HISyn: Human Learning-Inspired Natural Language Programming

Zifan Nan
North Carolina State University
Raleigh, North Carolina, USA
znan@ncsu.edu

Hui Guan
University of Massachusetts Amherst
Amherst, Massachusetts, USA
huiguan@cs.umass.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

ABSTRACT

Natural Language (NL) programming automatically synthesizes code based on inputs expressed in natural language. It has recently received lots of growing interest. Recent solutions however all require many labeled training examples for their *data-driven* nature. This paper proposes an *NLU-driven* approach, a new approach inspired by how humans learn programming. It centers around Natural Language Understanding and draws on a novel *graph-based mapping algorithm*, foregoing the need of large numbers of labeled examples. The resulting NL programming framework, HISyn, using no training examples, gives synthesis accuracy comparable to those by data-driven methods trained on hundreds of training numbers. HISyn meanwhile demonstrates advantages in interpretability, error diagnosis support, and cross-domain extensibility.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Source code generation**.

KEYWORDS

Program synthesis, natural language programming

ACM Reference Format:
Zifan Nan, Hui Guan, and Xipeng Shen. 2020. HISyn: Human Learning-Inspired Natural Language Programming. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409673

1 INTRODUCTION

Recent years have witnessed a growing interest in Natural Language (NL) programming, where, code synthesizer automatically produces programming code based on NL input from users. It is especially appealing in cases where such an intuitive interface offers conveniences to general users (e.g., IoT [40], Smarthome [25]), and cases where there are many domain-specific APIs difficult for a programmer to memorize (e.g., Python libraries [2]).

Existing approaches to NL programming fall into two classes: *data-driven* and *rule-driven* approaches. The former features the reliance on many labeled input-code pairs as training data to build up some statistical models; the latter depends on some predefined domain-specific rules. The *rule-based* approach had shown some success in the early stage of the field development (e.g., Smartsynth [25]), but have gradually lost attractions due to the lack of robustness and the difficulties in generalizing across domains. The *data-driven* approach has dominated recent efforts, represented by the adoption of deep learning to map NL queries to code via various neural networks (e.g., [2, 14, 29, 38, 39]). Although this approach has shown more promise than the previous *rule-driven* approach, its requirement of large numbers of labeled examples hinders its adoptions, especially for domains where labeled examples are scarce. Although recent proposals show the possibility of generating examples for a certain domain [3], it is yet unclear how well these methods can generate truly representative examples in complex domains.

In this work, we propose *NLU-driven approach*, an approach driven by *natural language understanding (NLU)*. It is inspired by how humans code. Rather than going through thousands of examples as a *data-driven approach* does, a programmer can start coding after she reads through the documentation of the language or API of interest. She may check a few examples, but the number is far less than what a *data-driven approach* usually needs. The key is in understanding the language or API, the central feature NLU-driven NL programming builds on.

More specifically, the NLU-driven approach features (1) deep processing of programmers' intentions and API documents written in natural languages via NLU, and (2) the leverage of the deep understanding rather than training examples for code synthesis.

Compared to data-driven approaches, the NLU-driven approach has three appealing properties. First, by avoiding the need for a large number of labeled examples, it saves users the (often heavy) burden in collecting/generating examples, and makes NL programming possible for domains where labeled examples are scarce. Second, built on understanding rather than data, it avoids the bias that training data examples bring to data-driven methods. Finally, as a "white-box" approach, its better interpretability makes the diagnosis of synthesis errors easier to do.

Our exploration leads to *HISyn* (for "human learning inspired synthesizer"), the first NLU-driven NL programming framework. HISyn is equipped with several distinctive features.

(1) Deep NL understanding for code synthesis. A deep natural language understanding is the key to human learning-inspired code synthesis. Although NLP has been used in software maintenance [1, 4, 10, 12, 15, 21, 36, 50, 51, 53], its usage in NL programming is still

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409673

preliminary. Unlike prior synthesis studies that use shallow NLP just for assistance, HISyn takes modern NLP as the first-order tool. With it, HISyn automatically builds up the intermediate representation of input queries, and the knowledge base of the programming APIs, preparing the foundation for synthesis to work on. It uses *NL dependence analysis* to capture the deep relations among the different parts of the input query, and WordNet (a lexical database of English with 117,000 synsets) to capture the semantic associations of English words.

(2) Framework architecture design for cross-domain extensibility. Domain differences are inherent to natural language programming. Different domains have different terminologies, API definitions, grammar, query patterns, and so on. Previous non—data-driven studies are domain-specific; rules and elements specific to the target domain are tightly integrated with the synthesizers, making them difficult to port to other domains. HISyn strives to ensure cross-domain extensibility in design. Drawing on inspirations on how modern compilers deal with domain/language varieties, it creates architecture with the front end producing a unified intermediate representation (*NL dependence graph*) for an arbitrary domain, on which, the back end operates to generate the code in the target API. Neither the front end nor the back end requires changes across domains. The HISyn design encapsulates domain-specific elements into separate modules equipped with an easy-to-use interface. For a new domain, the developer only needs to use the interfaces to extend those domain-specific modules; no changes are needed for the HISyn framework.

(3) Graph-based algorithm for mapping. Based on the intermediate representation, HISyn employs *grammar graph-based translation* to generate code to materialize a user's intention in the target programming APIs. The novel algorithm first annotates each node in the IR with candidate APIs and then uses path finding on the reverse API grammar graph to identify the appropriate APIs, their order, and assemble them into the final code. The algorithm bridges the gap between the user's intention and the APIs by leveraging both the semantic connections at the natural language level and the syntactical constraints at the API grammar level.

We evaluate HISyn on three domains, the domain of Text Editing, the domain of air travel queries, and the domain of program source code analysis. Although combing with a few examples could potentially help, to examine the potential of pure NLU-based approach, HISyn is designed to use no examples. Our experimental results show that without any training examples, HISyn can produce code as accurately as those by a representative *data-driven* NL programming framework trained on hundreds of examples. The study validates the cross-domain portability of the core of HISyn, and demonstrates the large potential of the NLU-driven approach for NL programming while saving the burden of collecting large numbers of training examples.

2 BACKGROUND

HISyn employs standard NL processing techniques, such as *Tokenization*, which splits a piece of text into *tokens* (i.e., words), *POS Tagging*, which labels tokens (words) with their Part-Of-Speech (POS) tags, *Lemmatization*, which reduces a word to its basic form called *lemma*, and *Named Entity Recognition (NER)* recognizes named entities or pre-defined categories such as person names, organizations, quantities, and locations.

HISyn, in addition, heavily leverages *dependency parsing*, which goes a deeper level than those listed NLP techniques, analyzing the *dependency relations* between tokens in a sentence, and outputs a *dependency graph*. A dependency relation is composed of a subordinate word (called *dependent*), a word on which it depends (called *governor*), and an asymmetrical grammatical relation between the two words (called *dependency type*). Figure 1 shows the dependency parsing result for an example sentence generated by the Stanford CoreNLP dependency parser[1] (along with the code to synthesize). A dependency relation is marked as an arrow pointing from a governor to a dependent and is labeled with the dependency type. The arrow from "flight" to "cheapest" with the label "amod" indicates that "cheapest" is an adjective modifier of "flight". All the dependency relations form a directed graph, which is called dependency graph.

3 OVERALL FRAMEWORK

The framework design of HISyn is shown in Figure 2. It has three main components: (1) a domain knowledge constructor that processes the domain knowledge to assist code synthesis; (2) a front end that transforms an NL-based query to a dependency graph which serves as the basis for the intermediate representation that the back end works on; (3) a back end that employees *grammar-graph–based translation* to generate code based on the IR.

The domain knowledge constructor takes two files as inputs: (1) a document that contains all the API and their descriptions; (2) a grammar file that contains the context-free grammar written in Backus-Naur form (BNF). The constructor parses the input files and generates two outputs: (1) an API knowledge base for semantic mapping between words in NL-based queries and APIs; (2) a grammar graph that defines the search space for code generation. A formal definition of grammar graph will be introduced in Section 4.4.2.

The front end takes an NL-based query and applies light regulation first to avoid term confusions for words that have domain-specific meaning. It then uses multiple NLP techniques including POS tagging, Lemmatization, NER, and dependency parsing to produce a dependency graph as an intermediate representation (IR). HISyn prunes non-essential words (called *function words*) from the IR based on the POS tag and the dependency relations of each word. Figure 3(a) shows the pruned dependency graph of Figure 1. This pruned IR is fed to the back end for code generation.

The back end then employs a novel synthesizing algorithm called *grammar graph-based translation* to generate code according to the pruned IR. *Grammar graph-based translation* first maps each node in the IR to a set of APIs based on the lemma and synonyms of words in each API's description. The APIs corresponding to each node are called *candidate APIs*. One node can be mapped to several candidate APIs or no candidate API at all. In the case where multiple nodes have a mapping to the same candidate API, HISyn uses a *longest match scheme* to group the nodes as a cluster and selects the candidate APIs for that cluster instead of each node in that cluster. Grammar graph-based translation then translates the IR annotated

[1] http://corenlp.run/

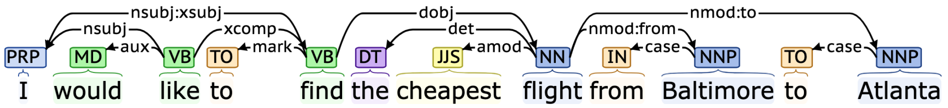


Figure 1: Dependency parsing result from StanfordCoreNLP [32]. The corresponding code is shown in Table 1

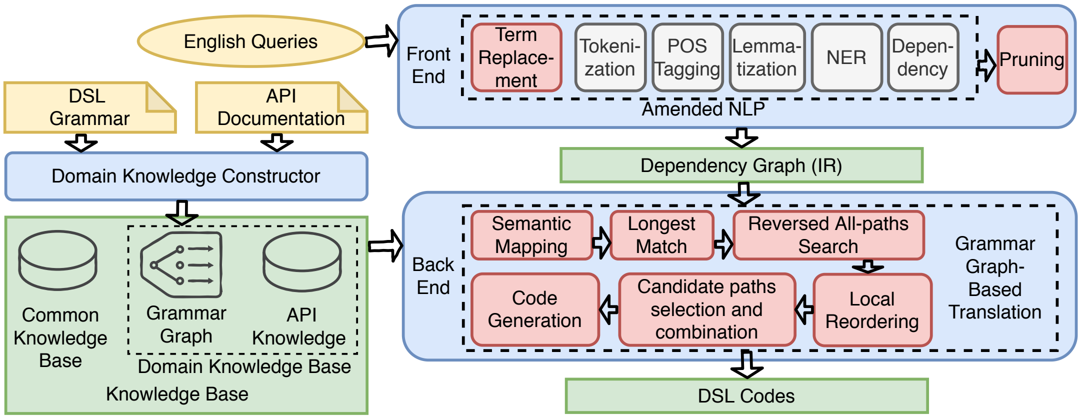


Figure 2: Framework Design.

with candidate APIs to DSL codes based on the grammar graph generated by the domain knowledge constructor.

HISyn features a modular design. By separating domain-specific modules from the core, it simplifies applications to a new domain. We next explain each of the main components of HISyn.

4 NLP AND IR

This section first introduces the domain knowledge constructor and the front end in the framework and then gives formal definitions of the dependency graph and grammar graph, which are two core data structures in HISyn.

4.1 Domain Knowledge Constructor

Domain knowledge constructor builds the common knowledge base and the domain knowledge base.

The common knowledge base stores the information that could be used in any domain. It contains three components: WordNet [34] synonym list, Named Entities, and preposition dictionary. The WordNet synonym list is used to help the Semantic Mapping step in the back end. The named entities (NE) are the labels for real-world objects, such as January's NE is Month, Baltimore's NE is City. The preposition dictionary is a dictionary we create for mapping a preposition to its semantic related words. For example, the semantic related words of preposition "from" include "start", "source", "origin".

The domain knowledge base stores domain-specific knowledge. Every domain will have its own knowledge base. It contains a grammar graph (defined in section 4.4.2) and an API knowledge base. The API knowledge base is generated by parsing the API documentation and contains each API' name, input/output types, and its natural language description. Both API documentation and grammar are stored in separate text files. API documentation is stored in plain text with labeled content (e.g., Name: forStmt). The grammar should be in Backus–Naur form [35]. Before synthesizing the queries, HISyn will parse the text files and construct the domain knowledge base.

4.2 Query Processing via Amended NLP

For a given NL query, the NLP engine goes through the following steps: tokenizing, lemmatization, POS tagging, named entity recognition (NER), and dependency parsing. HISyn then represents the result as a dependency graph, which is used as the basis for the intermediate representation (IR).

One special complexity for code synthesis is terminology confusion. In a specific domain, some terms have a special meaning that confuses the standard NLP parser. For instance, 'for statement' in ASTMatcher domain is a term indicating a type of statement, i.e. for loops in programming languages. However, the NLP parser will take 'for' as a preposition, and provide a parsing result that far away from its meaning in the original query.

We address this issue by letting users add light regulations to the input queries. Specifically, HISyn requires all of the domain-specific terms to be put inside a pair of punctuation marks (double angle brackets as default). For example, "find for statement" should be written as "find <for statement>". With the special words explicitly marked, HISyn can easily replace them with some common terms (mostly noun terms), and replace them back after NLP steps. Users can specify their signs if angle brackets are terms of the target domain.

4.3 Pruning

In natural language, some words in a sentence are *function words* that express grammatical relationships among other words with little lexical meaning. These words usually have no mapping to any APIs. Thus, inside the dependency graph, the edges related to these words are *trivial edges.* In this stage, we prune these trivial edges with function words based on the dependency relation of the edges and the POS tags of the tokens.

The default prunable dependency relation include the following labels: (1) 'det', the determiners (a, an, the, some, etc.); (2) 'case', used for prepositions (In StanfordCoreNLP enhanced++ dependency parser, prepositions is labeled inside 'nmod', thus prunable.); (3) 'mark', the word introducing a clause subordinate to another clause; (4) 'ref', the relative word introducing the relative clause; (5)'aux', a function word associated with a verbal predicate that expresses categories such as tense, mood, aspect, voice or evidentiality; (6) 'cop', a copula is the relation between the complement of a copular verb and the copular verb; (7) 'conj:and', a conjunct is the relation between two elements connected by a coordinating conjunction (and, or, etc.) (8)'punct', punctuation; (9)'acl:relcl', a relative clause modifier of an noun. (The parser also puts other specific relations between the noun and its relative clause, thus this edge is prunable.)

The default prunable POS tags include the following labels: (1) 'PRP', the personal pronoun; (2) 'MD', modal; (3) 'PRP$', possessive pronoun. (4)'WP$', possessive wh-pronoun.

HISyn will traverse the entire graph, prune the trivial edges and nodes with dependency relations and POS tags listed above. Figure 1 and Figure 3(a) show the dependency graph before and after the pruning. It is worth noting that as a cross-domain framework, users could configure HISyn to add or delete the dependencies or POS tags to refine the pruning process in new domains.

4.4 Intermediate Data Structures

This section gives the formal definition of two core data structures used in grammar graph-based translation: dependency graph and grammar graph.

4.4.1 Dependency Graph. Dependency graph is the output from the front end of HISyn. It contains the word form, lemma form, POS tag, and named entity label of each token in a NL-based query and the dependency relations among the tokens. It is used as the basis of the intermediate representation in guiding code generation in the back end. We give the formal definition of a dependency graph as follows:

Definition 1 (dependency graph)**.** A dependency graph is a directed acyclic graph, $G_{NL} = (\mathcal{N}_{NL}, \mathcal{E}_{NL})$, where

- each node $n = (word, lemma, POS, NE) \in \mathcal{N}_{NL}$ corresponds to a token, which includes the word, lemma, POS tag, and named entity tag; NE is empty if the word is not an named entity.
- each edge $e = (n_i, n_j, dep) \in \mathcal{E}_{NL}$ is a direct edge, corresponding to a dependency relation. The edge points from n_i (called *governor*) to n_j (called *dependent*), with the dependency relation dep.

4.4.2 Grammar Graph. A context-free grammar (CFG) is a quadruple $(\mathcal{T}, \mathcal{NT}, \mathcal{S}, \mathcal{P})$ [6], i.e. (terminal symbols, nonterminal symbols, start symbol, productions). To enable efficient valid code search, we introduce a representation, *grammar graph*, to represent a CFG.

A grammar graph defines the search space for code generation. Given a grammar graph, the code generation problem is transformed to the problem of finding a subgraph called *code generation tree* from the grammar graph. The definition of a code generation tree will be introduced in Section 5.5.

In HISyn, the nodes in a grammar graph are of three types, *non-terminal nodes*, *derivation nodes*, and *API nodes*. Edges are of two types, *'or' edges* and *concatenation edges*. We first introduce the definitions of the nodes and edges and then the definition of a grammar graph. In the definitions below, $|*|$ denotes the number of elements in a set. Figure 3(e) is a partial grammar graph in the Air Travel Information System (ATIS) domain. We use the following production rules to illustrate a grammar graph:
$query ::= pred\_set|min\_f|...|project$
$min\_fare ::= MIN\_FARE(col\_fare, pred\_set)$
$pred ::= eq\_dprt, eq\_arr|...$

Definition 2 (Non-Terminal Node)**.** A non-terminal node $n_N \in \mathcal{N}_N$ represents a unique non-terminal symbol in $\mathcal{NT}$. We use $\mathcal{N}$ to represent the set of non-terminal nodes derived from a CFG.

The number of non-terminal nodes is equal to the number of non-terminal symbols (i.e., $|\mathcal{N}_N| = |\mathcal{NT}|$). For example, in Figure 3(e), the node `min_fare` is a non-terminal node.

Definition 3 (Derivation Node)**.** A derivation node $n_D \in \mathcal{N}_D$ corresponds to the string in the right-hand side of a production rule in a CFG. $\mathcal{N}_D$ is used to represent the set of derivation nodes derived from a CFG.

The number of derivation nodes is no more than the number of production rules (i.e., $|\mathcal{N}_D| \leq |\mathcal{P}|$). This is because different production rules could have the same right-hand side string. An example of derivation nodes in Figure 3(e) is node `MIN_FARE(c..,p..)` ('c..' and 'p..' are the abbreviations of the arguments 'col_fare', 'pred_set').

Definition 4 (API Node)**.** An API node $n_A \in \mathcal{N}_A$ corresponds to an API function name. $\mathcal{N}_A$ represents the set of API nodes derived from a CFG.

The number of API nodes is equal to the number of APIs. In Figure 3(e), the node `MIN_FARE` is an API node.

Definition 5 ('Or' Edge)**.** An 'or' edge $e_O \in \mathcal{E}_O$ is a directed edge that points from a non-terminal node n_N to a derivation node n_D, denoted as $e_O = (n_N, n_D)$. $\mathcal{E}_O$ represents the set of 'or' Edge derived from a CFG.

An 'or' edge corresponds to a production rule; its tail represents a non-terminal symbol while its head represents a string of one or more terminal or non-terminal symbols. The number of 'or' edges is equal to the number of production rules (i.e., $|\mathcal{E}_O| = |\mathcal{P}|$). Because some production rules can share the same non-terminal symbol as their left-hand side, a non-terminal node corresponding to the non-terminal symbol can also be tails of more than one 'or' edge. Applying a production rule is equivalent to select one of the 'or' edges. The edges from the non-terminal node `query` to other nodes in Figure 3(e) are 'or' edges.

Definition 6 (Concatenation Edge)**.** A concatenation edge $e_C \in \mathcal{E}_C$ is a directed edge that points from a derivation node n_D to an API node n_A or a non-terminal node n_N, or from an API node n_A to a non-terminal node n_N, denoted as $e_C = (n_D, n_A)$ or $e_C = (n_D, n_N)$ or $e_C = (n_A, n_N)$. $\mathcal{E}_C$ represents the set of concatenation edges derived from a CFG.

When a concatenation edge is (n_D, n_A) or (n_D, n_N), it means that the API represented by n_A or the non-terminal symbol represented by n_N is a sub-string of the symbol sequence represented by n_D. When a concatenation edge is (n_A, n_N), it means that the non-terminal symbol is one of the arguments of the function represented by n_A. In Figure 3(e), $e = (MIN\_FARE(c.., p..), MIN\_FARE)$ is a concatenation edge (n_D, n_A); $e = ((eq\_dprt, eq\_arr), eq\_dprt)$ is a concatenation edge (n_D, n_N); $e = (MAIN\ \_FARE, col\_fare)$ is a concatenation edge (n_A, n_N).

A grammar graph contains all three types of nodes and two types of edges.

Definition 7 (Grammar Graph)**.** A grammar graph is a directed graph, $G_G = (\mathcal{N}_G, \mathcal{E}_G)$, where $\mathcal{N}_G = \mathcal{N}_N \cup \mathcal{N}_D \cup \mathcal{N}_A$ and $\mathcal{E}_G = \mathcal{E}_O \cup \mathcal{E}_C$.

All the concatenation edges $\mathcal{E}_C$ directed from a n_D or an n_A are ordered, and the order corresponds to the order of symbols inside a derivation, or the order of arguments of an API. In Figure 3(e), the number on each e_C indicates the order of the symbols or arguments.

For a non-terminal node n_N that has multiple outgoing 'or' edges, the first 'or' edge will be marked as the default edge and selected to complete a path during synthesis.

5 GRAMMAR GRAPH-BASED TRANSLATION

Grammar graph-based translation is the synthesizing algorithm used in the back end of HISyn to generate code in the target programming APIs. The algorithm centers around the dependency graph for the selection and the ordering of APIs and relies on the grammar graph for code lowering and correctness. It takes the IR of an NL-based query (dependency graph) as the input and annotates each node in the IR with candidate APIs. The annotated dependency graph will go through *reversed all-paths search*, *local reordering*, *candidate paths selection and combination*, and *code generation* steps to finalize the DSL code. We next explain each major step in detail.

5.1 Semantic Mapping

Semantic mapping associates a list of candidate APIs to every node and the edges whose dependency relation dep is a preposition relation ($nmod : preposition$) in a dependency graph. These lists of candidate APIs offer the basis for generating the target DSL code.

Node Mapping. A node $n = (word, lemma, POS, NE)$ in a dependency graph falls into three categories based its *NE* tag and domain knowledge; each category is treated differently to create its candidate API list. The mapping rules for each category are described below:

(1) *Domain Term Nodes.* A *domain-specific term* is a term that is explicitly marked by a user in the input query. It is defined in Section 4.2. A node is a domain node if its *word* is a domain-specific term. It has a one-to-one mapping with an API and thus its candidate API list contains only one API. For example, in ASTMatcher domain, the term 'for statement' corresponds to the API 'forStmt', and 'if statement' corresponds to the API 'ifStmt'.

(2) *Named Entity Nodes.* A node is a named entity node if its *NE* tag is not empty. Because a named entity is usually used as an argument of an API, HISyn generates a list of candidate API for a named entity node by first identifying the APIs related to the node's *NE* tag and then using the node's *word* form as an argument of each API. For example, the node 'Baltimore' is mapped to '[CITI(Baltimore)]' in Figure 3(a, b).

(3) *Regular Nodes.* A node is a regular node if it doesn't fall into the above two categories. For a regular node, a mapping exists if the node's *word*, *lemma*, or one of its synonyms is a token in an API's description. A regular node can have many or zero mapped candidate APIs.

Edge Mapping. An edge (n_i, n_j, dep) is a *preposition edge* if its dep is a preposition relation such as $nmod : from$ and $nmod : to$. A mapping from an API to a preposition edge exists if any of the dep's semantic related words is a token in the API's description. A dep's semantic related words are pre-defined in a preposition dictionary. HISyn only considers preposition edges because prepositions are used to express the temporal or spatial relations between two nouns. These relations contain important semantic information between the nodes connected by preposition edges. Thus, HISyn applies edge mapping to extract such semantics.

We refer to the dependency graph annotated with the lists of candidate APIs as an *annotated dependency graph.*

5.2 Longest-match Scheme

The longest-match scheme is designed to handle cases where a phrase (more than one word) is used to refer to one API on the grammar graph. For example, the 'numeric letter' in Text Editing refers to the API `NUMBERTOKEN`. But if we map "numeric" and "letter" separately, there will be two mapped APIs, `NUMBERTOKEN` and `CHARTOKEN`, respectively.

Inspired by the longest-match principle used in many Scanners to tokenize strings in compilers, we use the *longest-match scheme* to determine the grouping of some nodes in the dependency graph. A single set of candidate APIs will be identified for the group instead of each node within the group.

If two nodes in the dependency graph are connected by a dependency edge with modifier relations (amod, nmod), the edge is called *modifier edge* and these nodes are within a *modifier group.* Other nodes connected to a node in a modifier group with modifier edges also belong to the same modifier group. The node which does not have the governor is the top governor of this group.

For example, consider three modifier edges: A -mod-> B, A-mod-> C, C -mod-> D. Then these four nodes are in one modifier group, and A is the top governor. At this step, each node has its API candidates. For each API inside the A's candidates, HISyn checks how many times this API is also a candidate of other nodes; the result is taken as the score of that API. The APIs with the highest score (ties can happen) are the longest-match candidates for the phrase. Then nodes with candidate APIs stay inside the modifier group, while the other nodes will be treated as regular nodes.

The modifier group will then be treated as one node in dependency graph. The edges that link to the nodes inside the group will link to the top governor with the directions and dependency relations stay unchanged. In Figure 3(a)(b), "flight" and "cheapest" are in one modifier group, and the API `MIN_FARE` is the longest-match API for this modifier group.

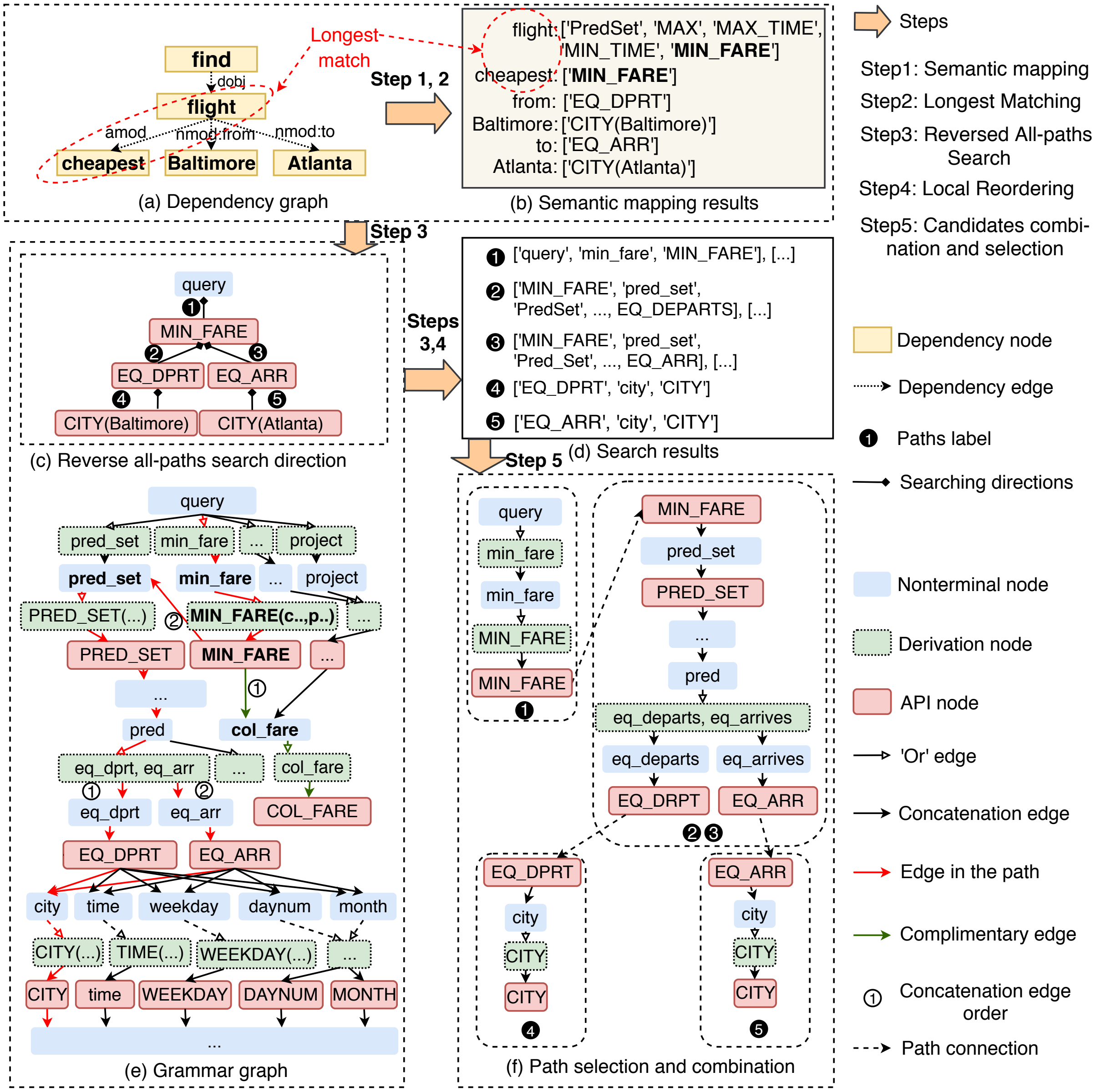


Figure 3: Running examples for grammar graph based translation

5.3 Reversed All-paths Search

Reversed all-paths search identifies a set of candidate paths in the grammar graph for each dependency edge in the annotated dependency graph. These sets of candidate paths will be pruned, reordered, and combined in later steps to determine the orders of candidate APIs. We next introduce several important concepts used in the search algorithm and then describe the algorithm.

A *reversed grammar graph* is a directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in the grammar graph. A *candidate path* is a directed path in the reversed grammar graph or grammar graph. The *root of a candidate path* is the node with no incoming edges; the *leave of a candidate path* is the node with no outgoing edges. Both the root and the leave of a candidate path are API nodes.

For each edge, $(governor, dependent, dep)$ in the annotated dependency graph, the algorithm conducts Breadth-First Search (BFS) on the *reversed grammar graph* to identify a set of candidate paths. The search starts from each candidate API of the node *dependent* and stops when the start symbol of the grammar is reached. During the search, a visited node will not be added to the path again to avoid an infinite loop if recursion exists in the grammar. If a path reaches one of the candidate APIs of the node *governor*, it will be recorded. The paths that end at the start symbol will also be recorded. After getting all the candidate paths for each edge, we apply two pruning strategies to reduce the number of candidate paths: (1) If there exists at least one path that contains any one

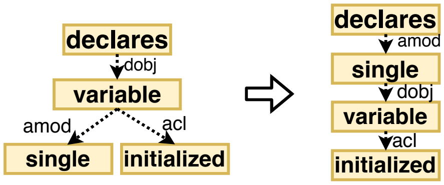


Figure 4: Local reordering

of the candidate APIs for the node *governor*, the paths ending at the grammar's start symbol will be discarded. (2) If a candidate API of the *governor* is not the destination of any paths from any dependent's candidate API, it indicates that this candidate API is not related to the *dependent*. This candidate API will be deleted from the candidate API list of the *governor*. All the paths starting from the deleted candidate API will also be removed. The remaining candidate paths are used to generate final DSL codes in later steps.

The algorithm is motivated by our observation that, when describing the queries in English, people typically follow a gradually refining way. They state the main interest or purpose first, then provide detailed information with dependents. For example, consider the query in Figure 1:

`I would like to find the cheapest flight from Baltimore to Atlanta.`

This query contains the following information: it wants to query about a flight; this flight has the cheapest fare; this flight's departure city is Baltimore; this flight's arriving city is Atlanta.

The edges on the dependency graph show such relations (Figure 3(a)). Inside the graph, the words "cheapest", "Baltimore" and "Atlanta" are all dependents of the "flight" with 'nmod' (nominal modifier) relation. It is consistent with the structure patterns of ATIS query language, which start with the main object or purpose first, then use arguments to specify the details.

Therefore, the direction of edges in the dependency graph pointed out the basic structure of the code: the APIs mapped from the dependent are inside the arguments of the API mapped from the governor. Correspondingly, on the grammar graph, the APIs mapped from the dependent should be the descendants of the API mapped from the governor. In Figure 3(c), we indicate the search direction on the grammar graph. The numbers label the paths for each <start, goal> pair, and Figure 3(d) shows the search results correspondingly.

5.4 Local Edge Reordering

After the reversed all-paths search, HISyn applies local edge reordering to fix the structure of the dependency graph. Although the dependency relations in a dependency graph can guide the ordering of the APIs when generating DSL code, the following two dependency relations are exceptions we observed from our observation of a small set of examples:

(1) *Dependency relations related to subject.* The relative order of a subject and a verb in the code snippets can be the opposite of that specified by their dependency relation. When parsing an English query, the NLP engine makes the verb as the governor of the subject. This structure is sometimes consistent with the code grammar. For example, the code for the description "`A adds B`" is "*add(A, B)*". However, exceptions exist. For example, the code for "`A returns B`" should be "*A(returns(B))*" rather than "*returns(A, B))*".

(2) *Dependency relations related to modifiers.* A modifier is always the dependent of the noun it modifies. But an API can modify another API either by using the second API as an argument or being the argument of the second API. For example, in ASTMatcher domain, "`A declares a single variable`" corresponds to the code snippet "*A(declStmt(hasSingleDecl(varDecl())))*". Because "`single`" modifies the '`variable`', "*hasSingleDecl*" appears before "*varDecl()*". However, the code for "`a global variable`" is "*varDecl(hasGlobalStorage())*". Although "`global`" modifies "`variable`", the API "*hasGlobalStorage()*" is after " *varDecl*".

Given an edge $e = (A, B, dep)$ in a dependency graph whose *dep* is one of the above two types of dependency relations, HISyn conducts reordering if any of the two situations occur: (1) There is no path from B's candidate APIs to A's candidate APIs. (2) There exists at least one path from A's candidate APIs to B's (i.e. reversed direction) that is shorter than the shortest path from B's candidate APIs to A's. The rationale is that a shorter path on the grammar graph is preferred because the corresponding dependent and governor in the dependency graph are neighbors.

If reordering is needed, HISyn changes the positions of the governor and the dependent, as shown in Figure 4. Then reversed all-paths search is re-applied to the reordered edges to find all the candidate paths.

5.5 Candidate Paths Selection and Combination

After reversed all-paths search and local edge reordering, this step combines the sets of candidate paths into *code generation trees* for generating DSL code. We first give a formal definition of code generation tree and then describe the algorithm to generate such trees.

Definition 8 (Code Generation Tree)**.** A code generation tree (CGT) $G_T = (\mathcal{N}_T, \mathcal{E}_T)$ is a subgraph of grammar graph $G_G = (\mathcal{N}_G, \mathcal{E}_G)$, where $\mathcal{N}_T \subset \mathcal{N}_G$ and $\mathcal{E}_T \subset \mathcal{E}_G$. A CGT is a directed acyclic graph whose undirected counterpart is a tree. The root of a CGT is a non-terminal node that corresponds to the start symbol. We define the size of a CGT as the number of API nodes in the tree.

The algorithm first reverses every candidate path so that the direction of each edge in the path is the same as the one in the grammar graph. It then uses only one of the candidate paths from each edge in a dependency graph and combines the selected candidate paths based on the structure of the dependency graph into a CGT. Because an edge in the dependency graph can have more than one candidate path, this step results in many CGTs.

There are two cases for candidate paths combination: siblings paths combination and parent-child paths combination.

(1) *Siblings paths combination.* Two dependency edges are *siblings edges* if they have the same governor. Two candidate paths are *sibling paths* if the corresponding dependency edges are siblings. Because a dependency edge can have many candidate paths, HISyn generates all the combinations of the path candidates from each sibling edge. For each combination, HISyn combines sibling paths by joining the same nodes in the paths to build a prefix tree. If two sibling paths do not have any nodes in common, they are ignored.

A grammar check will be applied to remove prefix trees that are not correct. For each node in a prefix tree, if it is a non-terminal node, HISyn checks if it has only one derivation node as a child; if it is an API node, HISyn checks if the child nodes are the subset of the API arguments. The grammar check at this step only checks the existence of the child nodes, the order of child nodes is considered in the code generation step.

For edges with no siblings, each of the candidate paths will be treated as a prefix tree. The up-right Figure (labeled path 2,3) in Figure 3(f) is one of the prefix trees for paths from `MIN_FARE` to `EQ_DPRT` and from `MIN_FARE` to `EQ_ARR`.

(2) *Parent-child paths combination.* A dependency edge e_1 is the parent of another edge e_2 if e_1's dependent/head is e_2's governor/tail. These two dependency edges are called *parent-child edges*. Two candidate paths are parent-child paths if their dependency edges are parent-child edges. HISyn combines a parent path with the child path by adding an edge directed from the leave of the parent path to the root of the child path. In Figure 3(f), path 4 and path 5 are connected to prefix tree (2,3), and prefix tree (2,3) is connected to path 1.

After all the siblings paths combination and parent-child paths combination, the final connected prefix trees become the code generation trees. The code generation trees then will be transformed to the final code expression in the next step.

5.6 Code Generation

At this stage, HISyn has translated the IR into a set of code generation trees. This step first selects the *minimum CGT* and then uses the minimum CGT to generate the correct code expression. A minimum CGT is the one that has the smallest size (i.e., the smallest number of API nodes).

The rationale of using a minimum CGT for code generation is as follows. The more APIs in a code expression, the more information it conveys. Because all the key information contained in a query is already mapped to candidate APIs during our synthesis process, the smallest CGT is preferred to avoid including redundant or unnecessary APIs in the generated code.

A CGT might miss some important nodes for generating the grammar-correct code. We fill those missing nodes by applying two rules: (1) If a derivation node is in the CGT, then all the non-terminal nodes who are children of the derivation node should also be in the CGT. This is because a derivation node is the right-hand side of a production rule and its children are the non-terminal symbols or terminal symbols in the right-hand side. (2) If an API node is in the CGT, then the descendants of the API nodes should also be in the CGT. This is because an API's descendants will generate the API's arguments. We refer to the filled CGT as a *completed CGT*.

If there are multiple minimum CGTs, HISyn will complete all of these minimum CGTs and choose the minimum completed CGT. If ties still exist, this situation happens when one keyword in a query can be mapped to two or more APIs that have similar syntax roles in the grammar–that is, APIs that can be derived from the same non-terminal and have the same terminals as input. Then HISyn will select the API with the shortest description (All the API descriptions are processed by the NLP engine and the function words are removed when counting the length of the description). This heuristic assumes that a more complex API needs more words to describe it. If this API is needed, the query should be more specific. Thus the limit keywords only lead to an API with less description.

The completed CGT will be transformed into a code expression. HISyn simplifies the CGT by removing all the non-terminal nodes and derivation nodes so that only API nodes are kept in the CGT. The simplified CGT can then be structured to an ordered API-argument sequence (i.e., DSL code) based on the directed edges.

6 EVALUATION

We conduct a set of experiments to examine the efficacy of the HISyn framework cross domains. To test the potential of a pure NLU-driven approach, we use no labeled examples for HISyn. We use the experiments to answer three questions: (1) Compared to data-driven methods trained on many labeled examples, can HISyn produce comparable results without any training example? (2) How does the length of a query affect the accuracy of HISyn ? (3) What are the reasons that cause errors?

We describe the experiment settings in Section 6.1, report our experiment results and comparisons in Sections 6.2, and provide a detailed error analysis in several representative cases in Section 6.3.

6.1 Methodology

6.1.1 Dataset. We use three datasets with different DSLs to evaluate HISyn. The first dataset[2] is the DSL designed for repetitive Text Editing tasks. The second dataset[3] is the DSL designed for the Air Travel Information System (ATIS) [7]. Both of these two datasets come from the work [8]. The third dataset is the DSL in LLVM/Clang, designed for AST node matching.

Text Editing language is a command language that aims to free Office suite application end-users from understanding syntax and semantics of regular expressions, conditionals, and loops. This DSL has 52 APIs in total. The dataset for Text Editing includes 467 English queries and DSL pairs.

Air Travel Information System (ATIS) is a standard benchmark for querying air travel information. This ATIS DSL is designed based around SQL style operations and provides support for predicates/expressions that correspond to important concepts in air-travel queries, arrival/departure locations, times, dates, prices, etc. It has 51 APIs in total. The dataset for ATIS includes 535 English queries and DSL pairs.

ASTMatcher is a tool in Clang/LLVM for constructing AST Matching expressions to find code patterns of interest. It represents a domain with high complexity but scarce labeled examples. There are a total of 505 ASTMatcher APIs with a full-fledged data type hierarchy (over 200 types). We collect 50 ASTMatcher expressions from Clang-tidy and let 5 graduate students write the English descriptions independently. Each ASTMatcher expression is described using a single English sentence from at least two students.

The Text Editing datasets and ATIS datasets do not provide the API documentation and grammar. We manually created these two domain documentations based on the English description and codes. The ASTMatcher dataset provides the official documentation. We generate grammar based on this documentation.

[2] shorturl.at/npFIS

[3] shorturl.at/sxyS5

Table 1: NL queries and codes examples

DSL	Query	Code
ASTMatcher	Find for statements whose init portion declares a single variable which is initialized to the integer literal 0.	forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(hasInitializer(integerLiteral(equals(0))))))));
ATIS	I would like to find the cheapest flight from Baltimore to Atlanta	EXTRACT_ROW_MIN_F(COL_FARE()), AtomicRowPredSet(AtomicRowPred(EQ_DEPARTS(CITY(baltimore), ANY(), ANY(), ANY(), ANY()), EQ_ARRIVES(CITY(atlanta), ANY(), ANY(), ANY(), ANY())))

Text Editing	Insert ":" after #1st word.	INSERT(STRING(:),
		Position(AFTER(WORDTOKEN()), IntegerSet(INTEGER(1))),
		IterationScope(LINESCOPE(), BConditionOccurrence(ALWAYS(), ALL())))

6.1.2 *Evaluation Metrics.*

We use all the test cases in each domain in the experiments. We use DSL codes synthesis accuracy to evaluate the performance of HISyn. The synthesis accuracy is the ratio between the number of *correctly* synthesized DSL code expressions and the number of total test cases. A synthesized DSL code is *correct* if it is identical to the ground truth code, including the APIs, variables, values, and their relative order.

6.1.3 *Methods for Comparison.*

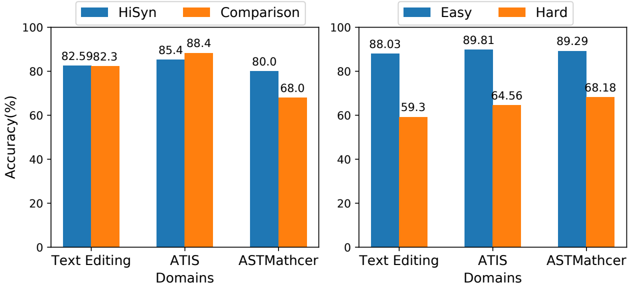
We compare HISyn with a data-driven synthesizer [8] in Text Editing domain and ATIS domain, and a rule-driven synthesizer in ASTMatcher domain. We choose it for comparison because (i) As HISyn, it is a generic synthesizer that takes NL queries as inputs and produces code expressions for multiple domains. (ii) on the available training data in Text Editing and ATIS domains, it gives accuracy comparable to the existing domain-specific NL-based synthesizers.

In this prior work, the data-driven synthesizer [8] builds a generic NL-based synthesizer using machine learning. It takes as input DSL definition and training data consisting of NL/DSL pairs and builds a synthesizer by learning the weights and classifiers to rank the output of keywords-programming based translation.

The prior work applies to the Text Editing domain and ATIS domain, but not the ASTMatcher domain. It is because this domain has over 500 ASTMatcher APIs. A large number of APIs brings many complexities to this domain. With various usage conditions, a data-driven method requires a large number of training examples to train a statistical model that possibly covers most of the use cases of APIs. The real-world labeled cases in this domain are yet scarce. Rule-based approach works better in this scenario. We hence build a rule-based synthesizer as a comparison in this domain. It is based on previous work [25], which uses simple NL processing and heavily relies on type checks and domain heuristics.

6.2 Accuracy of HISyn

Figure 5(a) reports the overall accuracy of HISyn. The accuracies of the data-driven method (on the first two domains) come directly from the previous paper [8], obtained after creating/collecting hundreds of labeled examples and using them to train. HISyn, without using any training example, achieves comparable accuracies with them, 82.59% for text editing and 85.47% for ATIS.



(a)Comparison with other methods (b)Comparison on query difficulties

Figure 5: Accuracy of HISyn in three different domains

In the ASTMatcher domain, the rule-driven synthesizer achieves 68% accuracy, 12% lower than HISyn. The result shows the benefits of NLU and its mapping algorithm over rule-driven methods.

Figure 5(b) shows the accuracy comparison on queries with different lengths. The average query lengths of Text Editing domain, ATIS domain, ASTMatcher domain are 7, 12, and 9. HISyn achieves 88.03% , 89.81%, 89.29% accuracy on shorter queries in three domains, and 59.3%, 64.56%, 68.18% on longer (often harder) queries.

The much higher accuracy on the easy queries is due to two major reasons: (1) Given a hard query, an NLP engine in the front end is more likely to generate a wrong dependency graph. (2) In the back end, a hard query increases the number of paths to combine and select, making it harder to generate the right CGTs.

6.3 Error Analysis

Unlike the "black-box" method in Neural Network-based data-driven approaches, the interpretable nature of HISyn makes error diagnosis easy to do. In this section, we analyze three major cases where HISyn fails to synthesize the correct code expressions to provide some insights. Both the front end and the back end could cause errors. The errors in the front end are typically caused by the ambiguity in natural language or incorrect semantic mapping. The errors in the back end result from wrong decision paths selection and combination steps.

6.3.1 *Limitations in NLP Engines.*

NLP engines may cause errors. Consider the following two test queries:

Query-1: Would you **tell** me the cheapest one way fare from Boston **to Oakland**?

Query-2: Please give me information concerning a flight from Washington DC to Philadelphia the earliest **one** in the morning.

Query-1 is an example of lexical ambiguity. The dependency parser treated 'Oakland' as the noun modifier of 'tell' (i.e. tell ... to Oakland), resulting in the wrong dependency edges in the IR. Query-2 is an example of semantic ambiguity. 'one' is considered as a number and taken as a part of the phrase 'one in the morning'. But 'one' is a pronoun in this query. This semantic ambiguity leads to the mapping of API `Time(1am)`, which is not the intention of the original query.

When we manually fix the parsing errors, HISyn gives correct synthesis results. As NLP techniques progress, NLU-driven synthesis is expected to provide even better results.

6.3.2 *Incorrect Semantic Mapping.*

This type of errors is caused by passive voice or a word that combines the semantic meaning of several APIs. For example:

Query-3: Print any word that **is followed** by 'team'.

Query-4: **Prepend** each line with "#".

In query-3, 'is followed' here means the position before the word 'team', and it should be mapped to API `BEFORECOND()`. The document of the API has "before" relation described, and the synthesizer fails in recognizing that 'follow' essentially means that relation but in a different way, and hence results in a wrong mapping. In query-4, the word 'prepend' means 'insert at the beginning', which can be mapped to two APIs, `INSERT()` and `START()` (`START()` indicates the location of insertion is at the beginning). However, the semantic mapping step only identifies one API (`INSERT()`) for each node. The missing of 'beginning' leads to the error. Applying a deeper semantic analysis for semantic mapping may help this situation.

6.3.3 *Wrong Decision in Code Generation Step.*

In the code generation step, we use the minimum CGT to generate the final code expression. It works in most of the cases because the minimum CGT covers all the information inside a query with the least number of APIs. But exceptions exist.

Query-5: In every line, delete the text after "//"

The correct code expression should be:

```
REMOVE(SelectString(TEXTTOKEN(),
        BConditionOccurrence(
          AFTERCOND(STRING(//), IMM()), ALL())),
      IterationScope(LINESCOPE(),
        BConditionOccurrence(ALWAYS(),ALL())))
```

The synthesized code expression is:

```
REMOVE(SelectString(TEXTTOKEN(),
        BConditionOccurrence(ALWAYS(),ALL())),
      IterationScope(LINESCOPE(),
        BConditionOccurrence(
          AFTERCOND(STRING(//), IMM()), ALL())))
```

The error is the location of the condition "after '//' ". "After '//' " is the condition for deleting the text. But in synthesized code, it is inside the condition for iteration.

This error occurs when the following candidate paths from the two dependency edges are combined: $e_1 = (delete, line, nmod\colon in)$ and $e_2 = (delete, STRING(//), nmod\colon after)$. e_1 has one path p_{11} = *[REMOVE, IterationScope, LINESCOPE]*. e_2 edge has two paths p_{21} = *[REMOVE, SelectString, BConditionOccrance, AFTERCOND]*, p_{22} = *[REMOVE, IterationScope, BConditionOccrance, AFTERCOND]*. When combining paths of e_1 and e_2, the CGT combined from p_{11} and p_{22} will have smaller size (5 APIs) than the combination of p_{11} and p_{21} (6 APIs), since *[REMOVE, IterationScope]* in p_{11} and p_{22} will be combined as prefix. Thus the smaller CGT will be chosen to generate the final code expression. Using deeper semantics to guide the paths selection and combination steps could be one of the solutions to avoid this error.

6.4 Threats to Validity

There are several factors that may threaten the validity of HISyn.

The quality of API documentations. Like in human programming, the quality of the documentation of the target API or language is important. The API description decides which APIs will be the candidates for a mapping element in IR. If the description is not precise, an API may not be selected as a candidate, which leads to errors in results. If the description of several APIs is similar, more unrelated APIs will be selected, and result in more paths in later steps and cause errors potentially. The issue can be mitigated by providing some guidelines or even tools to help document developers in ensuring the quality of documentation.

The quality of queries. The queries are the only source information that describes the user's intention. Thus, the quality of queries directly affects the synthesized results. A query with grammar errors could mislead the NLP engine and result in a wrong dependency graph; a query with imprecise descriptions will not describe the intention clearly and could lead to wrong semantic mappings. The issue can be mitigated by extending the front end of HISyn into an interactive module which may clarify users' intentions by interacting with users.

7 RELATED WORK

NLP has been used for software maintenance and other purposes [1, 4, 10, 12, 15, 21, 36, 50, 51, 53]. Our discussion concentrates on work closely related with code synthesis.

Various specifications are used for code synthesis. A specification can be first-order logic expressions [18, 23], a set of examples [16, 17, 44], natural language [8, 19, 25, 26, 43, 48], partial programs [13, 45] or any other form that is easier to write than the expected program. As HISyn is a natural language-based synthesizer, we concentrate on prior work on program synthesis from natural language (NL).

Rule-based approaches have been developed to synthesize programs for domain-specific tasks such as smartphone automation [25], SQL queries [27, 48], and SpreadSheet data analysis [19]. In contrast, HISyn is featured by cross-domain extensibility.

Recent efforts have been spent on machine learning-based approaches [2, 5, 8, 9, 11, 14, 20, 22, 24, 28–30, 37–41, 43, 46, 49, 52]. For example, Desai and others [8] presented a general framework for constructing program synthesizers given a domain-specific language (DSL) definition and training data that contains example pairs of English sentences and the expected programs in the DSL.

Quirk [40] uses the semantic mapping approach that learns to map natural-language descriptions of "if-then" rules to executable code. Lin [28] leverages recurrent neural networks (RNNs) for NL to code translation. Chen [5] applies LSTM-based sequence-to-sequence model with other specifications for code synthesis. Applying these approaches to program analysis would require many training examples to cover the vast space of possible code complexities and situations. HISyn avoids the barrier by taking full advantage of the domain knowledge, NL dependencies, and the grammar graph-based translation.

Another body of work is API learning. This work tries to identify some statistical patterns of API usage. Examples include code search tools [31, 33], API usage pattern mining [42, 47], API sequence generation [14]. These studies rely on statistical machine learning techniques. They hence require a large set of examples, requiring extra efforts when applying to other domains.

8 DISCUSSIONS

Our study has focused on the core technical challenges. Some engineering considerations may need to take in the practical usage of HISyn. For instance, even though HISyn gives reasonably accurate results, the accuracy could be potentially improved with some interactive features added. The tool could provide some informative feedback to users, such as the dependency graph from the NL query and the semantic mapping between the nodes and the APIs (like Figure 3 (b)). That could give programmers insights on the generated code expressions and help them replace erroneous parts. At programmers' demands, the tool may also show other candidates (not only the minimum heuristic) in the semantic mapping list of a node (optionally in the order of promise) to offer programmers more choices. Prior studies [17, 19, 27] have explored interactivity for NL-based code synthesis; we decided to focus this work on the core challenges and leave the interactivity as a feature to add in the future.

The NLP tool we select to use is not the one with the most cutting-edge NLP techniques, but the one with common adoptions for its maturity. There has been much progress in NLP in recent years, which provides promising techniques to improve NLP results. We foresee that as these techniques become more mature and get integrated into practical NLP tools, the accuracy of HISyn could get further improved.

9 CONCLUSION

This paper introduces NLU-driven code synthesis, a new approach to NL programming. Experiments on framework HISyn demonstrate that NLU-driven, without using any training example, can produce results comparable with data-driven methods trained on hundreds of labeled examples. It saves the burden in example collections, avoids dataset-caused biases, and at the same time, gives much better interpretability and support for error diagnosis. In domains already having many labeled data, the NLU-driven method could potentially combine with data-driven methods to reduce biases and increase interpretability, which is left for the future to explore.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1525609, CCF-1703487, CNS-1717425. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[2] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[3] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S Lam. 2019. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 394–410.

[4] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.

[5] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 602–612.

[6] Keith Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.

[7] Deborah A Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the ATIS task: The ATIS-3 corpus. In *Proceedings of the workshop on Human Language Technology*. Association for Computational Linguistics, 43–48.

[8] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 345–356.

[9] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 990–998.

[10] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. 2019. A System-Wide Debugging Assistant Powered by Natural Language Processing. In *Proceedings of the ACM Symposium on Cloud Computing*. 171–177.

[11] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL (1)*. http://aclweb.org/anthology/P/P16/P16-1004.pdf

[12] Michael D Ernst. 2017. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[13] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* 52, 1 (2017), 599–612.

[14] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[15] Hui Guan, Xipeng Shen, and Hamid Krim. 2017. Egeria: a framework for automatic synthesis of HPC advising tools through multi-layered natural language processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[16] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[17] Sumit Gulwani. 2012. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*. IEEE, 8–14.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

[19] Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 803–814.

[20] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Acm Sigplan Notices*, Vol. 50. ACM, 416–432.

[21] Sonia Haiduc, Venera Arnaoudova, Andrian Marcus, and Giuliano Antoniol. 2016. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 898–899.

[22] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.

[23] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[24] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. 2004. Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *The AAAI-2004 workshop on supervisory control of learning and adaptive systems*. San Jose, CA.

[25] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 193–206.

[26] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 593–604.

[27] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment* 8, 1 (2014), 73–84.

[28] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D Ernst. 2017. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01* (2017).

[29] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. *arXiv preprint arXiv:1802.08979* (2018).

[30] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[31] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 260–270.

[32] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60. http://www.aclweb.org/anthology/P/P14/P14-5010

[33] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.

[34] George A Miller. 1998. *WordNet: An electronic lexical database.* MIT press.

[35] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, and John McCarthy. 1963. Revised report on the algorithmic language Algol 60. *Commun. ACM* 6, 1 (1963), 1–17.

[36] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. 2018. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[37] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[38] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[39] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. *arXiv preprint arXiv:1802.04335* (2018).

[40] 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)*. 878–888.

[41] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL (1)*. 1139–1149. https://doi.org/10.18653/v1/P17-1105

[42] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean-Code Search and Idiomatic Snippet Synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 357–367.

[43] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples.. In *IJCAI*. 792–800.

[44] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 515–527.

[45] Armando Solar-Lezama and Rastislav Bodik. 2008. *Program synthesis by sketching*. Citeseer.

[46] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. 2017. Building natural language interfaces to web apis. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 177–186.

[47] Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 54–57.

[48] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 63.

[49] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[50] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.

[51] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2019. CPC: automatically classifying and propagating natural language comments via program analysis. (2019).

[52] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[53] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.