J Inf Process Syst,
http://dx.doi.org/10.3745/JIPS

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

A Survey of Automatic Code Generation from Natural Language

Jiho Shin*, Jaechang Nam*

Abstract

Many researchers have carried out studies related to programming languages since the beginning of computer science. Besides programming with traditional programming languages (i.e., procedural, object-oriented, functional programming language, etc.), a new paradigm of programming is being carried out. It is programming with natural language. By programming with natural language, we expect that it will free our expressiveness in contrast to programming languages which have strong constraints in syntax. This paper surveys the approaches that generate source code automatically from a natural language description. We also categorize the approaches by their forms of input and output. Finally, we analyze the current trend of approaches and suggest the future direction of this research domain to improve automatic code generation with natural language. From the analysis, we state that researchers should work on customizing language models in the domain of source code and explore better representations of source code such as embedding techniques and pre-trained models which have been proved to work well on natural language processing (NLP) tasks.

Keywords

Survey, Software Engineering, Source Code Generation, Naturalistic Programming

1. Introduction

The programming languages we used to develop software products have limitations. First, the cost of learning different programming languages is high for novice developers [1, 2]. Considering the number of languages that exist, the cost can be very high. Second, software products are getting too complex, leading even expert developers to have a difficult time understanding code that others developed [3-5]. Last, programming language limits our expressiveness because we have to translate logical thinking into a foreign (programming) language [3, 5].

These problems state that programming has high entry barriers. White and Sivitanides [6] presented a theory that certain programming languages (i.e. procedural or object-oriented programming languages) require cognitive characteristics of formal operations, which is the highest cognitive development level. Unfortunately, many people fail to achieve formal operations until the end of their college years. Ghezzi and Mandrioli [7] state that to be a software engineer, one must master the foundation, design methods, technology, and tools of the engineering discipline. They also claim that software engineers must have a solid background in all fundamental areas of mathematics. All here refers to 1) traditional continuous mathematics (differential/integral analysis and calculus), 2) discrete mathematics (logic, combinatorics, and algebra), and 3) statistics and probability theory.

Programming with natural language might be one of the potential ways to mitigate these barriers although it may not directly resolve these challenges now. One of the main difficulties in programming is that it requires the ability to deal with abstractions and logical thinking, to form a hypothesis, to solve problems systematically, and to perform mental manipulations [6]. These skills have to be done within the syntax of a certain programming language. However, we use these skills quite ‘naturally’ with our natural language. Thus, programming with natural language could be an initial step and an important direction to mitigate the entry barriers of programming.

Manuscript received Month 06, 20XX; accepted Month 06, 20XX.
Corresponding Author: Jaechang Nam (jcnam@handong.edu)
* School of Computer Science and Electrical Engineering, Handong Global University, Pohang, Korea (jihoshin@handong.edu, jcnam@handong.edu)

In this sense, many researchers have studied code generation from natural language [1, 8-12]. With a natural language description, the system generates a corresponding code snippet or fully runnable source code. For example, Pegasus [13], a tool that generates Java source code from the natural language description, receives a description of a program as in Fig. 1a. With the received program description, it will produce fully runnable source code, as shown in Fig. 1b.

```
Take the matrix ([2, 2, 1], [1, 4, 3]).
Print it.
New line.
If the first element of the second row of the matrix is
smaller than 3 then write "I can understand you!".
```

(a) An example of input program description in Pegasus

```
// Take the matrix ([2, 2, 1], [1, 4, 3]).
long[][] matrix74 = new long[2][3];
matrix74[0] = new long[]{2, 2, 1};
matrix74[1] = new long[]{1, 4, 3};

// Print it.
System.out.print("(");
for (int i7 = 0; i7 <= 1; i7++)
{
   System.out.print("[");
   for (int i8 = 0; i8 <= 2; i8++)
   {
      System.out.print(matrix74[i7][i8]);
      if (i8 < 2) System.out.print(", ");
   }
   System.out.print("]");
   if (i7 < 1) System.out.print(", ");
}
System.out.print(")");

// New line.
System.out.println("");

// If the first element of the second row of the matrix is
// smaller than 3 then write "I can understand you!".
if (matrix74[1][0] < 3)
   System.out.println("I can understand you!");
```

(b) An example of output source code in Pegasus.

Fig. 1. Pegasus examples [13]

However, these studies have vague research goals and motivations because each proposed approach has studied without general and formal descriptions of the program with inputs and outputs. For instance, Pegasus [13], requires a program description of code in a line-by-line fashion to produce a corresponding source code as the output. Other approaches, such as Macho [14], take an abstract description of code and a test case example to supplement the ambiguity caused by the abstraction and produce the corresponding program. There are many approaches that generate source code from natural language, but their forms are very variant, and the goal of the research field is unclear.

Related studies that are closest to our work is the survey of Pulido-Prieto and Juárez-Martínez [5] and Allamanis et al. [3]. Pulido-Prieto and Juárez-Martínez [5] list approaches that assist programming with natural language elements by enhancing the expressiveness in programming. Allamanis et al. [3] list approach that uses machine and deep learning techniques on source code corpora. The difference between their work and ours is that their surveyed approaches do not have to handle actual natural language texts (natural language elements [3] or source code [5]) as input. Also, their approaches do not have to produce the source code of a program as an output. All the approaches that our survey lists handle actual natural language texts as input and produces the corresponding source code.

The goal of this paper is to survey and review the approaches that generate source code with natural language descriptions. The first study in this research field [11] has started many years ago, but the follow-up studies have not been actively conducted because of the limitations of the technology. In the past decade, this research field got more attention since various deep learning and natural language processing techniques [8, 15-17] have been proposed. From surveying and analyzing the trend of this research field, we suggest that more studies related to deep neural architectures for source code generation should be explored. Another practical improvement can be explored by studying the better representation of source code such as embedding techniques and pre-trained models which have been proved to work well on NLP tasks.

The contribution of this paper is:

1) We made a taxonomy for different approaches with the form of input and output.

2) With this taxonomy, we categorized the different approaches and analyzed the research trends.
3) We suggest future directions for how the research should proceed in natural language programming.

2. Related Work

There are three surveys related to this research field. Pulido-Prieto and Juárez-Martínez [5] surveyed approaches of naturalistic programming tools and languages. They also organized a table that shows the different properties and functionalities of each approach. Allamanis et al. [3] surveyed approaches that applied machine learning techniques on big code in the point of 'naturalness'. It is about approaches using a machine or deep learning techniques on source code corpora to extract patterns that help software engineering practice. They claim that these approaches are modeled to learn the 'naturalness' of developed source code. Song et al. [18] surveyed approaches about techniques and algorithms that generate code comments from source code. They categorized the approaches into three groups: comment generation algorithms based on information retrieval, comment generation algorithms based on deep neural networks, and other comment generation algorithms.

2.1 A Survey of Naturalistic Programming Technologies [5]

Pulido-Prieto and Juárez-Martínez [5] listed 31 different approaches about tools and programming languages that assist users to program with the elements of natural language features. The elements of natural language, which they call *naturalistic programming,* is a formal and deterministic implementation of features from natural language. Examples of natural language features consist of deixis, expressiveness, phrases, anaphoric relations, context, ambiguity, etc. They have categorized the approaches to tools and languages. They tabulated tools by whether they provide the following functionalities: automatic code generation, reserved keywords, pre-defined grammars, data dictionaries, predefined code fragments, multiple programming language support, indirect references, industry-focused, learning-focused, documentation-focused. Second, they tabulated languages by the following functions: code generation, data dictionaries, English-like expressiveness, indirect references, learning-focused, and report generation.

The difference between their survey and ours is that 1) they have a broader concept of natural language because they consider certain features while we only consider actual natural language texts. 2) The approaches they list do not necessarily produce source code because they focus on assisting in programming, while our survey focuses on automatically generating source code.

2.2 A Survey of Machine Learning for Big Code and Naturalness [3]

Allamanis et al. [3] list machine and deep learning approaches that learn about many aspects of different source code corpora. The authors categorized the models into three groups according to the form of different modalities they focus: 1) code-generating models, 2) representational models of code, and 3) pattern-mining models.

Code-generating models are probabilistic models that describe the corpus of source code in a stochastic process for generating source code. The training data has the form of a source code corpus and can have an empty context or context with other non-code modality. When the context is an empty set, the model's probability distribution is a language model of code. When the context information is in the form of other non-code modalities (i.e. images or natural language), we can describe it as a code-generative multimodal model of code. The context information can also be in the modality of code, making the probability distribution a transducer or translation model to generate code from one language to another.

Representational models of code capture intermediate representations such as vector embedding. These representations are not necessarily human-interpretable but capture properties of the code by learning the conditional probability distribution of a code property such as variable types. With representational models, researchers build prediction models for property extraction. These prediction models that predict properties of source code are used to specific applications for developers, e.g., identifier naming [19, 20], code search [15], program analysis [21, 22], code fixing [23, 24], code summarization [25, 26], API search [27], and comment prediction [28].

The pattern-mining models of source code focus on capturing human-interpretable patterns that can directly help software engineers. An example application includes finding API patterns [29, 30], idiom mining [31], defect prediction [32, 33], clone detection [34], code summarization [35], etc.

The focus of our survey is classified as a code-generative multimodal model because receiving natural language description is considered receiving non-code modality for context. Our survey differs from that of Allamanis et al. [3] because they only consider techniques regards to machine or deep learning. Furthermore, the survey of Allamanis et al. considers mostly on approaches that learn the representation of code. There are some approaches regards to generation of code but most of them do not focus on generating from natural language description of code [3]. However, our survey comprises both machine and non-machine learning techniques that generate source code from natural language.

2.3 A Survey of Automatic Generation of Source Code Comments [18]

Song et al. [18] listed various methods that generate code comments from source code. They categorized the approaches into three big groups. The first group is an information retrieval-based algorithm, and they are subdivided into the following subgroups: VSM/LSI based algorithms, code clone detection-based algorithms, LDA based algorithms, and other information retrieval-based algorithms. The second group is deep neural networks-based algorithms with the following subgroups: RNN based algorithms with single and multi-encoders, and other neural network-based algorithms. The last group is other comment generation algorithms with the following subgroups: software word usage model-based algorithms, ontology-RDF based algorithm, Stereotype based algorithms, and other algorithms.

They also organized evaluation metrics used in code commenting approaches. The metrics are divided into two groups: human evaluation metrics and automatic evaluation metrics. Human evaluation metrics are subdivided into three subgroups: content in terms of relevance to code, the feature of natural language, and effectiveness in terms of roles. Automatic evaluation metrics are subdivided into four subgroups: BLEU, METEOR, ROUGE, and CIDEr.

The survey of Song et al. [18] is different from our survey in that the approaches generate natural language comments from source code which has opposite directions than that of the approaches in our survey. However, they are related because they both have to learn the features of natural language and source code. They use similar techniques such as information extraction and neural network models. The major difference between these tasks is that natural language generation is more robust while source code has high sensitivity.

3. Survey Criteria

In this section, we explain the criteria for listing the approaches of source code generation with natural language descriptions. First, we include all papers of approaches that generate source code from natural language in the surveys from Pulido-Prieto and Juárez-Martínez [5] and Allamanis et al. [3]. Second, we searched for more papers on Google Scholar to supplement the list of approaches because the aforementioned surveys do not focus on approaches that generate source code from natural language. The following are the steps of how we selected the papers in detail:

- When searching the literature, we used the following search strings: *natural language programming, naturalistic programming, natural language (source) code generation/generator, natural language abstract-code/pseudo-code/skeleton-code generation/generator, natural language (source) code synthesis.*
- From the results, we only consider research papers.
- We completed the list of papers that meet the criteria below:
  1. The approach must receive program descriptions in the form of natural language.
  2. The approach must generate the corresponding source code modality of the program description.
  3. Approaches may require additional input and/or output, but it is optional. Additional input and output include actual input values and/or files to be computed, and the output result when the program is executed.

We found 35 papers based on the criteria and categorized them by using notations defined in Section 4.

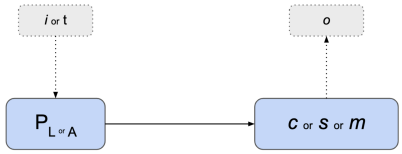

Fig. 2. Overall structure of approaches.

4. Notations

This section specifies each notation used to categorize the approaches. We divided the notations into three parts: program form, input form, and output form. The taxonomy we organized uses these notations to show the different structures of each approach. The notations with upper-case letters indicate the modality of natural language while notations with lower-case letters indicate modalities of non-natural language. Natural language modality has the form of description. Modalities of non-natural language have the form of the actual value of notation or source code.

4.1 Program Form

A notation of a program form is an essential part of all the approaches. As seen in Fig. 2, *'$P_{L\ or\ A}$'* is on the left-hand side of the structure, indicating that the approaches receive a natural language description of the program. On the right-hand side of the structure, *'c or s or m'* are produced, indicating that approaches generate a certain form of a program as output.

4.1.1 P_L - description of code (line-by-line).

'P_L' is a notation in the form of natural language where it describes a code in a line-by-line fashion. For example, "read an integer *i* and increment *i* until 100. While incrementing, add *i* to an integer *sum*. Print the integer *sum* on the screen". This is an example code that sums integer values from 0 to 100. It has a similar structure of pseudocode in the form of natural language. Due to its structural property, it lacks abstraction and makes the natural language description less 'natural'.

4.1.2 P_A - description of code (abstract).

'P_A' is a notation in the form of natural language where it describes the source code more abstractly. The followings are examples: "a program that sums integers 0 to 100"' or "read a file and split the content by commas"'. This form of code description is much more abstract and 'natural' than notation *'P_L'*, but it is more ambiguous than *'P_L'*.

4.1.3 c - actual program code.

'c' is an actual program code generated from a natural language description of a program. The code is complete and runnable.

4.1.4 s - code snippet.

's' is also in the form of actual source code similar to *'c'*. The difference is that *'c'* is a full-runnable code while code snippet is just a part of code that is not directly runnable but has some functions that the user wants to perform or handles only a certain part of logic in a program.

4.1.5 m - intermediate program.

'm' is an intermediate form of a program to mediate the two extreme languages: natural language and programming language. For instance, it can be in the form of knowledge representation for a better understanding (for both computer and human perspective); it can be in the form of a skeleton code where it helps users develop from code structure; it can be in the form of a regular expression.

4.2 Input Form

The input form is the notation that is read into the generated programs. As stated in Fig. 2, some approaches require input form, but it is not mandatory. Since it is an 'input' form, it is on the left-hand side of the structure. In Section 5, categories with an input form are marked as a subscript of *'P'* (e.g. $P_{L\{i\}}$).

4.2.1 i - the actual input.

'i' indicates an input form that corresponds to the actual input that is read into the target program. For instance, operand literals in a mathematical word problem-solving program;

excel file in an excel formatting program; and a corpus of source code in a program synthesis domain, etc.

4.2.2 t - test case.

't' is an input form of an actual test case. We made this notation for disambiguation because test cases comprise both actual input and the corresponding oracle (expected output).

4.3 Output Form

The output form is a notation that is produced from running the generated programs. As stated in Fig. 2, some approaches always produce output while others do not. The notation of the output form is on the right-hand side of the structure. There is only one type of output form which is *'o'* - the actual output. Notation *'o'* indicates the actual output that is resulted from the generated program. In section 5, categories with output form are marked as a subscript of *'c'* (e.g. $c_{\{o\}}$).

Table 1. The Approaches of automatic code generation from natural language in their categories

Category	Approaches	Short Description	Year
$P_{L\{i\}} \Rightarrow c_{\{o\}}$	1. Heidorn [11]	1. Interactive simulation programming	1973
	2. Little & Miller [12]	2. Commands → Web/MS Word code	2006
	3. Gulwani & Marron [37]	3. MS Excel translator (Nlyze)	2014
	4. Shi et al. [38]	4. Number word solving problem	2015
	5. Mandal & Naskar [39]	5. Number word solving problem	2017
$P_L \Rightarrow c$	1. Price et al. [1]	1. NaturalJava	2000
	2. Chong & Pucella [40]	2. NL interface framework	2004
	3. Begel et al. [41]	3. Spoken Java	2005
	4. Vadas & Curran [2]	4. NL → Python	2005
	5. Knöll & Mezini [13]	5. Pegasus	2006
	6. Lieberman & Ahmad [43]	6. MOOIDE	2010
	7. Le et al. [44]	7. SmartSynth	2013
	8. Lanhäußer et al. [45]	8. Natural Language Command Interpreter	2016
$P_A \Rightarrow s$	1. Allamanis et al. [8]	1. Source code ↔ natural language	2015
	2. Gvero & Kuncak [46]	2. Free-form queries → Java expression	2015
	3. Quirk et al. [47]	3. Semantic parsers for IFTTT recipes	2015
	4. Rahothaman et al. [48]	4. SWIM	2016
	5. Ling et al. [16]	5. Latent predictor networks	2016
	6. Desai et al. [10]	6. Program synthesis of NL → DSL	2016
	7. Yin & Neubig [17]	7. Probabilistic grammar model → Python	2017
	8. Lin et al. [49]	8. NL → shell commands	2017
	9. Zhong et al. [51]	9. Seq2SQL	2017
	10. Sirres et al. [52]	10. Free-form code snippet search	2018
	11. Lin et al. [50]	11. NL → shell commands	2018
	12. Gu et al. [15]	12. Deep code search	2018
	13. Schlegel et al. [53]	13. Step-by-step programming with NL	2019
$P_A \Rightarrow m$	1. Liu & Lieberman [54]	1. Metafor	2005
	2. Clark et al. [55]	2. CPL	2005
	3. Clark et al. [56]	3. CPL-lite	2010
	4. Somasundaram & Swaminathan [57]	4. Natural language compiler	2011
	5. Soeken et al. [58]	5. Assisted BDD using NLP	2012
	6. Kushman & Barzilay [59]	6. Semantic unification of regex from NL	2013
$P_{A\{t\}} \Rightarrow c$	1. Cozzie et al. [14]	1. Program description + unit test → Java	2011
	2. Cozzie & King [61]	2. Program description + unit test → Java	2012
	3. Manshadi et al. [62]	3. Programming by examples with NL	2013

5. Category

In this section, we list the categories of automatic code generation approaches with their form of structures notated with code, input, and output forms described in Section 4.

Table 1 shows the list of papers we surveyed. The list is grouped by their categories and is sorted by the year within the categories.

5.1 $P_{L\{i\}} \Rightarrow c_{\{o\}}$

Approaches in this category require a natural language description that has the form of a line-by-line fashion and actual input values to be computed. What these approaches produce for the $P_{L\{i\}}$ is fully runnable source code with the actual output value, $c_{\{o\}}$.

However, they are only able to generate source code for specific domains such as Excel functions or mathematical problems. By focusing on the specific types of programs, the approaches in this category could generate complete runnable source code when specific types of inputs and program descriptions are given. Also, the ability to handle the abstraction of natural language description is constrained and has a form of a line-by-line fashion.

5.1.1 An interactive simulation programming which converses in English [11].

Heidorn [11] proposed a natural language conversing system that takes a description of a program to generate its corresponding source code. The program belongs to a certain domain where programs answer specific questions. The system first checks the completeness of the description of the program. If the first stated program description is sufficient, the program will produce the corresponding problem-solving program with the answer. If the program description is insufficient or ambiguous, the system will ask questions to resolve the ambiguity. The program takes in both description of the program and the input values which mostly consist of number literals to be computed. The system continuously asks users questions until a sufficient amount of ambiguity is resolved to answer the question from the program description. The generated source code is in the form of GPSS (Gordon's Programmable Simulation System or General-Purpose Simulation System). GPSS is a general-purpose programming language for a discrete-time simulation developed by Geoffrey Gordon [36]. It is used to show process flow-oriented simulation such as simulating workflows in factories.

5.1.2 Translating keywords into executable code [12].

Little and Miller [12] proposed a system that translates keyword commands to executable code in web and Microsoft Word. This approach generates fully runnable code from the natural language description with the resulted web page or a document file. For instance, when a user types in a keyword command “click search button”, it is translated to the text “click(findButton(“search”))”. The translated source code is then executed in the form of the web as an output. They implemented a similar function in the domain of Microsoft Word. For example, when a user types in “left margin 2 inches” the corresponding Visual Basic code “ActiveDocumen-t.PageSetup.LeftMargin = InchesToPoints(2)” will be generated with the document file.

5.1.3 NLyze: interactive programming by natural language for spreadsheet data [37].

Gulwani and Marron [37] proposed a natural language-based interface for spreadsheet programming. This approach translates a natural language description of Excel functions such as algebraic calculations and table configurations to generate and rank corresponding program candidates. Aside from the description of an Excel function, NLyze takes in an input spreadsheet file as a source. As the outcome, the spreadsheet program candidates and the modified spreadsheet is generated.

5.1.4 Automatically solving number word problems by semantic parsing and reasoning [38].

Shi et al. [38] proposed a system that uses semantic parsing and reasoning to generate source code that calculates mathematical word problems. The system receives a description of a mathematical word problem with the input numbers. Then, the system generates the corresponding source code and the answer to the math problem. The generated source code is in the form of the DOL language (abbreviation for DOlphin Language) which is a semantic representation language designed by Shi et al. [38].

5.1.5 Natural language programming with automatic code generation towards solving addition-subtraction word problems [39].

Mandal and Naskar [39] proposed a system that generates a program solving mathematical word problems from natural language descriptions. The approach extracts relevant information from the natural language description and stores them into an object-oriented template. The mapped template is used to

perform mathematical operations to solve the problems. For information extraction, they used natural language semantic role labeling and made a template called the OIA triplet (Owner-Item-Attribute) and stored the template with the extracted information. The input and output form of the system is identical to the approach in Section 5.1.4.

5.2 $P_L \Rightarrow c$

Approaches in this category take in program descriptions in the form of a line-by-line fashion to produce the corresponding source code. Some approaches take in actual values of input or produce an output, but they fall into this category because they do not always require them compared to the category $P_{L\{i\}} \Rightarrow c_{\{o\}}$.

Many approaches in this category are also capable of generating fully runnable source code and they can generate in a general-purpose language [1, 2, 13, 41, 45]. However, most of them are restricted in their 'naturalness' of the input program description. Many approaches were in the early phase of this research field. They use techniques such as semantic parsing and information extraction. The reason that the input natural language descriptions were restricted is that these early techniques cannot fully capture the knowledge of natural language modalities.

5.2.1 NaturalJava: A natural language interface for programming in Java [1].

Price et al. [1] proposed an interface for creating Java programs with line-by-line natural language descriptions of a program. The system is composed of three distinct subsystems to produce Java code from natural language descriptions. First, they use a natural language processing system called Sundance, which takes in a natural language description of the program to extract information. With the extracted information, they generate case frames that represent the essentials of the program. Case frames are a syntactic representation of sentences and pattern-based templates used in Sundance. Second, the case frames are passed to a subsystem called PRISM, which is a knowledge-based case frame interpreter that interprets case frames to generate program abstract syntax tree (AST) of Java. Lastly, the generated ASTs are passed to a Java AST manager called TreeFace, which translates the ASTs to an actual Java source code.

5.2.2 Framework for creating natural language user interface for action-based applications [40].

Chong and Pucella [40] proposed a framework that creates interfaces of action-based applications with natural language descriptions. The natural language descriptions are mostly interactions of users and the system. Their main component of this approach uses type-logical grammar to translate the natural language description of a program into a higher-logical expression. The resulted logical expressions are then passed to the action interpreter, which executes the corresponding action calls on the target application.

5.2.3 Spoken Programs (Spoken Java) [41].

Begel et al. [41] proposed a voice programming interface that receives spelling word (speech), natural language, or paraphrasing text, to produce the corresponding Java program. This approach has especially dealt with ambiguity in the context of speech recognition such as homophones. However, ambiguity still resides in the point of written natural language.

5.2.4 Programming with unrestricted natural language [2].

Vadas and Curran [2] proposed a natural language interface for programming. With unrestricted syntax, they used wide-coverage syntactic and semantic methods to extract information from the natural language description. It uses a combinatorial grammar parser to get the syntax of the natural language description. Although the input may take in unrestricted forms of description, the translation is done in a line-by-line fashion. For evaluation, they did a study of how people gave programming instructions in natural language. They found that most people preferred using programming terms than those of simple natural language.

5.2.5 Pegasus - First steps toward a naturalistic programming language [13].

Knöll and Mezini [13] proposed a programming language that reads structured line-by-line natural language (in English, German, and Arabic [42]) and produces the corresponding program in Java. The basic features of Pegasus consist of reading natural language, generating source code, and expressing natural language. When reading natural language, it extracts keywords that have logical meaning such as *if* or *then*. With these keywords, it can extract the location of each statement and command clause. These pieces of information are then stored in a storage, which the authors define as *the brain*, in the form of an idea notation. The idea notation is a data structure defined to keep the syntax and the semantics of the natural language description. For the generation of source code, Pegasus uses the meaning-library, which is a database defined to match an idea notation and the corresponding Java commands. Pegasus can also express the program in natural language. Since the input programs are stored in *the brain* in the form of the idea notation, the system can backtrack to input natural language descriptions from the idea notation they have stored.

5.2.6 Natural language programming of a multiplayer online game [43].

Leiberman and Ahmad [43] proposed MOOIDE for creating MOO with a natural language description of the program. MOO is a text-based, multiplayer, online, virtual reality system. MOOIDE analyzes the natural language description using natural language parser to capture information. It also has an interacting mechanism to add new elements that the user wants to elaborate in the simulated world. The natural language parser uses anaphora resolution and common-sense knowledge to guarantee that objects behave as intended. The user can simulate or create virtual space using MOOIDE with typing in program descriptions as in Fig. 3.

```
There is a chicken in the kitchen.
There is a microwave oven.
You can only cook food in an oven.
When you cook food in the oven, if the food
is hot, say "The food is already hot."
Otherwise make it hot.
```

Fig. 3. MOOIDE program description examples [43].

5.2.7 SmartSynth: Synthesizing smartphone automation scripts from natural language [44].

Le et al. [44] proposed a smartphone script generation tool that processes natural language description of a smartphone program. The system is designed to program on smartphones with various platforms using natural language. The synthesis algorithm uses natural language processing to identify different components and their dataflow relations. It also uses type-based program synthesis to infer other missing dataflow and generates the script from reverse parsing. While it processes the description, SmartSynth interacts with the user to resolve the ambiguity or unknown elements not specified in the input description.

5.2.8 NLCI: A natural language command interpreter [45].

Landhäußer et al. [45] proposed a natural language command interpreter that takes in natural language description of a program and produces source code relevant API calls based on the ontology. The construction of the ontology can be automated if APIs use descriptive names for their components. NLCI is a domain agnostic because the domain knowledge can be changed and fine-tuned by changing the ontology before code generation. To check this attribute, they tested their system on two very different domains. First, they tested on Alice which is a 3D tool designed to teach children to create animation programs. Second, they tested on openHAB which is an API for home automation.

5.3 $P_A \Rightarrow s$

Approaches in this category take in program description that is more abstract and ‘natural’. The generated source code, however, is in the form of a code snippet, rather than fully runnable source code contrast to the categories that take P_L as 5.1 and 5.2.

Also, they can generate source code in general-purpose languages [8, 10, 15, 17, 46, 52, 53]. However, these approaches cannot generate fully runnable programs as ones from the approaches in Categories 5.1, and 5.2. because many of these approaches use machine learning techniques to generate source codes [8, 10, 15-17, 51, 52]. These techniques suffer in generating sound and complete code due to *loss* and *error*.

While natural language modalities are very robust in these *losses*, source codes in programming languages are very much affected by them. Still, this field of study is promising as quality and quantity data are piling up in software repositories.

5.3.1 Bimodal modeling of source code and natural language [8].

Allamanis et al. [8] proposed bimodal modeling of natural language and source code. This literature is the first generative model to apply a neural language model on both source code and natural language. The model uses aggregated datasets of natural language and source code. They experimented on using additive representation and the element-wise multiplicative representation in aggregating the datasets. They proved that element-wise multiplicative representation of data works better than the additive data. With the aggregated data, they used a neural language model called the log-bilinear model to train the aggregated data. Since their model is bimodal, they can produce source code from natural language descriptions, or they can generate natural language descriptions from source code. Their results indicate that generating source code from natural language is much more difficult than generating natural language from source code. This is because the generation of natural language is a much more robust task while the generation of source code consists of many obstacles such as considering the strict syntax of a programming language. The target programming language they used for the dataset was C#. They evaluated data from Stack Overflow, Dot Net Perls and have achieved 0.26 MMR average.

5.3.2 Synthesizing Java expressions from free-form queries [46].

Gvero and Kuncak [46] proposed a code assistance tool for developing Java code. The system takes in a free-form query that can contain both natural language and code modalities to produce candidates of corresponding Java code expressions. The tool is composed of the following subsystems: 1) a customized natural language processing tool for information extraction, 2) a matching algorithm for connecting queries with code ingredients, 3) probabilistic context-free grammar (PCFG) models trained with Java corpus, and 4) an algorithm to generate Java expressions using the artifacts produced from other subsystems.

5.3.3 Learning semantic parsers for IFTTT (if-this-then-that) recipes [47].

Quirk et al. [47] proposed a semantic parser that maps natural language description to an IFTTT recipe. IFTTT is widely used for web services to create chains of simple conditional statements. These chains trigger events for controlling web applications such as Gmail, Facebook, and Instagram. They trained a log-linear model in character-level with the n-gram features. They used a large corpus of IFTTT recipes aggregated with their natural language descriptions.

5.3.4 SWIM: Synthesizing what I mean [48].

Rahothaman et al. [48] proposed a system that translates free-form user queries into the APIs of interest and their corresponding code snippet. First, the natural language query is mapped to the API of interest. Second, the system retrieves a structured call sequence and its usage patterns of the target API. Last, they generate idiomatic code snippets from the structured call sequence. The query does not have to contain API specific keywords to generate the idiomatic snippet.

5.3.5 Latent predictor networks for code generation [16].

Ling et al. [16] proposed a neural architecture called Latent Predictor Networks that marginalizes multiple predictors for efficient training and scalable generation of source code. The advantage of marginalization is that it can choose different contexts for training and the granularity of generated code. They used source code and natural language data from two card games *Magic the Gathering* and *Hearthstone* and Django. They also integrated an attention method to handle structured input sequences. The resulted BLEU and accuracy scores averaged 68.7 and 24.4 respectively.

5.3.6 Program synthesis using natural language [10].

Desai et al. [10] proposed a program synthesizing framework that takes in a natural language description of a program and a training dataset to generate the corresponding source code. The training dataset consists of text-code pairs of a domain-specific language and the corresponding description. With

the input dataset, the system learns the language and generates a code snippet that matches the description of the program. The system learns any language from the input dataset enabling the system to be language agnostic. If the user can provide any programmable code and text pairs, the system can generate the corresponding program of code. They evaluated their method on the Air Travel Information System (ATIS), Automata Theory Tutoring, and Repetitive Text Editing and correctly translated in top3 ranks with the average percentage of 91.1%.

5.3.7 A syntactic neural model for general-purpose code generations [17].

Yin and Neubig [17] proposed a method to parse natural language descriptions to generate Python code snippets. They modeled neural architecture that uses a probabilistic grammar model to explicitly capture the syntax of the programming language as prior knowledge. They also found that this approach is effective in scalability when generating complex programs. They stated that this approach outperformed many code generations approaches that use semantic parsing. They evaluated *Hearthstone*, Django, and IFTTT and resulted in BLEU and accuracy score averaging 80.2 and 43.9 respectively.

5.3.8 Natural language to shell commands [49, 50].

Lin et al. [49, 50] proposed a system that translates a natural language description to generate the corresponding shell command. The translation is done by using recurrent neural networks (RNNs) and semantic parsing. When the input natural language description is read, the descriptions are preprocessed with semantic parsing technique called named entity recognition (NER) to capture the details of the tokens. Then, the tokens are read into the RNN encoder-decoder model that is used to translate a natural language template to a program template. They used the nearest neighbor clustering to evaluate the compatibility of an entity that is generated. The measured compatibility is used to cluster commands in each group and the arguments are filled accordingly to the generated commands.

5.3.9 Seq2SQL: Generating structured queries from natural language using reinforcement learning [51].

Zhong et al. [51] proposed a model that generates a structured query from a natural language using reinforcement learning. The natural language description, in the form of questions, is translated to corresponding SQL queries. The model uses the mixed objective of reward weights and cross-entropy loss to train executions of the database and learn policies to generate conditions of SQL. This approach leverages the structure of SQL to limit the range of generated queries to simplify the generation problem. They experimented on WikiSQL and averaged 54.5 in accuracy scores.

5.3.10 Augmenting and structuring user queries to support efficient free-form code search (COCABU) [52].

Sirres et al. [52] proposed a free-form search engine that resolves the vocabulary mismatch problem. With natural language or Java expressions, they augment the query to resolve the vocabulary mismatch problem and finds code examples that have high relevance from software repositories such as GitHub and StackOverflow. This approach does not generate source code, but it retrieves code snippets that are fully functional because it finds code snippets from software repositories. The downside, however, is that it cannot produce or generate source code that is new or not existing in the software mentioned software repositories.

5.3.11 Deep Code Search [15].

Gu et al. [15] proposed a deep neural network and a code search tool that takes in a natural language description of a snippet for retrieving the corresponding code snippet. The deep neural network, CODEnn (abbreviation for COde Description Embedding Neural Network), is trained to capture the semantic similarities of natural language description and the code snippet. The two different modalities are trained and embedded into unified vectors. When a code snippet and a description are semantically similar, the embedded vectors will be close to each other. With this model, they implemented a deep learning-based code search tool, DeepCS, and evaluated their approach. DeepCS recommends top K most relevant code snippets from a natural language description. They evaluated on Java corpus of 18M methods from GitHub and overall, they averaged MRR of 0.60.

5.3.12 Vajra: Step-by-step programming with natural language [53].

Schlegel et al. [53] proposed an end-user programming paradigm for Python. Vajra takes natural language descriptions and generates corresponding Python code snippets. The user types in natural language command to a specific spot in source code. Then, their system generates a list of possible statements and their associated parameters that are most similar in semantics. There are procedures that the user can choose to resolve ambiguity in the process by clicking from multiple candidate snippets. The core technique used in this study is semantic parsing.

5.4 $P_A \Rightarrow m$

Approaches in this category take an abstract natural language description of a program to generate a corresponding intermediate program form. As stated in Section 4, intermediate code needs secondary work to be processed before execution, i.e. implementation for a skeleton program, a compilation for medium (assembly) language, and code integration for a regular expression.

These approaches differ from other categories in that they generate what we define as intermediate code. They are not necessarily in the form of source code but if they are, they are in the form of abstraction of source code i.e. a skeleton code that is not runnable. We added these approaches in this category because, even though they are not in the form of programming language or fully runnable codes, they receive natural language modalities and generates an output of a source code modalities that helps developers develop programs.

5.4.1 Metafor: Visualizing stories as code [54].

Lie and Lieberman [54] proposed a program editor that uses descriptive statements about a program to create scaffolding code fragments that can be used for the designers and developers. The system interacts with the user and uses the dialogue information for disambiguation. The system captures different objects, functions, and descriptions and makes those pieces of information as class abstraction and generates a skeleton program code in python language.

5.4.2 CPL & CPL-lite [55, 56].

Clark et al. [55, 56] proposed CPL which stands for Computer-Processable Language. CPL receives natural language descriptions and generates an intermediate representation that restricts the descriptions to a subset of natural language so that both humans and computers can understand better than the two extreme languages, i.e. programming and natural languages. CPL uses heuristics to resolve the ambiguity in the natural language description. They have three types of sentence input: facts, questions, and rules. In CPL-lite, they added a mechanism to define queries in a comprehensive and controllable way. CPL-lite does not use heuristics but a more restricted interpreter to handle the ambiguity. The form that is generated is a program called knowledge machine (KM) which they proposed in [12]. KM is a mature, advanced, frame-based language with well-defined semantics, used previously in several major knowledge representation projects.

5.4.3 Automatic programming with natural language compiler [57].

Somasundaram and Swaminathan [57] proposed a compiler that parses the natural language description of a program to generate intermediate representation to help the compiler to convert them into the target language with minimal effort. They aim to reduce ambiguity by parsing through the natural language descriptions of a problem statement and generate the corresponding object-oriented program. The key component of their approach consists of a syntactic analyzer, symbol table, lexical analyzer, semantic analyzer, intermediate code generator, and a code generator.

5.4.4 Assisted behavior-driven development using natural language processing [58].

Soeken et al. [58] proposed an assisted flow for Behavior Driven Development (BDD) where the user provides an acceptance test composed of natural language dialogue with the computer about code pieces. The system extracts code information using natural language processing techniques from the dialogue to produce skeleton code.

5.4.5 Using semantic unification to regular expression from natural language [59].

Kushman and Barzilay [59] proposed a translator that takes free-form queries and performs a semantic unification to generate the corresponding regular expression. The introduced ambiguity from the two modalities differ, but regular expression also has multiple representations of the equivalent expressions. The author exploits this flexibility to facilitate translation by finding a form that is more similar to the natural language. They evaluated their technique on a set of natural language queries and their corresponding regular expressions gathered from Amazon Mechanical Turk [60].

5.5 $P_{A\{t\}} \Rightarrow c$

Approaches in this category receive an abstract natural language description of code and a set of unit tests (example inputs and their oracles), to generate fully runnable code.

They can handle abstract or 'natural' description and requires unit tests as a requirement can improve the ambiguity from the 'naturalness' of the description. Using unit tests enables them to handle abstract descriptions and generate fully runnable source code in a general-purpose language.

5.5.1 Macho: Programming with man pages [14, 61].

Cozzie et al. [14, 61] proposed a tool that generates source code from receiving abstract natural language and a unit test that has one or more examples with correct input and oracle of the program. In an abstract natural language description, there is always the challenge to resolve the ambiguity. Macho first parses the abstract natural language and create multiple candidate programs due to the ambiguity. Then, it checks the candidates with the unit test to see which fits the best. When presenting the candidate programs, Macho uses a raking system by using a probabilistic model. The project was trained on a large database of open-source Java codes.

5.5.2 Integrating programming by example and natural language programming [62].

Manshadi et al. [62] proposed a system that generates source code from a natural language description. The system receives unconstrained instructions and one or more input and output examples. This approach differs from the others because they use the technique of version space algebra [63]. This technique is used to decompose a problem into simpler problems that can be individually solved to reduce the complexity of problem-solving. They also use the method of probabilistic programming by example (PPbE) to reduce the number of possible solutions.

6. Discussion

As stated in the introduction, it is clear that code generation from natural language has potential and is promising in software engineering practice. The followings are our technical and trend analysis, current challenges, and future directions after we have organized the survey.

Our overview of the observation is depicted in Fig. 4. The figure shows the position of categories in their relation to the abstraction of description and completeness of source code. The categories can be grouped into 4 regions:

(1) Group 1 is the upper-left region. They have high completeness of source code, but they can handle the lowest abstraction of description

(2) Group 2 is in the upper-middle region. They also have high completeness of source code and have a medium abstraction level is that having test cases, *t*, as a requirement hurts the abstraction or 'naturalness' of the high abstraction of $_{A}$.

(3) Group 3 is in the lower-right region. They have high abstraction or 'natural' description, but they generate source code of low completeness.

(4) Group 4 is in the top-right region where researchers should aim for a future goal. This is where approaches generate source code with high completeness that can also handle abstract and 'natural' descriptions.

6.1 Technical Analysis

Approaches in category $P_{L\{i\}} \Rightarrow c_{\{o\}}$ are implemented by parsing the natural language description and translating the description through hard-coded grammar to generate fully runnable source code. This

could be done by hard coding because the number of possible grammars is limited by the 'naturalness' of the natural language description and the specific domain of the programming language. By limiting the two axes ('naturalness' and domain) in the search space, they were able to implement the grammar that generates fully runnable source code.

Approaches in category $P_L \Rightarrow c$ are also implemented by parsing natural language descriptions. However, they expanded the axis of the domain-specific program to generate a general-purpose program [1, 2, 13, 41, 45]. Due to the expansion of the search space, they implemented additional disambiguating grammars to correctly map the natural language description with the correct source code.

Approaches in category $P_A \Rightarrow s$ expand the search space in both axes to achieve 'naturalness' and general-purpose program [8, 10, 15, 17, 46, 52, 53]. Approaches that can handle abstract natural language to generate general-purpose language use probabilistic language models or machine/deep learning models [8, 15-17, 51, 52]. Due to the techniques' inevitable loss and error, the generated source code's completeness is reduced to the snippet level. Despite reducing the space of completeness, the approach's performance is still very low.

Approaches in category $P_A \Rightarrow m$ are implemented by parsing the abstract natural language description and generates an intermediate level of source code [55-57]. Some approaches reduce the axis by generating domain-specific language such as regular expression [59]. Other approaches generate a general programming language of a skeleton program [54, 58].

Approaches in category $P_{A\{t\}} \Rightarrow c$ are implemented by parsing the abstract natural language description to generate a complete general-purpose program. Even though the search space is large, the approaches effectively find the target source code by exploiting the provided oracles. However, providing the right number of effective oracles is difficult and cannot be done by the system.

6.2 Trend Analysis and Current Challenges

From surveying the approaches that generate source code from natural language, we could see the current research trends and challenges in this research field.

First, approaches that require or produce certain types of actual input or output forms [11, 12, 37-39] tend towards being domain-specific. It is somewhat obvious because approaches with certain input or output forms imply that the program should be bound to them. All approaches in category $P_{L\{i\}} \Rightarrow c_{\{o\}}$ are domain-specific. $P_{A\{t\}} \Rightarrow c$ does not fall into this tendency because input forms *'t'* are test cases, which are not bound to any specific type of input forms. They are just an example of an input/output oracles.

Second, former studies in this field are implemented with techniques such as semantic parsing to find and extract information from the natural language descriptions. With the information extracted, they corresponded to them with certain codes with similar keywords. By using such techniques, the code generation was done in a line-by-line translating fashion. This means that every specific detail of the source code has to be given in the description. This weakens the abstraction and 'naturalness' of program descriptions. These aspects can be observed from categories such as $P_{L\{i\}} \Rightarrow c_{\{o\}}$, and $P_L \Rightarrow c$. These categories are depicted in the upper-left part of Fig. 4 where the approaches generate fully runnable code but lack abstraction or their 'naturalness' in handling their natural language descriptions.

Last, the following studies tried to resolve the restrictions of abstraction and 'naturalness' by using neural networks and probabilistic language models [8, 10, 15-17, 51, 52]. With abundant source code data in software repositories, these techniques have been studied actively. By using machine learning techniques, it resolved the restriction of handling 'natural' program descriptions and domain-specific programming languages. It also made possible to generate source code even if the details of the program description are missing. It was possible to infer from the abstract description of the program. However, due to the inevitable *loss* and *error* that arouse in machine learning techniques, it introduced new challenges. With the *loss* and *error*, the generated source code was syntactically unsound. To improve the soundness of the generated code, researchers narrowed down the range of code to be generated, from completely runnable code to code snippet. By reducing the range of generation, helped approaches to produce more sound algorithms. However, the performance of generating sound source code is still poor [8, 15-17, 51]. Researchers in this area have to work on building models that are more sound and complete for these models to be used in practice. Another point is that these models are originally built for natural language. Although programming language and natural language have similar characteristics, the difference still does exist, leading to code generative models to perform poorly. These tendencies can be

found by looking at categories: $P_A \Rightarrow s$, and $P_A \Rightarrow m$. These categories are depicted in the lower-right part of Fig. 4, where the approaches do not have restrictions in handling their natural language descriptions but lack completeness in their generated source code of a program.

From looking at the characteristics of each category and their release dates, we could see the current trend of this research field. The approaches started from handling descriptions with a low level of abstraction (P_L) to a higher abstraction of natural language descriptions with supplementary information ($P_{A\{t\}}$) then finally to approaches with high abstraction without additional input (P_A). This flow can be seen in Fig. 4, moving from Group 1 → Group 2 → Group 3.

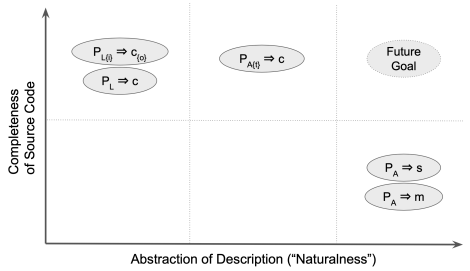

Fig. 4. The position of categories respect to the abstraction of description and the completeness of produced source code.

6.3 Future Directions

As stated in 6.1, the current direction of this research field is going from handling descriptions with a low level of abstraction to handling descriptions with higher abstraction. From handling the 'naturalness' of descriptions, we have lost the completeness of generated source code. To ensure the usability of code generation with natural language approaches, we need to enhance the ability to generate complete source code while maintaining the abstraction of description. To do this, we suggest two important directions to follow.

First, we need to keep working on using probabilistic language models because using these approaches will enable us to keep freedom in terms of using 'natural' descriptions and domain independence. By using these techniques, new code can be generated from the descriptions while approaches using semantic parsing in a line-by-line fashion will require every information of code to be in the descriptions. This means that we should move from Group 3 to Group 4, not from Group 1 or 2 to Group 4 in Fig. 4.

Second, to resolve the issues that probabilistic language models and deep neural networks face, more 'source code' focused probabilistic language models should be exploited. Designing new model architectures and other tuning techniques (i.e. optimizers and regularizers) in the source code domain will enhance the performance of models because the models we use now is for natural language. There are distinct features that programming language have, but natural languages do not. The features are mostly very formal. Using the formality of programming language as specialized features as an objective function will help achieve better performance. By fine-tuning these techniques to source code domain, it will help the model fit within the correct syntax. Thus, it will be able to generate a much more complete source code. Another point is that these techniques receive a very abstract form of descriptions, having much more ambiguous descriptions. Ambiguity resolution is more crucial in the approaches that use probabilistic language models than approaches using semantic parsing techniques, but they never tend to do so. Approaches with P_L have focused on many disambiguating technologies that have a 'natural' interface such as asking questions to users. The ambiguity resolving techniques exploited in the approaches with P_L should be applied to probabilistic models for disambiguation to generate more complete program code. Another practical improvement can be made by studying the better representation of source code. The state-of-the-art benchmarks of NLP techniques show that fine-tuning deep pre-trained models improve most of the tasks [64-66]. However, exploiting the representation of large pre-trained models on source code corpora is currently unknown.

For future work, we are interested in applying machine and deep learning models directly on source code corpora to learn their features and find supporting knowledge to assist in software engineering practice. One application could be applying neural network models on Javadocs and their related code to

generate source code from natural language program descriptions. We need to learn and capture more raw knowledge of source code corpora to build complete generative models of source code. A second application could be using an issue report in the form of natural language to automatically generate patch source code. Currently, patch generation involves intermediate processes such as generating test cases and applying them to generate patches. By predicting patches from the issue report, we can evade these intermediate processes. Another topic for future work will be integrating a conversation mechanism to probabilistic language models. When the natural language description is too broad or ambiguous, the system will ask users to elaborate on the description for disambiguation. Conversing programs, shown from categories with P_L works well as a disambiguation technique and have the most 'natural' way of working. After capturing the knowledge of different features of source code and natural language and handling ambiguity, there will come a time when developers can program better from natural language. In a very distant future, we could communicate with computers by our natural languages as the famous *Jarvis* system introduced in the movie, *Iron Man*.

7. Conclusion

In this paper, we have surveyed and reviewed the different approaches of automatic code generation with natural language descriptions. After listing the approaches, we categorized them by their structure and analyzed the technical issues and the current trend according to their categories. In trend analysis, we have found that in former studies [1, 2, 11-13, 40], researchers focused on generating complete code by sacrificing either in the 'naturalness' of natural language descriptions or the domain of generated source code. Following studies focused on resolving the restrictions by exploiting machine/deep learning techniques and statistical language models. Using these techniques, recent studies resolved both restrictions [8, 10, 15, 17, 46, 52, 53], but they sacrificed the completeness of the generated source code. However, these techniques show promising potential as we live in the era of flourishing data of source code and rooms for improvement lie in applying language models with source code data. One practical improvement we suggest is exploiting a deeply pre-trained representation of source code corpora and fine-tuning to different tasks. The research field of NLP showed that this approach has significantly improved many different tasks. By resolving these issues and keeping the freedom of domain and 'naturalness', we believe this research field will bring a new paradigm to the software engineering discipline.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (NO.2018R1C1B6001919).

References

[1] Price, D., Rilofff, E., Zachary, J., & Harvey, B. (2000, January). NaturalJava: a natural language interface for programming in Java. In *Proceedings of the 5th international conference on Intelligent user interfaces* (pp. 207-211). ACM.

[2] Vadas, D., & Curran, J. R. (2005, December). Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005* (pp. 191-199).

[3] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 81.

[4] Neubig, G., & Allamanis, M. (2018, June). Modelling Natural Language, Programs, and their Intersection. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorial Abstracts* (pp. 1-3).

[5] Pulido-Prieto, O., & Juárez-Martínez, U. (2017). A survey of naturalistic programming technologies. *ACM Computing Surveys (CSUR)*, *50*(5), 70.

[6] White, G. L., & Sivitanides, M. P. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education*, *13*(1), 59-68.

[7] Ghezzi, C., & Mandrioli, D. (2005, May). The challenges of software engineering education. In *International Conference on Software Engineering* (pp. 115-127). Springer, Berlin, Heidelberg.

[8] Allamanis, M., Tarlow, D., Gordon, A., & Wei, Y. (2015, June). Bimodal modelling of source code and natural language. In *International conference on machine learning* (pp. 2123-2132).

[9] Clark, P., Porter, B., & Works, B. P. (2004). Km–the knowledge machine 2.0: User's manual. *Department of Computer Science, University of Texas at Austin*, *2*(5).
[10] Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., & Roy, S. (2016, May). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering* (pp. 345-356). ACM.
[11] Heidorn, G. E. (1973, January). An interactive simulation programming system which converses in English. In *Proceedings of the 6th conference on Winter simulation* (pp. 781-794). ACM.
[12] Little, G., & Miller, R. C. (2006, October). Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology* (pp. 135-144). ACM.
[13] Knöll, R., & Mezini, M. (2006, October). Pegasus: first steps toward a naturalistic programming language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (pp. 542-559). ACM.
[14] Cozzie, A., Finnicum, M., & King, S. T. (2011, May). Macho: Programming with Man Pages. In *HotOS*.
[15] Gu, X., Zhang, H., & Kim, S. (2018, May). Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (pp. 933-944). IEEE.
[16] Ling, W., Grefenstette, E., Hermann, K. M., Kočiský, T., Senior, A., Wang, F., & Blunsom, P. (2016). Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.
[17] Yin, P., & Neubig, G. (2017). A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.
[18] Song, X., Sun, H., Wang, X., & Yan, J. (2019). A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, *7*, 111411-111428.
[19] Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2015, August). Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 38-49). ACM.
[20] Bavishi, R., Pradel, M., & Sen, K. (2018). Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*.
[21] Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to represent programs with graphs." *arXiv preprint arXiv:1711.00740* (2017).
[22] Chae, K., Oh, H., Heo, K., & Yang, H. (2017). Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the ACM on Programming Languages*, *1*(OOPSLA), 101.
[23] Gupta, R., Kanade, A., & Shevade, S. (2018). Deep reinforcement learning for programming language correction. *arXiv preprint arXiv:1801.10467*.
[24] Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017, February). Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.
[25] Hu, X., Wei, Y., Li, G., & Jin, Z. (2017). CodeSum: Translate program language to natural language. *arXiv preprint arXiv:1708.01837*.
[26] Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016, August). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 2073-2083).
[27] Gu, X., Zhang, H., Zhang, D., & Kim, S. (2016, November). Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 631-642). ACM.
[28] Movshovitz-Attias, D., & Cohen, W. W. (2013). Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Vol. 2, pp. 35-40).
[29] Fowkes, J., & Sutton, C. (2016). Parameter-free probabilistic API mining at github scale. In *Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering (FSE'16). ACM*.
[30] Nguyen, T. D., Nguyen, A. T., Phan, H. D., & Nguyen, T. N. (2017, May). Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp. 438-449). IEEE.
[31] Allamanis, M., & Sutton, C. (2014, November). Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 472-483). ACM.

[32] Murali, V., Chaudhuri, S., & Jermaine, C. (2017, August). Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 151-162). ACM.
[33] Wang, S., Liu, T., & Tan, L. (2016, May). Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 297-308). IEEE.
[34] White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016, August). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (pp. 87-98). ACM.
[35] Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., & Sutton, C. (2017). Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, *43*(12), 1095-1109.
[36] Schriber, T. J. (1974). *Simulation using GPSS*. MICHIGAN UNIV ANN ARBOR.
[37] Gulwani, S., & Marron, M. (2014, June). 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* (pp. 803-814). ACM.
[38] Shi, S., Wang, Y., Lin, C. Y., Liu, X., & Rui, Y. (2015, September). Automatically solving number word problems by semantic parsing and reasoning. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (pp. 1132-1142).
[39] Mandal, S., & Naskar, S. K. (2017, December). Natural Language Programing with Automatic Code Generation towards Solving Addition-Subtraction Word Problems. In *Proceedings of the 14th International Conference on Natural Language Processing (ICON-2017)* (pp. 146-154).
[40] Chong, S., & Pucella, R. (2004). A Framework for Creating Natural Language User Interfaces for Action-Based Applications. *arXiv preprint cs/0412065*.
[41] Begel, A., & Graham, S. L. (2005, September). Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)* (pp. 99-106). IEEE.
[42] Mefteh, M., Ben Hamadou, A., & Knöll, R. (2012). Ara_Pegasus: A new framework for programming using the Arabic natural language. In *International Conference on Computing and Information Technology (March 2012)* (pp. 468-473).
[43] Lieberman, H., & Ahmad, M. (2010). Knowing what you're talking about: Natural language programming of a multi-player online game. In *No Code Required* (pp. 331-343). Morgan Kaufmann.
[44] Le, V., Gulwani, S., & Su, Z. (2013, June). Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services* (pp. 193-206). ACM.
[45] Landhäußer, M., Weigelt, S., & Tichy, W. F. (2017). NLCI: a natural language command interpreter. *Automated Software Engineering*, *24*(4), 839-861.
[46] Gvero, T., & Kuncak, V. (2015, October). Synthesizing Java expressions from free-form queries. In *Acm Sigplan Notices* (Vol. 50, No. 10, pp. 416-432). ACM.
[47] Quirk, C., Mooney, R., & Galley, M. (2015, July). 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)* (pp. 878-888).
[48] Raghothaman, M., Wei, Y., & Hamadi, Y. (2016, May). Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 357-367). IEEE.
[49] Lin, X. V., Wang, C., Pang, D., Vu, K., & Ernst, M. D. (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*.
[50] Lin, X. V., Wang, C., Zettlemoyer, L., & Ernst, M. D. (2018). NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*.
[51] Zhong, V., Xiong, C., & Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.
[52] Sirres, R., Bissyandé, T. F., Kim, D., Lo, D., Klein, J., Kim, K., & Le Traon, Y. (2018). Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*, *23*(5), 2622-2654.
[53] Schlegel, V., Lang, B., Handschuh, S., & Freitas, A. (2019, March). Vajra: step-by-step programming with natural language. In *IUI* (pp. 30-39).
[54] Liu, H., & Lieberman, H. (2005, January). Metafor: Visualizing stories as code. In *Proceedings of the 10th international conference on Intelligent user interfaces* (pp. 305-307). ACM.

[55] Clark, P., Harrison, P., Jenkins, T., Thompson, J. A., & Wojcik, R. H. (2005, May). Acquiring and Using World Knowledge Using a Restricted Subset of English. In *Flairs conference* (pp. 506-511).
[56] Clark, P., Murray, W. R., Harrison, P., & Thompson, J. (2009, June). Naturalness vs. predictability: A key debate in controlled languages. In *International Workshop on Controlled Natural Language* (pp. 65-81). Springer, Berlin, Heidelberg.
[57] Somasundaram, K., & Swaminathan, H. (2011). Automatic Programming through Natural Language Compiler. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)* (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).
[58] Soeken, M., Wille, R., & Drechsler, R. (2012, May). Assisted behavior driven development using natural language processing. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (pp. 269-287). Springer, Berlin, Heidelberg.
[59] Kushman, N., & Barzilay, R. (2013, June). Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 826-836).
[60] Harinarayan, V., Rajaraman, A., & Ranganathan, A. (2007). *U.S. Patent No. 7,197,459*. Washington, DC: U.S. Patent and Trademark Office.
[61] Cozzie, Anthony E., and Samuel King. "Macho: Writing programs with natural language and examples." (2012).
[62] Manshadi, M. H., Gildea, D., & Allen, J. F. (2013, June). Integrating programming by example and natural language programming. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*.
[63] Mitchell, T. M. (1982). Generalization as search. *Artificial intelligence*, *18*(2), 203-226.
[64] Howard, J., & Ruder, S. (2018, July). Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 328-339).
[65] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, June). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (pp. 4171-4186).
[66] Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018, June). Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 2227-2237).

Jiho Shin https://orcid.org/0000-0001-8829-3773 (ORCID ID)
He received B.S. degree in computer science from Handong Global University in 2019. Since March 2019, he is with Information and Communication Engineering from Handong Global University as a M.S. Student. His current research interests include defect prediction, automatic patch generation and NLP.

Jaechang Nam https://orcid.org/0000-0003-1678-2185 (ORCID ID)
He received B.S. degree in computer science from Handong Global University in 2002, M.S. in computer science from Blekinge Institute of Technology in 2009, and Ph.D. in computer science and engineering from The Hong Kong University of Science and Technology in 2015. He is currently an assistant professor in the Department of Computer Science, Handong Global University, Pohang, Korea. His research interests include mining software repository (MSR), software quality prediction, transfer leaning in software engineering.