge this gap, we present NoviCode, a novel task that aims to translate NL descriptions from novices into complex executable programs. We curated a dataset where novice-centric NL instructions are paired with complex code programs. These code programs adhere to API specifications to facilitate the functional execution of the produced code. Having defined this task and created the benchmark, two additional challenges present themselves, namely: (i) How can we evaluate the efficacy of models on such complex code synthesis tasks? and (ii) What would be good models for addressing the non-trivial challenge of translating high-level language to low-level intricate code?

To address the evaluation challenge, we deliver, along with the task data, test suites that simulate diverse scenarios of execution flows to gauge the functional correctness of the generated code. Our evaluation methodology is designed to assess the functional correctness of the control flow elements within the generated complex programs, rather than merely looking at the *form* of the code or assessing an *end state* only.

To address the modeling challenge, we hypothesize that learning the alignment of the natural language queries with codes' hierarchical structures, rather than learning a simple end-to-end text-to-code model, can improve performance, particularly with programs as complex as we target here.

Our experiments with standard LLMs assessed by our functional evaluation suites demonstrate that the task is indeed challenging. Existing models struggle to produce the expected executable code with complex constructs. We further empirically confirm our hypothesis that learning the alignment of natural language spans to compositional hierarchical code structures is better than the de-facto standard end-to-end modeling.

The contribution of this paper is thus multifold. First, we define a novel code-synthesis task based on plain non-technical natural language. Second, we deliver a dataset of NL instructions obtained from non-programmers, paired with an expert-crafted evaluation suite that focuses on the function, rather than form of the generated code. Finally, we provide a novel modeling approach that aligns NL spans with the explicit compositional structures of the code. We further show that this approach outperforms the standard end-to-end baseline models on this task, even with the most capable contemporary generative LLMs.

2 The Challenge: Technical Code from Non-Technical Users

2.1 Terminology

In this work we are interested in synthesizing complex programs from descriptions by novice users. Before we begin, we define the relevant terms:

Natural Language Programming. We equate programming in natural language with the ability to translate a desired scenario using plain, everyday language to a working program (Harel, 2008).

Novices vs. Programmers. When crafting NL descriptions of programs, we distinguish two distinct types of personas, defined by their programming skills: **Programmers**, who often use technical terminology—such as functions, loops, iterations, conditions, exceptions, methods, and concepts that are common when expressing computations—which is not typically familiar to individuals without formal training in the field. **Novices**, who are devoid of programming skills, utilizing intuitive, everyday language to describe program functionality without using technical terms or referring to specific programming paradigms.

Simple vs. Complex Programs. In this work, we define a **simple** program as a simple statement that abstains from long sequences of actions or the incorporation of logical control-flow elements. A program is deemed **complex** if it executes a sequence of statements or has one or more control flow structures, such as loops or conditions.

2.2 Why is Programming in Natural Language Challenging?

Programmers and novices describe code differently—with programmers describing explicitly elements in the target code, like functions, arguments, and variable names. Programmers express ideas using technical jargon and programming concepts used in the code.

Let's consider for example the following articulation of a *simple* program by a programmer: ``*Start a server listening on port 8080*''. This example shows a close coupling of the description and the program code. This description explicitly specifies the arguments (*port*) and their respective values (*8080*) in the *start server* API. The ontological terms (*server* and *port*) in the description also exemplify the technical language used by programmers. Last, the description uses code concepts like *Start a server* and *listen on port*.

Moreover, when describing a *complex* program in natural language, programmers typically use a structured language to describe the logical control flows within the desired program. In their description, specific keywords (e.g., *if-else* and *for* or *while*) are used to describe conditionals or loops. Loops are also described using explicit quantifiers. Control flow structures, like sequences of operations, are often brought in the plain order of execution. For example, ``*Find the average of every three columns in Pandas DataFrame and then return the max value*''. This description exhibits the logical flow in the order it should appear in the code – it indicates a sequential operation, which begins in a loop and then commences with a simple call for a function (i.e., *max*). The loop is described using a quantified term (i.e., *every three*).

On the other end, Novices craft NL utterances unaligned with the program code they describe. They have no prior knowledge of the target program code or programming language, the API being used, or programming concepts in general—making their descriptions simpler using everyday language and diverging from the underlying code elements. When describing complex programs, novices use semantics that implicitly hints at loops or conditions. Furthermore, novices' discourse in the utterances is not constrained to the order of logical control flow structures in the code but to their more intuitive capture in their minds.

To illustrate, examine the following novice description of a complex program: ‘‘*Find me adjacent seats for Shakespeare in the Park, on the first weekend day the weather will be nice*’’. In this example, and depending on the specific API, the expected resulting program might need to contain a loop (i.e., iterating over *the weekend* days) that is contingent on the result of a condition (i.e., *the weather will be nice*). One should notice that the order of the predicates in the description differs from the order of the statement executed in the code. Furthermore, no explicit keywords were used to indicate the condition or the loop. Last, the NL terms do not necessarily align with the actual arguments within the code (i.e., "*weather will be nice*" will be inferred to a range of temperature and other weather features in the code). This creates an additional challenge, that of translating of language used in novices' utterances into the technical intricacies of the target code.

2.3 Why is the Evaluation of NL Programming Challenging?

Code generation models have traditionally been assessed by string-match-based metrics, such as BLEU (Papineni et al., 2002) and CodeBLEU (Ren et al., 2020). However, these methods fall short because they misjudge functionally equivalent solutions to the reference solution in the vast space of programs. As a result, recent work in code synthesis turned to evaluating models through the functional correctness of the generated code. In this line of work, a dataset sample is considered correct if it passes a set of unit tests.

HumanEval (Chen et al., 2021) is a notable step in this direction. It presents a dataset of handwritten unit tests for programming problems, crafted by programmers, which describes simple program execution, like those in programmers' repositories datasets such as Github.

At the same time, the granularity of functional correctness in the HumanEval work is limited to the final output of the generated program. In a complex program comprising multiple steps, we need to provide insights into which parts of the program are correct. Also, obtaining precise evaluation, especially for complex code with numerous edge cases, requires a comprehensive functional test suite. Last, functional correctness metrics are often task-specific. A metric idesigned for one type of task (e.g., generic programming problems) may not be suitable for another type (e.g., generating code accessing a certain API).

To overcome these challenges, in this work we create a set of functional correctness evaluation tests for programs that originate from novice descriptions. The described scenarios are thus accompanied by comprehensive unit tests, encompassing the various edge cases typical of complex programs, to give us a clear and more granular assessment of the generated code.

3 Task and Benchmark

Input The task takes as input (i) a novice-generated natural language utterance, which describes a complex code, and (ii) API specifications that allow for the functional execution of the relevant code. These API specifications serve as a bridge between the high-level complex NL instructions and the low-level code implementation.

Output The output of this task is a program code that satisfies the description given in the NL utterance. The output code is required to be syntactically and functionally correct and compile successfully following the API specifications.

Benchmark Creation In what follows we discuss the steps towards benchmark creation for the task. First, we collect intent-based program descriptions from non-programmers, serving as a window into the intuitive language constructs and phrasing preferred by this group, capturing a diverse range of plain language descriptions involving different logical control flows (Section 4.1). Second, we create a corresponding wide-coverage, hand-crafted test-suite validating the control flow and edge cases in the described systems, which is used for function-based evaluation (Section 4.2). Finally, we create a corresponding code-base solving these utterances, based on a domain-specific API specification which we formulate (Section 5).

4 Data Collection and Curation

Our benchmark creation begins with collecting natural language utterances from novices, in domains that are understandable to them (Section 4.1). Next, we assemble an evaluation dataset that combines these utterances with corresponding test suites to assess the quality of the generated code (Section 4.2). In addition, we synthetically generate NL utterance-code pairs to facilitate the training of models for the task (Section 4.3).

4.1 NL User Requests Collection Interface

The first step of our data collection aims at a scalable collection of natural language instructions from novices that depict complex execution in control flow structures. This collection is achieved via crowd-sourcing. We crowdsourced novice descriptions of complex code from crowd-workers[1] on the Amazon Mechanical Turk platform. We ground the collected NL utterances in a task-oriented dialogue-systems environment. This realm is a critical component of virtual assistants, which are responsible for understanding the user's intents (e.g., *set reminder*, *play music*, etc.). It is user-friendly and requires no coding expertise or familiarity with specific programming languages.

We focused our collection efforts on 9 domains and their inherent intents: clock, events, map, messaging, music, reminders, shopping, smart home, and weather. These domains are intuitive to the novice crowd worker, do not require any proficiency, and can be interleaved to describe the execution of a complex program. The API covers all these domains, yet it remained hidden from the novice crowd workers, preventing biases that may have influenced their natural language descriptions. This highlights the disconnect between the novice's phrasing and the anticipated program.

To stimulate the generation of creative descriptions, the interactive user interface simulates a mobile device. The interface guides the contributors through a series of steps to formulate complex queries, validate the control flow, and submit their utterances. To foster creativity, contributors were randomly presented with (1) a subset of potential domains, (2) a preferred control flow (e.g., *condition* or *sequence*) phrased in an intuitive manner, and (3) in certain cases, a restriction on the use of frequently recurring words (e.g., ‘‘*if*’’, ‘‘*then*’’, ‘‘*tomorrow*’’, ‘‘*tonight*’’, ‘‘*weather*’’, ‘‘*when*’’, etc).

Another tactic employed to foster creativity was providing the task with a single-step non-complex utterance example, randomly selected from the TOP (Gupta et al., 2018) and TOPv2 (Chen et al., 2020) datasets, and requesting the contributors to rephrase the utterance to express a more complex goal, and educating them what complex utterances might be like.

4.2 Human Crafted Test Suites

The second stage of our benchmark creation process focuses on the formulation of hand-written test suites. For evaluation of the code, we collected manually crafted Python test suites containing unit tests that are tailored to check the correctness of the code generated for the NL instructions from the first stage. We randomly selected 150 evaluation targets from the 1,200 collected utterances in the first stage (4.1). Two undergraduate and graduate computer science students, familiar with Python programming and unit testing, spent about 100 hours constructing the tests to carry out the task reliably.

Each annotator coded the functional test with a test scenario snippet, to be seeded with data used as input to the test, and a set of assertion tests accordingly. At runtime, we executed the unit test by combining the test scenario snippet, the generated code, and the assertion tests. Until the unit test execution, the generated code remains concealed from the programmers coding the unit tests, ensuring an impartial evaluation of the code-generating model (see Figure 2).[2]

4.3 Synthesized Training Dataset

We augment our dataset by following the Berant and Liang (2014) approach to generate synthesized utterances. This approach matches NL sentences with logical predicates. First, we sourced a seed lexicon specifying a canonical phrase (e.g., ‘‘*check weather*’’) for each logical predicate (`checkWeather`) in the scope of our domains. This lexicon was drawn from a held-out set, a subset of the larger dataset we collected via crowd-sourcing (section 4.1).

[1] 39 native English speakers with a high approval rate of above 99%.

[2] To prevent leakage of the problems in the evaluation test suites, we archived the plain content of the tests in a gzip file that is not accessible to crawlers or scraping.

```
# test scenario data seeding
data_model = DataModel(reset=True)
data_sender1 = Contact(text='all
advisors␣in␣the␣committee')
data_model.append(data_sender1)
data_sender2 = Contact(text='all
advisors␣in␣the␣committee')
data_model.append(data_sender2)
data_content = Content(text='confirmation')
data_model.append(data_content)
data_content_neg = Content(text='decline')
data_model.append(data_content)
for sender, content in zip([data_sender1, data_sender2], [data_content,
      ↪ data_content_neg]):
    data_model.append(
        Message(
            sender=sender,
            content=content
        )
    )
data_event_name = EventName(text='my␣meeting␣with␣them')
data_model.append(data_event_name)
data_model.append(
    CalendarEntity(
        event_name=data_event_name,
        participants=[data_sender1, data_sender2]
    )
)

# start code block to test
<Generated code embedded here />
# end code block to test

# assertions
test_results = {}

actual = data_model.get_data(CalendarEntity)
expected = [] # expected the program canceled the meeting
entity_assertions(expected, actual, test_results)

assert_test(test_results)
```

Figure 2: An example of the unit test code used for evaluating the utterance *“Check that I received confirmation emails from all advisors in the committee or cancel my meeting with them”*. At test time, we embed and execute the generated code within the test framework. This process allows us to evaluate the functional correctness according to the test scenario.

Second, we define a grammar, that along with the seed lexicon and a mock data generator,[3] can automatically generate a plethora of canonical utterances (*“check the weather in New York City tonight”*) paired with their execution code.

Having defined the predicates and potential arguments, we expand our grammar with naturally occurring phrases manifesting control-flow rules sourced from the held-out set, facilitating the generation of control flows with varying degrees of nesting.

Last, canonical utterances were recursively used to compose complex canonical utterances (e.g., *“check weather tonight and traffic in New York City”*). A generated utterance may not have the elegance of a genuine NL user query, though it still retains the semantics of executed through the code. In contrast to the crowd-sourced user queries, that we allotted for evaluation, the synthesized user queries are merely used as training data to fine-tune large language models.

Finally, this synthetic process is easily scalable. While we focus our synthesized dataset efforts on the specific set of the 9 domains and their inherent intents, adding a domain consists of sourcing a seed lexicon for the new domain, identifying predicates in the domain, and updating the grammar with the new domain grammar rules.

5 The API Specifications

To provide code frameworks that bridge the translation of NL descriptions to their respective code programs, we created an API that generically aligns spans in the user description to code data types and actions. The generated code must correctly utilize the API endpoints and be executable, allowing us to assess its functionality by executing corresponding tests. The API specifications followed the nine domain apps defined for sourcing the prompts (see Section 4.1). Nonetheless, the formulation of the API was designed independently from the process of collecting natural language user utterances in these domains.

The API code comprises multiple classes and is designed to provide a comprehensive and modular framework for interacting with the system's functionalities. Forty-nine interfaces define the different data type entities in our APIs. `Location`, `Contact`, `DateTime`, are examples of such data types. To perform actions with these entities, we expose an additional 11 classes with 34 action methods that are available to be executed. For instance, the `Messages` class defines methods to send a message and find messages received from specific senders, with desired content, or at a specific time.

Users provide natural language descriptions detailing the actions they want to execute and the specifics of each action. The model must interpret descriptions and associate them with the appropriate API classes and methods.

To map from user-specified text spans to domain objects, we assumed the API provides the methods `resolve_from_text` and `resolve_many_from_text`, which discern relevant spans of word sequences in an utterance and correlate them with specific entities. For instance, when a user queries for *“the weather on independence day”* the method `DateTime.resolve_from_text` is invoked with the temporal description spans from the request (*“on independence day”*) and returns a `DateTime` object which is stored in the code. These variables, representing entities, are then used in other API methods to execute the

[3] We used Faker, a Python package that simulates data such as names, addresses, and phone numbers.

```
events = Calendar.resolve_many_from_text("After␣every␣Astros␣game")
for event in events:
    date_time = DateTime.resolve_from_entity(event)
    content = Content.resolve_from_text("check␣the␣traffic")
    Reminders.create_reminder(date_time=date_time, content=content)
```

Figure 3: An example demonstrating methods to resolve entities from text spans and other entities in the user utterance *''After every Astros game remind me to check the traffic''*.

actions requested in the natural language description (e.g., `get_weather_forecast`). It is the API provider's responsibility to provide suitable implementations for these text-resolution functions. In our evaluation suite, we provide suitable mock implementation (see Section 6.3).

In certain cases, an entity needs to be inferred from another entity. For such instances, the API offers extra methods such as `resolve_from_entity` or `resolve_many_from_entity`. For instance, the phrase *''After every Astros game''* refers to a set of events on a public calendar. To infer the `DateTime` entity from each event we iterate on the events and call `resolve_from_entity` on each event (see Figure 3).

6 Evaluation

6.1 Method

An essential aspect of any task is its evaluation methodology and thus we introduce an automated execution-based evaluation measure for the proposed NoviCode task, addressing the challenge we outlined in Section 2.

The evaluation aims to quantify three key factors: (1) The model's efficacy in generating fully executable code, (2) The model's capacity to identify and generate the intended control flows reflected in the natural language utterances, and (3) The success of the model in generating all operations in every execution flow.

Instead of assessing the syntactic correctness of the generated code, we shift the evaluation method towards evaluating functional correctness, wherein a synthesized program is deemed correct if it satisfies a series of unit tests. We build upon this methodology by introducing a collection of human-crafted test suites encompassing a diverse range of test cases, meticulously designed to evaluate the accuracy, robustness, and the generated code logical structure. Thus, alongside the NoviCode task, we release our set of 150 handcrafted evaluation problems.[4]

A human expert manually created each functional test to ensure its accuracy in identifying true positives. To minimize the likelihood of false positives, the expert thoroughly designed multiple test scenarios particularly focusing on the truth or falsity in conditional control flows.

6.2 Metrics

To quantify a model's functional correctness evaluation score, we compute the pass@k (Kulal et al., 2019) score, employing the method used by Chen et al. (2021) and subsequent studies. We generate $n \geq k$ samples per task, count the number of correct samples $c \leq n$ which pass the unit tests, and calculate the unbiased estimator:

$$pass@k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \qquad (1)$$

6.3 Simulated API Implementation

To facilitate the evaluation by executing the functional correctness tests with the generated code, an implementation of the API was required to be in place. In a real-world scenario, a system is required to support the true execution of the API-specified classes and methods. For example, sending emails to selected contacts from the user's address book (e.g., *''mom and dad''*), or retrieving the weather forecast at a specific time and place relative to the user (e.g., *''independence day''* and *''my neighborhood''* respectively). We provide a mock-up implementation that simulates the proposed actions in the API specifications, allows test input data seeding, and supports evaluating state changes of the underlying data model following the invoked methods.

Another aspect of simulating the API is that it expects an extractive span comparison for instantiating data type entities from the NL description. One of the challenges in reliably defining these entities lies in matching the correct span (e.g., *''all advisors in the committee''*) to the expected data dictionary entry (e.g., *''Committee advisors''*). For that, we implemented fuzzy matching over

[4] The dataset and source code are publicly available at `https://github.com/biu-nlp/novicode`.

Control flow	Frequency
Sequences	57.8%
Conditions	31.8%
Loops	26.1%

Table 1: Frequency of the observed control flows found in the crowd-sourced utterances. Multiple control flows can be seen in a single utterance.

content words in the `resolve_*` string comparison methods, ignoring determiners and common prepositions and postpositions. This technique compares the input span with the expected span and produces a BLEU score, accepting a match if it exceeds a specific threshold. We used a BLEU score threshold of 50% (Tran et al., 2019).

7 Data Quality Assurance

NL User Utterances Collection We implemented a multi-step process to maintain the quality of the collected utterances. First, we vetted experienced Amazon Mechanical Turk crowd workers[5] contributing to this task to ensure they were non-programmers and had no coding proficiency or educational background in computer science or related fields. Secondly, we funneled our crowd workers through preliminary qualification tests, to verify they understood the task and its intricacies and could provide a wide range of creative responses, avoiding templated, repeating, and short answers. Last, we manually reviewed each input and tagged it based on control flow structures it exhibited (i.e., loop, sequence, and condition), as shown in Table 1. This process resulted in a collection of 1,200 verified utterances.

Human Crafted Test Suites An expert programmer reviewed each of the 150 selected problems in the evaluation set to ensure the quality of the functional correctness test suites in the benchmark. First, by manually providing code for every prompt, the expert confirmed that every problem had a corresponding code that correctly used the API classes and methods. Secondly, to ensure the test is executable and can validate the evaluated code, the expert programmer integrated the manually provided code into the functional correctness tests assigned to the problem and executed it to either pass the test suites upon a correct code or fail otherwise. This review process resulted in a human performance score of 100% in the functional correctness evaluation tests. This ensured that all tests were executable upon providing a correct code for the prompt and that the test formulation successfully checked multiple test scenarios.

```
[ Module
    [ events = Calendar.resolve_many_from_text('After␣every␣Astros␣game')
          ↪  ]
    [ For
        [ test
            [ iter
                [ events ]
            ]
            [ Name
                [ event ]
            ]
        ]
        [ body
            [ date_time = DateTime.resolve_from_entity(event) ]
            [ content = Content.resolve_from_text('check␣the␣traffic') ]
            [ Reminders.create_reminder(date_time=date_time, content=
                  ↪ content) ]
        ]
    ]
]
```

Figure 4: An example of the *cAST* in a bracket notation.

8 Better Code Generation through Intermediary Representation

Instead of translating natural language utterances directly to code, we propose to map the natural language to an intermediary structure that better encapsulates the control flow elements of the language. By explicitly representing the structure of the target complex program through a hierarchical structure, one that expresses the compositional control flows expressed in the code, we hope to improve the compositional generalization thus improving code generation accuracy relative to standard end-to-end text-to-code methods (Herzig and Berant, 2020).

Formally, given a natural language utterance x, our goal is to convert x into a surface code y. Instead of mapping directly to y, we learn to translate x to an intermediary logical form z, which can be deterministically converted to y. We use the text-to-code models to perform the transformation $x \to z$, and then use a deterministic transformation $z \to y$ to obtain y. Specifically, z is based on a *linearized compact AST* (cAST) which is derived from the abstract syntax tree (AST) of the code, and is linearized into text using a bracketted notation (Figure 4). The AST is a tree structure representing the structure of the program. Using the AST allows the model to

[5] Native English speakers with a high approval rate (99%) and significant experience (over 5,000 completed HITs) on the Amazon Mechanical Turk platform.

focus more on the logic and flow of the complex program rather than mere syntax (e.g., keywords, symbols, indentation), reducing the complexity of the output space. On top of that, we define a compact AST (cAST) that only retains the control flow logic structures, such as sequences, conditions, and loops, in a tree form. To create compact ASTs, we revert basic operations such as variable assignments or function calls to their fundamental code syntax. These elements are transformed from their AST hierarchical representation to appear as leaves in the tree's code form.

To translate the linearized cAST z back to code y, we parse the string into its cAST tree form and expand it back to the AST form using a tool we provide. The ASTs are translated into actual code using the official Python package for this (`ast`).

We note that previous work (Yin and Neubig, 2017; Yin and Neubig, 2018) also utilized ASTs to synthesize code better. Yet, these works model code generation as a series of classification problems of grammar rules required to reconstruct the AST. Our method uses the hierarchical tree form as a vehicle to directly map texts to hierarchical code structures.

9 Experiments

We set out to evaluate how existing models cope with the NoviCode task. Moreover, we aim to assess whether our representation-based learning approach (Section 8) outperforms a simple end-to-end Text-to-Code baseline.

To do so, we evaluated decoder-only generative large language models (LLMs) using in-context learning scenarios. We also tested pre-trained encoder-decoder LLMs that are fine-tuned with the synthesized trainset. We tested both a simple end-to-end scenario (denoted *base*) and one where we map to the hierarchical compact AST (denoted *cAST*), as defined above.

Data and Evaluation We synthesize a training dataset of 40K samples as discussed in Section 4.3. Every sample in our dataset is a tuple containing an NL user request, a matching Python code, and the cAST representing the code (Section 4.1). We evaluate models on this task using expert-crafted test suites for the functional correction of the generated code (Section 4.2).

Metrics To reliably estimate the functional correctness of a model, we generated 200 samples for each prompt ($n = 200$) and calculated the mean $pass@1$ and $pass@10$ scores.

In-context Learning. We conducted tests on OpenAI's GPT-3.5-Turbo[6], and GPT-4-Turbo[7] models using in-context learning (prompting) (Brown et al., 2020). We also experimented with open source LLMs including Meta's CodeLlama,[8] Mistra,l[9] and DeepSeek Coder.[10] Our experimental setup used in-context prompts that included the full API specifications and multiple examples of few-shot prompts randomly selected from the synthetically generated dataset. The examples count was capped at 11, aligning with the smallest context window of our models, which equals 16,385 tokens. This type of in-context prompting was feasible only with models that have a higher token limit for context windows, as the API code itself contained 15,397 tokens (including the Python Docstrings).

Fine-tuned Models. We further assess the NoviCode task by fine-tuning models that were previously shown to successfully perform program-synthesis tasks. These models include T5 (with 220M parameters) (Raffel et al., 2019), CodeT5 (220M) (Wang et al., 2021), and CodeT5+ (220M) (Wang et al., 2023a). Each model was fine-tuned using input-output tuples tailored to the specific experimental setups, utilizing data from the synthesized training dataset. The dataset of synthesized samples was split into train (80%), validation (10%), and test (10%) sets for the fine-tuning phase. All models were configured with a maximum input and output length of 512 tokens each. A learning rate of 5e-5 was employed, alongside a constant warm-up with step inverse decay, and the warm-up steps were capped at 1,000. We utilized the AdamW optimizer (Loshchilov and Hutter, 2019), and the experiments were conducted with a batch size of 8 and were set to run for a maximum of 20 epochs. However, the execution was halted if no improvement was observed for three consecutive epochs. An A100 GPU machine was employed for these experiments.

[6] gpt-3.5-turbo-1106.

[7] gpt-4-0125-preview.

[8] CodeLlama-7b-Instruct-hf.

[9] Mistral-7B-Instruct-v0.2.

[10] deepseek-coder-33b-instruct.

Model Name	Setup	pass@*1*	pass@*10*
GPT-4-Turbo	base	33.8 ± 0.1	51.6 ± 0.4
	cAST	**39.0 ± 0.3**	**55.6 ± 0.8**
GPT-3.5-Turbo	base	10.7 ± 0.0	29.0 ± 0.0
	cAST	10.3 ± 0.1	31.3 ± 0.2
CodeLlama-7B	base	1.5 ± 0.1	10.7 ± 2.1
	cAST	3.3 ± 0.0	17.2 ± 1.5
Mistral-7B	base	0.3 ± 0.0	2.1 ± 0.3
	cAST	1.2 ± 0.3	6.6 ± 1.2
DeepSeek-Coder-33B	base	8.8 ± 0.3	27.8 ± 0.2
	cAST	7.0 ± 0.4	26.2 ± 0.3

Table 2: Comparing LLMs with different input-output setups. We find that representing code in a structural form outperforms the basic text-to-code approach. Results are indicated by mean and std dev.

10 Results and Analysis

10.1 Results

Our experiments revealed that in most model architectures we tested, having an output of a compacted AST (cAST) structural form showed increased performance compared to the text-to-code method. The best results were obtained with the GPT-4-Turbo model, to which we supplied an in-context prompt containing the NL descriptions and expecting a *cAST* to be converted to code.

Analyzing the results, we observed that the *cAST* form better generated code that exhibited conditions and loops in control flows, forms which are more explicit in the ASTs. With sequences of operations, the two setups showed similar success. In cases where multiple control flows were present in the same utterance, the *cAST* output form also excelled compared to the text-to-code setup.

In-Context Learning We assessed this task on different commercial and open-source LLMs. Our assessment evaluated the standard text-to-code approach against our proposed structural representation code setup. The results are shown in Table 2. Our proposed approach using the code representation method showed the best result. This performance in an in-context learning format shows the promise of using this intermediate form to represent code in general.

Fine-tuned Models We turned to assess this task on three fine-tuned baseline models. The results are presented in Table 3. In the pass@k scores, the hierarchical code representation method (using cAST) outperformed all text-to-code strategies in all model architectures we tested.

Model Name	Setup	pass@*1*	pass@*10*
T5	base	0.0 ±0.0	0.0 ± 0.0
	cAST	9.7 ± 1.1	**19.6 ± 2.3**
CodeT5	base	8.1 ± 0.7	14.8 ± 1.6
	cAST	9.1 ± 0.5	18.8 ± 1.1
CodeT5+	base	11.3 ± 1.4	19.4 ± 2.8
	cAST	**11.7 ± 0.8**	18.8 ± 2.1

Table 3: Comparison of fine-tuned models' performance. Our proposed hierarchical code representation form achieved the highest performance scores across all models. Results are indicated by mean and std dev.

Training Size Learning Curve To understand the minimal train-set size required for fine-tune a model using our approach, we evaluated the best-performing CodeT5+ model with the proposed structural code representation approach, by incrementally increasing the train set used for fine-tuning. We observe that models trained on smaller datasets (fewer than 5,000 samples) yielded notably lower scores (Figure 5). As the train set expanded, the performance improved consistently and then reached a plateau. This implies manually sourcing a sufficiently large dataset would be difficult because of the substantial volume needed.

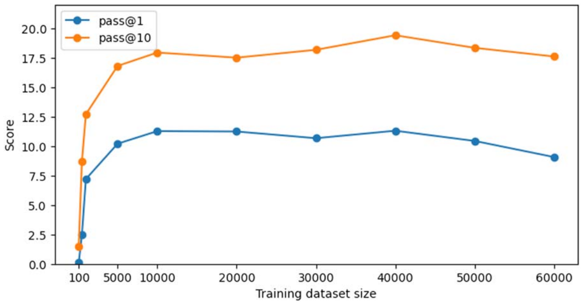


Figure 5: Comparison of fine-tuned models' performance for varying dataset sizes. The success of our proposed approach for this task is limited by the reliance on synthesized examples for fine-tuning models.

Control Flow	Success %
Sequences	68%
Conditions	46%
Loops	16%

Table 4: Model success in recovering complex descriptions to program code with control flows. The more explicit control flow structures appear in the NL description – the better the model can embed it in code.

10.2 Error Analysis

We conducted an in-depth error analysis of the results of the best-performing model (GPT-4-Turbo with a 128K token context window using the *cAST* in-context setup) to identify key areas for improvement. We based our analysis on a randomly selected subset of 150 outputs generated by the model.

Observing the model's efficacy in generating code programs with accurate control flows (Table 4), our analysis revealed that control flows that are more explicitly expressed in the NL description, such as *sequences* and *conditions* (e.g., conditions that were described using the function word *if*) were more accurately implemented in code. Loops, in contrast,often require implicit deduction from nuances like quantifiers (e.g., *every day*), noun phrases with conjunctions (e.g., *mom and dad*), or specific semantic terms (e.g., *my book club group*), and are harder to infer.

The model displays distinct patterns of errors that provide insight into its limitations and potential areas for improvement. We broadly classified errors into three categories: (i) Syntactic errors: malformed cAST (ii) Logical errors: Compiled code with runtime errors/exceptions and (iii) Semantic errors: Successful run with a wrong outcome. We hereby detail them in turn.

Syntactic Errors. These were rare, seen in only 7.2% of the cases, but critical, as the model generated code with incorrect syntax, such as malformed AST node labels or mismatched brackets. In that case, we were not able to reconstruct the final Python code from the code intermediate representation as cAST. A related case is where the model generated an (too-) lengthy output and reached its maximum token limits, neglecting the correct closure of brackets in the linearized tree.

Logical Errors. In 26% of the cases, we noted runtime errors where, although we successfully transformed the intermediate representation into a program code, it led to exceptions upon execution. Examples of these errors include referencing undefined variables, incorrectly calling functions with unexpected arguments, or illegal operations on data types, like attempting to iterate over objects that are not iterable.

Semantic Errors. These were the most common (53%), where the model succeeded in executing the generated code but failed to implement correctly the NL description, leading to a completely different output than expected. Semantic errors were detected following assertion failures as part of the functional correctness evaluation (Section 4.2). This error type is manifested in outputs containing irrelevant or incorrect data not present in the NL input, poor recall in identifying necessary arguments due to data type errors or omissions, and problems with variable reusability, especially in cases where antecedents and anaphors are distanced in the NL descriptions.

11 Related Work

Previous text-to-code datasets were delivered to facilitate the training and testing of the code generation capacity of contemporary models. Notable datasets include the CoNaLa Dataset (Yin et al., 2018), which contains programming questions from Stack Overflow along with code solutions, the CONCODE dataset (Iyer et al., 2018), the CodeSearchNet Corpus (Husain et al., 2019), and the CodeXGLUE dataset (Lu et al., 2021), constructed using code-comment pairs from GitHub across numerous domains in various programming languages. The majority of these include very technical jargon and relatively short statements, in contrast with our novice users language and lengthier intent expressions (Section 2.1).

Identifying instructions in Task-Oriented datasets can also be seen as equivalent to interpreting description as executions. Notable Task-oriented datasets are TOP (Gupta et al., 2018) and TOPv2 (Chen et al., 2020) datasets. However, these datasets fail to provide high-level abstractions that elicit complex code with sequences and control flow structures (e.g., loops and conditions).

On the front of evaluating text-to-code generation, automatic metrics for code generation evaluation initially adopted techniques similar to those used in machine translation. BLEU (Papineni et al., 2002) has been extensively used for evaluating code generation. However, its limitations have been increasingly recognized, including the disregard for semantic correctness and functionality. CodeBLEU (Ren et al., 2020) enhances BLEU by incorporating crucial code-related features such as syntactic and semantic similarity, data flow, and variable misuse. Despite its improvements, CodeBLEU still relies on a reference implementation, which may hinder its efficacy in cases where multiple correct solutions exist.

The HumanEval dataset (Chen et al., 2021) presented a recent and significant effort in this direction, where functional correctness tests evaluate the generated code. The evaluation tests in HumanEval present basic programming problems. Austin et al. (2021) presents a similar approach for evaluating correctness. Additional code generation benchmarks extended this approach using more challenging programming problems, which require an understanding of algorithms (Li et al., 2022; Hendrycks et al., 2021), or usage of external and advanced Python packages (Lai et al., 2022; Wang et al., 2023b).

Our benchmark presents code-generation tasks that concentrate on everyday tasks for laypeople, such as scheduling meetings in a calendar or checking weather forecasts. Furthermore, to support and extend the complexity, our benchmark introduces a private and unseen API, in contrast to using standard Python packages.

Other benchmarks challenged models by providing class-level Python code-generation tasks (Du et al., 2023). The challenge introduced in this paper is orthogonal to this challenge and does not generate class-level code.

Functional tests on the generated code are prone to false positives or true negatives upon low coverage of test edge cases. Some works (Liu et al., 2023) mitigate this using an automatic test input generation engine. In contrast with this approach, our evaluation method validates the data model affected by the generated code and not the generated code itself.

Few code generation benchmarks have also concentrated on the population composing the NL prompts in the benchmarks for code LLMs, specifically targeting non-expert beginner programmers (Babe et al., 2023). In contrast, the prompts in our benchmark were crafted by novice, non-programmers lacking any programming proficiency.

Last, other corpora capture how non-programmers express if-then clauses (Quirk et al., 2015). Yet, the tasks in this benchmark are expressed in a highly structured and noisy language and exhibit only a single control flow paradigm – conditionals.

12 Conclusion

In this paper we introduce NoviCode, a novel NL programming task of generating executable complex programs with control flow structures from novice NL user requests and API specifications that the program should comply with. As an integral part of our task, we propose an evaluation framework to assess model efficacy on this task, based on functional execution and denotation rather than on the code form. Building upon this task, we propose a novel representation that explicitly reflects the hierarchical structure of code, which outperforms all baseline models, open source (e.g., CodeT5+) or closed source (e.g., GPT-4-Turbo) models, in the evaluation tests. Finally, our analysis suggests that while generating working code based on language is a feasible task, ours is still a challenging one. Yet NoviCode constitutes a promising approach towards true *natural language programming*—where humans program in their native tongues—encouraging future research and development of this domain.

Limitations

Evaluation Test Size. Creating unit test suites for evaluating code generation models on this task is both time-intensive and requires Python testing skills and familiarity with specific APIs. This process significantly contributed to the development time of our evaluation dataset. On average, it took an experienced programmer about 9 minutes to create each unit test. Due to limited resources, we could only prepare tests for 150 out of the 1200 collected user utterances. Expanding the dataset for a more extensive evaluation is a key goal for future work.

Acknowledgments

We gratefully acknowledge the contribution of Tamar Gur for her invaluable assistance in this work. Additionally, we extend our gratitude to Royi Lachmy, Avshalom Manevich, and Shira Kritchman for their helpful comments and discussions. We also thank the anonymous reviewers and the action editor for their valuable suggestions.

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program, grant agreement no. 677352 (NLPRO), and grant agreement no. 802774 (iEXTRACT).

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2023. Studenteval: A benchmark of student-written prompts for large language models of code.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1415–1425, Baltimore, Maryland. Association for Computational Linguistics. https://doi.org/10.3115/v1/P14-1133

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. *CoRR*, abs/2107.03374.

Xilun Chen, Asish Ghoshal, Yashar Mehdad, Luke Zettlemoyer, and Sonal Gupta. 2020. Low-resource domain adaptation for compositional task-oriented semantic parsing. *CoRR*, abs/2010.03546. https://doi.org/10.18653/v1/2020.emnlp-main.413

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation.

Sonal Gupta, Rushin Shah, Mrinal Mohit, Anuj Kumar, and Mike Lewis. 2018. Semantic parsing for task oriented dialog using hierarchical representations. *CoRR*, abs/1810.07942. https://doi.org/10.18653/v1/D18-1300

David Harel. 2008. Can programming be liberated, period? *Computer*, 41(1):28–37. https://doi.org/10.1109/MC.2008.10

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps.

Jonathan Herzig and Jonathan Berant. 2020. Span-based semantic parsing for compositional generalization. *CoRR*, abs/2009.06040.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *CoRR*, abs/1808.09588. https://doi.org/10.18653/v1/D18-1192

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S. Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097. https://doi.org/10.1126/science.abq1158, PubMed: 36480631

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? Rigorous evaluation of large language models for code generation.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training LLMs on programming and natural languages. *ICLR*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics. https://doi.org/10.3115/1073083.1073135

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. 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 878–888, Beijing, China. Association for Computational Linguistics. https://doi.org/10.3115/v1/P15-1085

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: A method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. https://doi.org/10.1109/ICPC.2019.00034

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023a. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023b. Execution-based evaluation for open-domain code generation. https://doi.org/10.18653/v1/2023.findings-emnlp.89

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *CoRR*, abs/1805.08949. https://doi.org/10.1145/3196398.3196408

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

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation.

A Appendix: Code Intermediate Representation Scheme

The intermediate form for representing the code used Abstract Syntax Tree (AST) formulation as its baseline.

Let T be a tree, and N be the set of all nodes in T.

Let $n1 \in N$ be a node in T.

We express that the label of a node $n1$ as in the set of allowed labels L as follows:

$$\text{lab}(n1) \in L$$

Let T_{n_1} denote a subtree with n_1 as its root. Given that the terminals of T_{n_1} are $t_1, t_2, \ldots, t_n$ in order, the label of T_{n_1} can be expressed as the concatenation of labels of the terminals:

$$\text{lab}(T_{n_1}) = \text{lab}(t_1)||\text{lab}(t_2)||\ldots||\text{lab}(t_n)$$

We say that a subtree T_{n_1} starts with a string s if and only if:

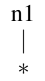
$$s \sqsupset \text{lab}(T_{n_1})$$

Compactization Rule

Let $L = \{'Assign',' AugAssign',' AnnAssign', 'Call',' Expr'\}$. And let the label of a node $n1$ in the set of allowed labels L as follows:

$$\text{lab}(n1) \in L$$

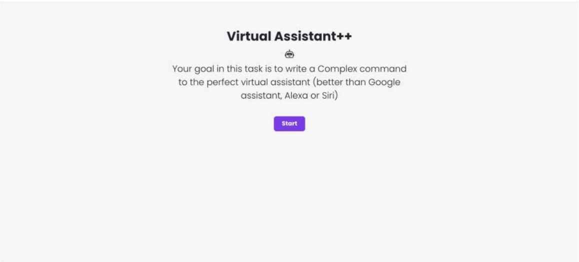
We say that a subtree T_{n_1} should be replaced with a node with the label of the unparsed value of that node.



Translates as follows:

$$unparse(n1)$$

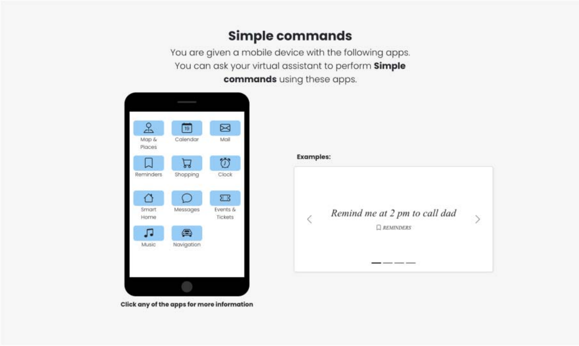
B NL User Utterance Elicitation Interface



Virtual Assistant++

Your goal in this task is to write a Complex command to the perfect virtual assistant (better than Google assistant, Alexa or Siri)

Figure 6: Task introduction screen.



Simple commands

You are given a mobile device with the following apps. You can ask your virtual assistant to perform **Simple commands** using these apps.

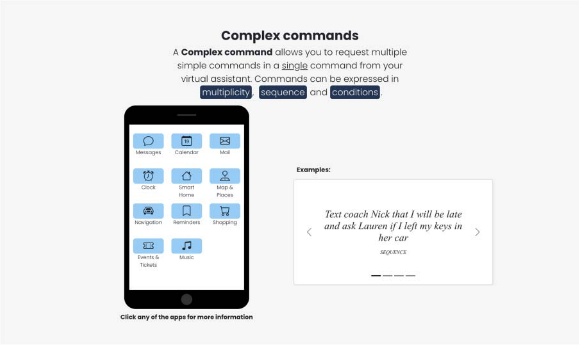
Examples:

Remind me at 2 pm to call dad

REMINDERS

Click any of the apps for more information

Figure 7: Educating crowd workers on simple instructions.



Complex commands

A **Complex command** allows you to request multiple simple commands in a single command from your virtual assistant. Commands can be expressed in multiplicity, sequence and conditions.

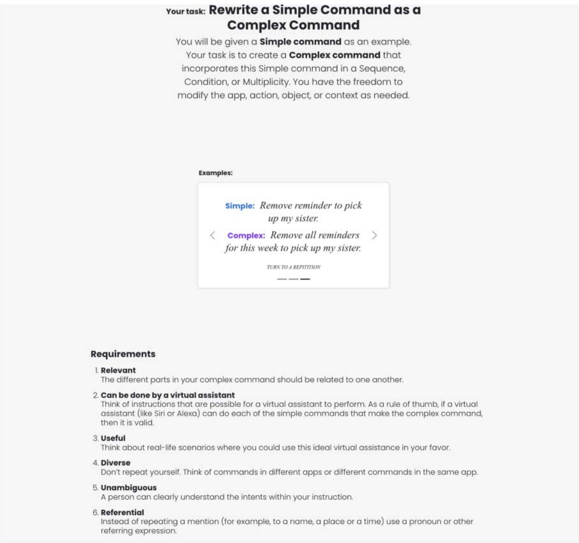
Examples:

Text coach Nick that I will be late and ask Lauren if I left my keys in her car

SEQUENCE

Click any of the apps for more information

Figure 8: Educating crowd workers on complex instructions.



Your task: **Rewrite a Simple Command as a Complex Command**

You will be given a **Simple command** as an example. Your task is to create a **Complex command** that incorporates this Simple command in a Sequence, Condition, or Multiplicity. You have the freedom to modify the app, action, object, or context as needed.

Examples:

Simple: *Remove reminder to pick up my sister.*

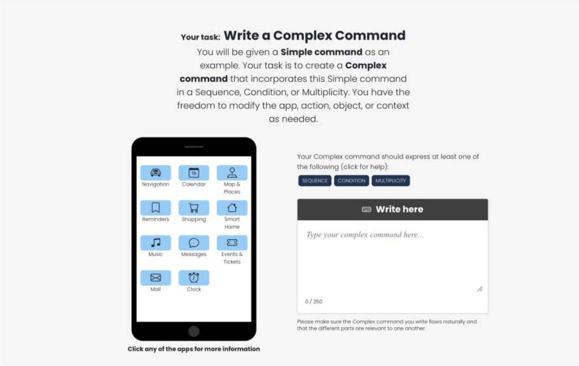
Complex: *Remove all reminders for this week to pick up my sister.*

TURN TO A REPITITION

Requirements

1. **Relevant**
   The different parts in your complex command should be related to one another.
2. **Can be done by a virtual assistant**
   Think of instructions that are possible for a virtual assistant to perform. As a rule of thumb, if a virtual assistant (like Siri or Alexa) can do each of the simple commands that make the complex command, then it is valid.
3. **Useful**
   Think about real-life scenarios where you could use this ideal virtual assistance in your favor.
4. **Diverse**
   Don't repeat yourself. Think of commands in different apps or different commands in the same app.
5. **Unambiguous**
   A person can clearly understand the intents within your instruction.
6. **Referential**
   Instead of repeating a mention (for example, to a name, a place or a time) use a pronoun or other referring expression.

Figure 9: Goals and instructions.



Your task: **Write a Complex Command**

You will be given a **Simple command** as an example. Your task is to create a **Complex command** that incorporates this Simple command in a Sequence, Condition, or Multiplicity. You have the freedom to modify the app, action, object, or context as needed.

Your Complex command should express at least one of the following (click for help):

SEQUENCE CONDITION MULTIPLICITY

Write here

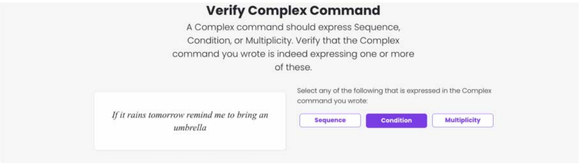
Type your complex command here...

0 / 250

Please make sure the Complex command you write flows naturally and that the different parts are relevant to one another.

Click any of the apps for more information

Figure 10: Task input form.



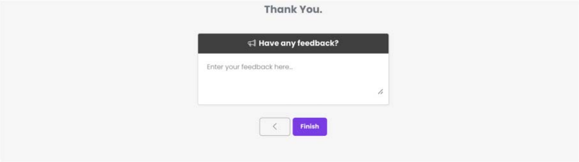
Verify Complex Command

A Complex command should express Sequence, Condition, or Multiplicity. Verify that the Complex command you wrote is indeed expressing one or more of these.

If it rains tomorrow remind me to bring an umbrella

Select any of the following that is expressed in the Complex command you wrote:

Figure 11: Response verification.



Thank You.

Have any feedback?

Enter your feedback here...

Figure 12: Feedback screen.