A Survey on Machine Programming Techniques

oIn this survey we review and classify the research on machine programming. We are specifically interested in the research that takes a description of code and generates software. We introduce a taxonomy with 5 dimensions and use it to classify the existing research. For each class, we look at the dominantly used approaches and go over several examples of using them in machine programming.

ACM Reference Format:
. 2020. A Survey on Machine Programming Techniques. 1, 1 (December 2020), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this survey, we review the recent research on machine programming approaches. We are specifically interested in cases where the output of the approach is software. We use the term "software" to cover any of the three possibilities: code that can be verified by humans that it is almost correct, code that is compilable, or code that is executable. Creating tools and coming up with techniques that can assist in generating software is important because generating software is a complex task [7] and can result in high cost, long duration of completion, low quality, and unrealized or misinterpreted requirements [83]. The complexity that has to be dealt with is either related to the software itself or the process by which software is defined and implemented. From the former perspective, complexity arises in different forms like entropy defined by the number of state variables and different values each can take [31]; coupling defined as the degree of interdependence among states [3]; adaptability, being able to support users with different abilities, e.g. support for disabled users; and resiliency, being able to work under different external circumstances, e.g. low internet speed. From the latter perspective, the complexity arises from the ambiguity and incompleteness inherent in the informal language used to specify the requirements [10], the fluidity of requirements that makes them subject to frequent revisions over the project's lifetime [50], and lastly due to the information lost in communicating or understanding requirements among different project stakeholders [18].

In general, the automated code and artifact generation research has either been focused on taking non-code artifacts and turning them into code or vice versa. Since code is well-structured whereas non-code artifacts are not, the transformation process from one to the other is asymmetric. In general, transforming from non-code to code is much more challenging not only from the research perspective - which requires relying on an expanding set of techniques including machine learning [1], natural-language processing [91], and heuristics but also from the practical perspective where such tools that can take a verbal description of the code and turn it into executable application or system are almost non-existent.

The research papers we review in this survey belong to the more challenging non-code to code transformation also known as machine programming. They take a description of the code (which could be textual description, examples, or sometimes code itself) and convert them into different types of the software mentioned earlier. The

Author's address:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

automated transformation of different kinds of software models to code has been a subject of study for several decades. This is an extremely challenging problem and a range of techniques - that we will review in this survey - have attempted to approach it using different strategies and making different simplifying assumptions. Recently there have already been several surveys of this area [1]. These surveys are normally technique-centric rather than input-output-centric, i.e., they use the underlying technique to classify the surveyed research regardless of what types of input these techniques are applied to, what types of output are generated, or what problems they are trying to solve.

More traditional techniques such as synthesis based formal logic and version-based algebra [29], inductive programming [40], and fuzzy logic [57] were reviewed in detail in prior surveys. While we include them in our taxonomy for completeness, we will focus more closely on the work that has emerged in the past decade as it has taken a very different approach to code generation. However, we still have to cover the entire space of code generation in the taxonomy with less focus on them when we review the related research since one objective of our taxonomy is comprehensiveness.

The first work in automated code generation dates back to the 1950s when Church introduced the problem of synthesizing an electronic circuit to realize a mathematical function which became the origin for research on combinatorial logic [86]. This was followed by the inductive programming approaches like first-order logic that could generate programs for formal languages in the 1960s [78]. In the 1970s inductive programming addressed learning of typically declarative or functional logic and often recursive programs from incomplete specifications [84]. The 1980's focus was on automation using patterns and reusability [11]. In the 1990s, we see the rise of fuzzy logic to capture uncertainties and the onset of using AI to generate code [87]. In the 2000s the focus was on model-driven approaches [55] and inductive programming synthesis [40]. Finally, the most recent work since 2010 has focused on using neural networks to translate input-output examples or natural language descriptions of code into code which frequently include machine learning as a major component in their techniques [1].

In what follows we start by introducing the taxonomy we use to categorize the reviewed research and then review the related work within each category.

2 OVERVIEW OF TAXONOMY

The techniques used to generate code and non-code artifacts significantly differ in the approach and complexity. On the one hand, we are transforming an unstructured expression of intent to an implementation in a structured programming language. On the other hand, we turn structured segments of code into "a programmatically-generated unstructured expression." We argue that given the complexity and applicable techniques to implement them, each group deserves to be studied, compared, and classified separately. However, note that the existing surveys and the accompanying taxonomies on tool-based artifact generation have been more involved with techniques than outcomes and therefore have not been able to provide a focused taxonomy on one side or another. For example, Allamanis et al build their survey on an important hypothesis that they call "the naturalness hypothesis" stating that "Software is a form of human communication; software corpora have similar statistical properties to natural language corpora, and these properties can be exploited to build better software engineering tools"; however, it is important to note that treating the naturalness is much more challenging when it is presented to us as the input rather than output. For example, consider the case where we are given this natural-language sentence: "There should be an alert when the driver drives fast," and told to convert it into executable code. Given the existing ambiguity and degrees of freedom in this rather simple sentence, we cannot use any elementary intelligent technique to turn the sentence into executable code. Whereas if we were given this line of code: "if (driver.speed > 10) alert.show('driver is going too fast')" and were asked to create a natural-language comment from this line of code, either a non-AI algorithm or a simple ML model would have worked. Relying on this

hypothesis has led to a comprehensive survey and taxonomy on techniques to address all kinds of naturalness and has made the taxonomy a bit less effective to provide a straightforward categorization.

2.1 Other Taxonomies

The taxonomy by [1] provides a detailed classification mechanism for the type of the technique, representation of non-code artifacts, representation of code, and the application. Although being very comprehensive in terms of covered techniques, their taxonomy does not differentiate between studies that use free-form textual non-code description and others that use code as the description. Another taxonomy provided in the survey by [29] includes a comprehensive taxonomy on programming synthesis for the input constraint, the program search space, and the synthesis techniques. However, it does not differentiate the studies that generate different types of code artifacts like functional code or unit tests. Another taxonomy is provided by [72] in which they categorize naturalistic programming technologies. They focus on the structural differences of the programs and use such characteristics as whether the program contains English-like expressiveness or indirect inferences. They offer a taxonomy on three broad application categories: being learning, industry, or reporting focused. However, compared to other surveys they pay less attention to the underlying techniques used to generate code based on these languages.

In this survey, we have decided to only consider studies that result in generating code. Therefore, we have been able to provide a more specific taxonomy based on input, outputs, processes, techniques, and solutions that deal with the complexity of generating code from any form of input used to explain the code intent.

2.2 Definition of Code Generation

Given that our taxonomy is focused on machine programming, it is important to precisely define machine programming. We define machine programming as rule-based or training-based techniques to turn a description of code Into code to solve a specific code-generation problem. Note that this definition of machine programming depends on five basic elements process, technique, type of input, type of output, and solution. Therefore, as shown in Figure 1, our taxonomy is founded on top of these five basic elements to categorize the surveyed research.

3 SECTIONS OF THE TAXONOMY

3.1 Types of Input

Input is the explanation of the desired intent of the software to be implemented. On one side of the spectrum, it could cover one or a few most expected scenarios in the shape of either natural language description or examples. On the other side, it could be a comprehensive set of formal or informal requirements covering both negative and positive scenarios.

3.1.1 Natural Description. The input is in the shape of natural language without any restrictions. Note that the following characteristics inherent in natural language introduce challenges to the code generation technique: qualitative ambiguity and contextual ambiguity.

3.1.2 Informal Requirements. This a set of requirements written in natural language by an analyst. Most of the software built in the industry is based on some sort of informal requirement. The informality of requirements due to using natural-language description to state them.

3.1.3 Formal Requirements. A formal software specification is a statement expressed in a language whose vocabulary, syntax and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.

3.1.4 Examples. Programming by examples (PBE) is a sub-field of program synthesis, where the specification comes in the form of input-output examples.

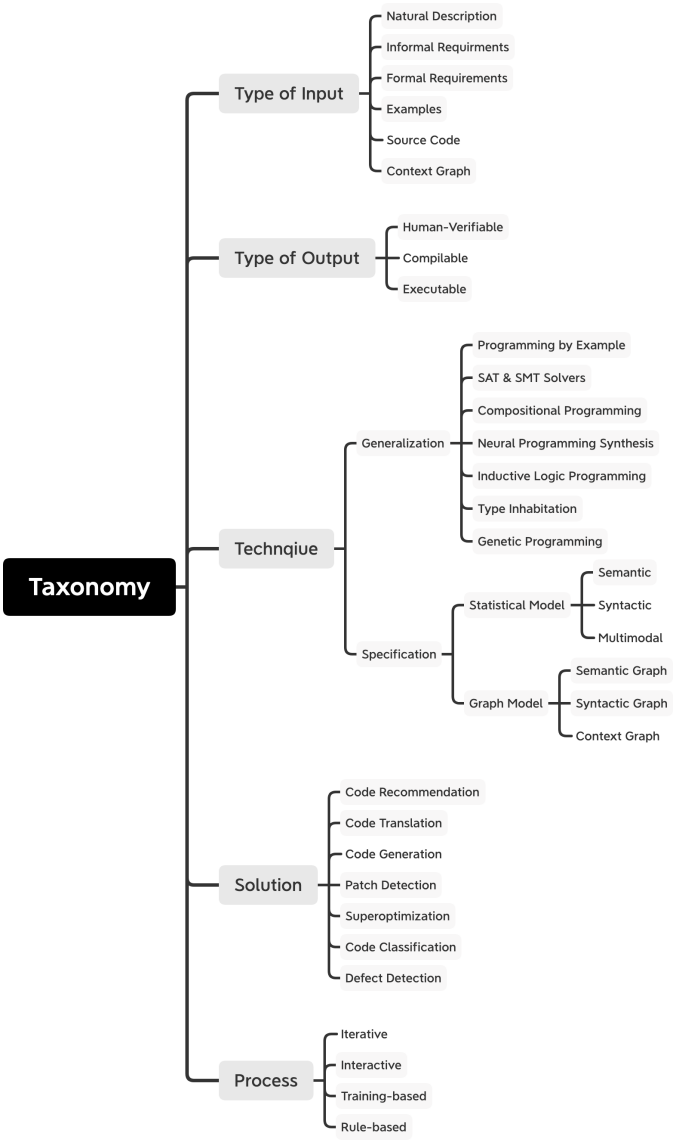


Fig. 1. Taxonomy

3.1.5 Source Code. Source code can be used as the input as well as output. For example, we can use source code as the input to generate source code in other languages or generate patches that can fix the bugs existing in the input source code.

3.1.6 Context Graph. User intent is the foundational information that drives what is included in the requirements used to document the software behavior. A small application might only have one user which can have several possible intents each applicable at a given state of the application. For larger applications there can exist more users each with several intents. These intents work together to define what we call a context graph for the application. The context graph defines how the application state changes based on the user intent. Recently there has been efforts to use a graph representation as the input to the code generation techniques with an anticipation that such representation would have a fundamental impact on the scalability and accuracy of the code generation techniques [65].

3.2 Types of Output

The existing research can be categorized into the following four groups in terms of their output. The list is sorted in terms of sophistication of the generated output, i.e., the generated output needs to go through a more rigorous validation process to be accepted.

3.2.1 Human Verifiable Code. Some of the code generation techniques generate a code segment that is not 100% valid in terms of its syntax; however, a human can provide a subjective validation asserting that the generated code is aligned with the correct implementation of the offered description.

3.2.2 Compilable Code. Some of the code generation techniques generate a code segment with a valid syntax - i.e., can be mapped to a valid AST by a compiler. However, the code might not be invoked or executed to generate the expected result.

3.2.3 Executable Code. Other techniques generate a code segment that can be invoked to create a result or used for testing other code known as unit tests. Begin able to produce the expected outcome or detect buggy code in case of a unit test can be used to validate the output.

3.3 Techniques

Code generation techniques also known as program synthesis techniques are, in general, defined as the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification [29]. However, many real-life application domains for program synthesis are too complex to be described entirely with formal or informal specifications. First, such a description would likely contain so many implementation details and special cases that it would be comparable in size to the produced program. Second, and most importantly, the users themselves often do not imagine the full scope of their intent until they begin an interaction with a programmer or a program synthesis system. Both of these observations imply that applying program synthesis to larger industrial applications is much a human-computer interaction (HCI) problem [62] as it is an algorithmic one. Therefore the terms programming synthesis is more specifically used to identify the algorithmic approaches.

There are two types of approaches to generating code: generalization and specialization. The former is based on generalization from examples or specifications to programs whereas the latter is based on specialization from existing general models to programs.

3.3.1 Generalization Techniques. Generalization techniques try to find an implementation that can produce an expected result given some examples or constraints regardless of how it is implemented. Therefore they can be described as a generalization from a set of examples and constraints to a general program that can produce such examples [40].

Programming by example (PBE): These techniques allow end-users to easily create programs by providing a few input-output examples to specify their intended task. The system attempts to generate a program in a

domain-specific language (DSL) that satisfies the given examples [76]. A related variation of PBE is programming by demonstration (PBD) In PBD, a human demonstrates a repetitive task in a few contexts; the machine then learns to perform the task in new contexts. From a learning perspective, the main difficulty with PBD is that it is only reasonable to expect one or two training examples from the user. PBE techniques try to learn from training sets of examples to DSL mappings. A major component in PBE synthesis techniques is the design of the domain-specific language (DSL). It needs to strike the right balance between expressiveness (to handle a range of common tasks in the target domain) and tractability (for the synthesis algorithm to learn correct programs efficiently).

Satisfiability (SAT) and Satisfiability Modulo (SMT) Solvers: Boolean satisfiability (SAT)-the problem of determining whether there exists an assignment satisfying a given Boolean formula-is a fundamentally intractable problem in computer science [26]. When applied to code generation, the Boolean formula stands for the expected output of the code for a given input, and the assignment stands for the code to be generated. While the language used by SAT is Boolean, the language used by SMT is first-order logic that makes it easier to express complex systems - similar to how assembly language is more comfortable than the machine language. In general, Program synthesis can be viewed as a second-order search problem, where the goal is to discover a function that satisfies a given specification, which roots in satisfiability modulo theories. Whereas traditional synthesis algorithms perform an expensive combinatorial search over all possible programs [35], new approaches take advantage of other approaches like machine learning to reduce this search space and improve the efficiency of SMT solvers [36].

Inductive Logic Programming: Inductive logic programming combines logic and inductive programming; some background constraints are specified using logic programming that has to be satisfied by a program that can fulfill a list of positive and negative examples. Inductive logic programming attempts to overcome three main shortcomings of the traditional inductive approach: 1) Restricted representation 2) Inability to make use of background knowledge, and 3) Strong bias of vocabulary [59].

Neural Programming Induction: In order to be able to tackle the NP hard nature existing in most generalization approaches, some researchers have leveraged neural networks that can either reduce the search space [60] or learn to select the best implementation from a set of candidates [28].

Compositional Synthesis: Compositionality is a fundamental notion in computation whereby complex abstractions can be constructed from simpler ones. Research studies have tried to apply this property to the paradigm of end-user programming from examples or natural language. The compositional synthesis tasks can be specified in a compositional manner through a combination of natural language and examples that can improve the expressivity and scalability of traditional synthesis techniques [77]

Type Inhabitation: Type inhabitation - also know as type-driven program synthesis - uses type specification to find a programs that can produce a desired type from the source type. The specification is the type of the desired expression, such as $List[String] \rightarrow Int$. The system then finds expression with the desired type, in this case, a function that transforms a list of strings into a single integer. This expression might be the length of the input list, the length of the first string in the list, the accumulated length of all strings, or anything else. Reducing the search space is an important consideration in type inhabitation. For example [30] introduces a succinct representation for type judgements that merges types into equivalence classes to reduce the search space.

Genetic Programming: Building on top of neural programming, genetic programming has the advantage of self-optimization. While trying to find a program that satisfies the constraints and realizing the examples, a genetic search algorithm becomes more optimized over time. In genetic programming, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (sexual recombination) operator appropriate for genetically mating computer programs [42].

3.3.2 Specialization Techniques. In contrast to generalization techniques, specialization techniques assume a priori knowledge about the structure of the output. For example, [73] considers generated code will be well-formed with respect to abstract syntax tree (AST).

Statistical Models: The statistical models use a priori knowledge in the source code and use it to either predict what keyword should come next given a set of observed keywords or try to generate substantially correct source code from the natural description of the code. For example, the likelihood of having an "else" keyword after an "if" keyword is much higher than the reverse sequence. Machine learning is the ideal tool to build statistical models that can capture such patterns.

There are different approaches to build statistical models. Semantic models only consider the probabilistic models of code and use it to make predictions irrespective of the syntax of the overall code and hence can produce code that could fail to compile [9]. Syntactic encoding models include a constraint that the generated code is required to be syntactically correct [91]. There is also a class of models that jointly model source code and natural language to either generate syntactically correct source code or meaningful natural language describing the code [2].

Graph Models: While statistical models have been successfully applied to capture programming patterns to support code completion and suggestion, they face challenges in capturing the patterns at higher levels of abstraction due to the mismatch between the sequential nature of such models and the structured nature of syntax and semantics in source code. The graph-based models try to overcome these challenges by capturing the non-sequential dependencies of the source code [64]. Another approach to create graph-based models is to use graph neural networks. In such models, not only code is represented by a graph but also the training of the neural network within this graph, which can eliminate or augment edges in the graph. The resulting graph is then used to make code predictions [13].

3.4 Measuring the Effectiveness of the Techniques

The measures used to evaluate models used for code generation are similar to those used in NLP research, including - but not limited to - the following metrics:

(1) **F1 score** is calculated from the precision and recall of the approach, where the precision is the number of correctly identified positive results divided by the number of all positive results, including those not identified correctly, and the recall is the number of correctly identified positive results divided by the number of all samples that should have been identified as positive.
(2) **Cross entropy** is a measure of the difference between two probability distributions for a given random variable or a set of events. [39]
(3) **Perplexity** is a measure of how well a probability distribution or probability model predicts a sample [74].
(4) **Min reciprocal rank** is a statistical measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by the probability of correctness [88].
(5) **Bilingual evaluation understudy (BLEU)** is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another [67].
(6) **Metric for evaluation of translation with explicit ordering (METEOR)** is a metric for machine translation evaluation that is based on a generalized concept of unigram matching between the machine produced translation and human-produced [6].
(7) **Recall-Oriented Understudy for Gisting Evaluation (ROUGE)** includes measures to automatically determine the quality of a summary by comparing it to other (ideal) summaries created by humans. The measures count the number of overlapping units such as n-gram, word sequences, and word pairs between the computer-generated summary to be evaluated and the ideal summaries created by humans reference translations [49].

One important quality metric of code is whether is can execute without errors for different input applied and if does how accurate the result is. The traditional NLP metrics mentioned above might not suffice to measure this quality.

3.5 Processes

One aspect that has been almost neglected is the process used in machine programming. It would make sense to ignore the process for smaller sizes of software like a code segment, a bug fix, or a unit test since the techniques introduced earlier would be able to generate them in one shot. However, when considering applications and systems as the output of such techniques, choosing the correct process becomes crucial. Specifically, since the provided requirements usually are ambiguous and their details are hashed out as they are turned into software, one can design a machine-based approach to clarify the requirements and explore their implementation in several iterations. Some research labs at Intel at Microsoft have been working recently to come up with the right approach to capture the user intentions in a more organized structure that can serve as a foundation to use machine programming to be able to create software applications [17].

3.5.1 Iterative. In an these approaches, multiple iterations are used to yield the output. Each iteration can extract some information from the input or generate a portion of the output [15].

3.5.2 Interactive. These approaches interact with users throughout the process. Each interaction can result in reducing gaps in information existing in the input [27].

3.5.3 Training-based. Training-based approaches are prevalent when the underlying technique uses machine learning. In these approaches pre-existing input-output pairs are used to create a static statistical model which is later used to make predictions based on the provided input. We will take a close look of several training-based approaches in subsequent sections.

3.5.4 Rule-based. Most generalization techniques like PBE rely on rule-based techniques like version algebra to generate programs. These techniques, while calculation intensive, can generate all possible programs that can satisfy a set of constraints.

3.6 Solutions

Different combinations of input and output can be used to solve one or more of the following Solutions:

3.6.1 Code Generation. Code generation solves the problem of replacing or aiding human developers with tools that can convert from any form of human understandable description of the code into code.

3.6.2 Code Translation. Code translation takes the code in one programming language and translates it into another "well-written" programming language. Note that being well-written is a key here; otherwise, a pair of compilers/decompiler can do the job.

3.6.3 Defect Detection. Being able to prevent bugs can reduce the cost of software development a lot. Fixing a bug detected during development is less costly than fixing the bug discovered by manual QA and far less costly than fixing a bug caught in production [83]. Defect detection can generate unit testing code based on the expected output used to detect defects during development.

3.6.4 Patch Detection. Patch detection generates code that fixes known bugs in the provided code. It is possible to learn a probabilistic model of correct code and recommend code to fix the known bugs by working with a set of successful patches obtained from open-source software repositories [89].

Type of Input \ Available Info	Syntactic	Shallow Semantic	Deep Semantic
Natural Description	No	Yes	No
Informal Requirements	No	Yes	No
Formal Requirements	Yes	No	Yes
Examples	No	No	Yes
Source code	Yes	Yes	Yes
Context Graph	Yes	Yes	Yes

Table 1. information form provided by each type of input

3.6.5 Code Classification. Code Classification specifies a class for a given block of code. The classification is important in many applications such as selecting the best place to execute it (CPU or GPU) [34].

3.6.6 Code Recommendation. Code recommenders provide suggestions to complete a partially written code. Note that this can go well beyond tools like IntelliSense that use the semantics of the code in addition to its syntax.

3.6.7 Superoptimization. Superoptimization is the process of automatically finding the optimal code sequence for one loop-free sequence of instruction (Wikipedia). Note that this can go well beyond using compilers that rely on brute force methods, which can be very costly [69].

4 SURVEYED PAPERS

In this section, we review the existing work on machine programming across the different dimensions of the taxonomy introduced in the previous section.

4.1 Across the Types of Input

User intent is the foundational information that drives what is included in the requirements used to document the software behavior. A small application might only have one user which can have several possible intents each applicable at a given state of the application. For larger applications, there can exist more users each with several intents. The decision on how to represent the user intent will have a fundamental impact on all three metrics that we defined earlier to differentiate the code generation techniques. Scholar has used various types of inputs and their combinations to represent the intent. However, more important than the actual input, it is the form of the information that is embedded in the input and used for machine programming. Each types of input provides one or more forms of information listed below:

Syntactic: provides information about the positional role of features. For example, when source code is converted to AST and used in that form, each token will carry meaning with respect to its position in the tree.

Shallow-semantic: provides a limited amount of information on what the features means beyond its syntactic information. For example, the name given to a variable or method provides insights on what its function is intended to be.

Deep-semantic: provides a higher level of information - either directly or indirectly - beyond both syntactic and shallow-semantic forms. For example, examples provide direct deep semantics on what the source code is intended to do or executed code provides information on what the actual source code function is.

4.1.1 Natural Language Description as Input. Using natural language to describe input has been receiving increasing attention due to the advances in natural language processing (NLP). In the studies that we have

surveyed It comes at the top rank in terms of being used to describe the intent for the program to be generated. Natural language can capture much more information than examples and it is much easier for non-technical people to produce compared to source code, therefore it is more practical to be used to describe the intent. However, there are two complexities in using it as the source. The first one, which is a technical problem, is the complexity involved in parsing the text and extracting meaningful information from it. The second one, which is a systematic problem, is the complexity involved in disambiguating the natural language. techniques exist that try to tackle either of these issues [23].

The natural language description is used to generate executable code in the overwhelming majority of the surveyed studies [51], [35], [24], [12]. For example [51] applies a complex neural network based on attention model introduced by [16] to convert code in Hearthstone card games. As shown in Figure 2, they use specific components to extract information from different areas of text.

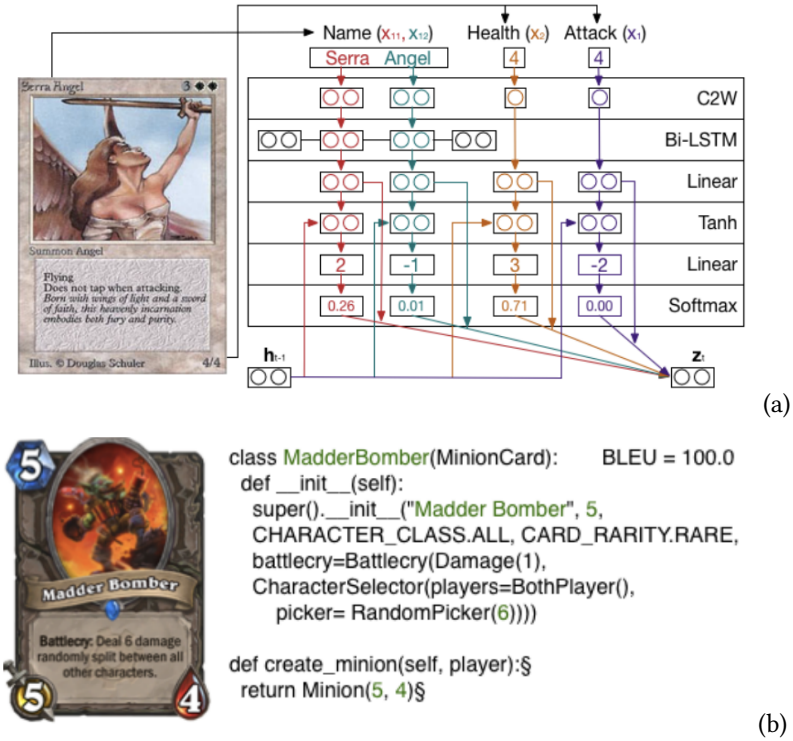


Fig. 2. [51] used text input provided in Hearthstone cards to generate code. a) The model used to convert text to code. b) An example input and output.

However, there are several studies that generate code that can only be compiled but fall short in generating the expected output [91], [2], [73], [37] or would not compile at all but are close to the code that could have been implemented by a human [63], [19], [32], [80]. Researchers have used both generalization and specialization techniques to transform natural language descriptions into code. Among generalization techniques, inductive logic programming [24], [19], [37] and neural programming synthesis [63], [68], [79] techniques are used most often Statistical syntactic model is the overwhelmingly prominent technique among the specialization ones [32],

[47], [58], [54]. The major difference between generalization and specialization techniques is that, the former, tries to iteratively map the provided description to a set of predefined components or operations whereas the latter build a model by training a neural network on text-code pairs.

As an example of an generalization technique applied to natural language description, [63] proposes a neural network augmented with a small set of basic arithmetic and logic operations that can be trained end-to-end using backpropagation and inducing compositional programs that are more complex than the built-in operations. On the other hand, as an interesting specialization technique based on natural language description [47] builds a joint probability model on both input text and output spec tree. It uses a Bayesian generative model to capture relevant natural language phenomena and translate the English specification into a specification tree, which is then translated into a C++ input parser.

4.1.2 Source Code as input. Source code is the second type of input in the papers we have reviewed. Source code is one of the two top input types in terms of the amount of information the input contains - the other one being the context graph. It provides syntactic information when it is parsed to an AST or other syntax-specific forms [4], [14], [38], [52], shallow semantics via the name assigned to variables and methods [43], [71], and deep semantics when it is executed and provides the functional state for the code [21].

Given the information-rich nature of source code, studies have used source code almost equally for both syntactic and semantic techniques. In terms of the type of Solution, source code has been used mostly either for code generation or code recommendation. In the former cases, the technique is used to translate from one language to another [38], [37] and in the latter it helps a programmer to complete their partially written code [33], [64], [75], [21]. For example, as shown in Figure 3, [75] uses a CRF-based approach [45] to assign proper names and types to in input JavaScript program.

```
function chunkData(e, t) {
  var n = [];
  var r = e.length;
  var i = 0;
  for (; i < r; i += t) {
    if (i + t < r) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, r));
    }
  }
  return n;
}
```

(a)

```
/* str: string, step: number, return: Array */
function chunkData(str, step) {
  var colNames = []; /* colNames: Array */
  var len = str.length;
  var i = 0; /* i: number */
  for (; i < len; i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

(b)

Fig. 3. [75] takes source-code as input and generates source-code with proper names and types. a) The input source code. b) The generated output.

When the source code is used as the input, the generated output is almost half of the time not compilable but at a state pretty close to what a human can verify the closeness by comparing the generated code with what the actual intent has meant to be. Finally most of the approaches that use source code as input, rely on a training process to create their model. This is the case since the structured nature of source code allows the machine learning based techniques to easily extract either syntactic or semantic features and use them for training [90], [53]. The other reason for this is the abundance of available public source code that can be used for creating statistical models [8].

4.1.3 Examples as Input. Examples are used frequently with generalization techniques. Examples are usually in the form of a few input/output pairs which identify the positive scenarios that a program is intended to do. Examples provide deep semantic information that can be used to generate programs with a few lines. They can implement numerical algorithms [48], [24] or simple text manipulation tasks [46], [77], however, they normally fall short to process exhaustive requirements that are required for larger applications. As an example in [77], authors offer an approach that takes several examples of extracting numbers from alphanumeric strings. The approach will then perform a systematic search over that state space to generate a program that can apply the transitions given in the examples. Figure 4 shows an example input and output for this approach. Here the input is an alphanumeric string. The output removes the first letter and splits the numeric and non-numeric parts coming after the first letter.

	Ex 1	Ex 2	Ex 3	Ex 4
Input	G2	G12345	G1234B	G123456
Output	G2	G12345	G1234B	null
"1-5 numbers"	2	12345		
"4 numbers"			1234	
"a single letter"			B	

(a)

```
Filter(
  DisjTok(
    ConcatTok(
      CharTok('G'),
      Interval(NumChar, 1, 5)),
    ConcatTok(
      CharTok('G'),
      ConcatTok(Interval(NumChar, 4), UpperChar))
  )
)
```

(b)

Fig. 4. [77] takes a few examples of what is desired and generates code for it. a) The input examples. b) The generated output.

Techniques that use examples are referred to as "Programming by Example" or PBE and majority of them use statistical generalization technique like neural programming synthesis to search for the solution [48], [15], [81], [56] and some use deterministic generalization techniques like version space algebra [27], [70] to find all possible programs satisfying the examples and then use a ranking algorithms to select the most appropriate one - which could be the shortest or the fastest. When examples are used as the input, the overwhelming majority of the approaches are used to generate executable code. This is the case since having the input and output at hand, there is no ambiguity in what the generated code is intended to do and the approach can only be considered as complete if it can be executed to generate the desired output.

Some scholars have used examples as a complement to other input types like source code or natural language. For example, [35] synthesis a program iteratively from a library of existing code components, and use examples to rank the synthesized programs. [77] proposes an approach that combines examples and natural language to generate code which is used for synthesizing algorithms used for expressive string manipulation.

4.1.4 Other type of Input. Other types of input that are used less frequently include formal and informal requirements, and context. Using formal requirements is probably the oldest approach to code generation; these approaches use low level logic to describe the system and use SAT and SMT solvers to find programs that satisfy them [22], [85]. Informal requirements are the de facto standard to define applications before they are turned into code by programmers, however, we have not seen any study that uses informal requirements to drive code generation. We think this could be due to two reasons. First, similar to natural language description, information extraction and disambiguation tend to be very challenging tasks. Second, studies that try to represent the inter-related requirements in a fashion that can correctly convey the user intent are almost nonexistent.

The last type of input, context - which we define as a set of user intents under different states for application entities - to define the application is - in our view - essential to produce software at the scope of a standalone application or systems but it has not been the focus of machine programming research so far.

4.2 Across the Types of Output

Majority of the studies we have surveyed aim to produce code that can be executed. This is no surprise since the ultimate goal of machine programming is to generate fully-functional applications that can be executed upon generation to carry out the user intent. However, when input is either not precise enough or not easy to extract the existing information from, producing executable code becomes a very challenging task. Therefore many studies suffice to either generate code that is compilable but not executable [91], [38], [73] or even produce code that contain syntactic problems but can be verified by a human that it is very close to what a human programmer would have generated [34], [41], [61]. There is also a third class that approaches that generates syntactically correct output which is not necessarily executable [2].

4.2.1 Executable Output. The major output of the surveyed studies generate smaller yet executable code. In contrast to specialization ones, generalization techniques in general produce executable code with an overwhelming majority. This is expected since generalization techniques normally have clearly-defined inputs and outputs and use predefined components. They also tend to produce a few lines of code [81], [56]. Among generalization techniques, neural programming synthesis is the major technique that produces executable cod [21], [68]. The success of this technique lies on two components: first, like other generalization techniques, it relies on clearly defined input-output pairs, second, contrasting other generalization techniques it utilizes machine learning to solve the challenging task of ranking and refining the solution among the large number of candidates [81]. Other approaches like version space algebra normally use computationally heavy operations and heuristics that are less effective in refining the candidate solutions [27]. As an example [68] presents an generalization technique called that learns to generate a program incrementally without the need for an explicit search. Once trained. This approach can automatically construct computer programs that are consistent with any set of input-output examples provided at test time. Figure 5 provides a few examples that can be handled by this approach. Note that, as apposed to the PBE example shown earlier, being a neural network in essence, this approach can only predict the output without offering an explicit source code that can do the conversation.

Input v	Output
[CPT-00350	[CPT-00350]
[CPT-00340]	[CPT-00340]
[CPT-114563]	[CPT-114563]
[CPT-1AB02	[CPT-1AB02]
[CPT-00360	[CPT-00360]

Input v	Output
732606129	0x73
430257526	0x43
444004480	0x44
371255254	0x37
635272676	0x63

Input v	Output
John Doyle	John D.
Matt Walters	Matt W.
Jody Foster	Jody F.
Angela Lindsay	Angela L.
Maria Schulte	Maria S.

Fig. 5. [68] uses a neural programming synthesis technique to produce outputs following the patterns observed in provided input examples.

Although the majority of approaches producing executable output do so by showing they can produce the expected output for a given input, a few studies use a different measure. For example, [66] presents an automated repair method based on symbolic execution, constraint solving, and program synthesis. In this approach, the repaired code has to pass a given set of tests which are formulated as constraints. Such constraints are then solved by iterating over a layered space of repair expressions.

4.2.2 Human-Verifiable Output. There is a trade off between the accuracy of the generated code - which increases the execution rate - and size of the generated code. Whereas generalization techniques focus on the former factor, specialization techniques tend to produce larger applications [58]. In doing so they often produce code that is neither executable nor compilable; however, it is very close to what could have been written by a human programmer. These approaches use a training based process most of the time to create a model to predict code

given a crude description, e.g., natural language description. They normally train their model over a very large repository of public code [8] and therefore are capable of producing code for almost any provided input. However, due to their statistical nature, they do not necessarily generate a fully accurate code that can be executed to produce the expected output. For example [58] uses a recurrent neural network to generate code from the user intention. Although the generated output is very close to the expectation, it contain several errors as shown in Figure 6.

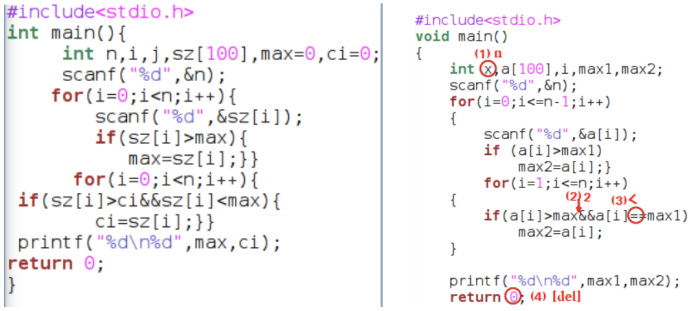


Fig. 6. [58] uses an RNN-based network to generate code from user intention. The code can be verified by human that is close to the intent but contains several errors. a) Expected output b) Generated Output

4.2.3 Compilable Output. The third group of studies we have surveyed try to create a balance between executability and size of the out. They produce code that can be compiled but not necessarily produce the expected outcome. An overwhelming majority of such studies use specialization techniques and not very surprisingly rely on syntactic models to be able to produce code that is syntactically correct [53], [38]. Almost half of studies in this category are related to code generation solutions [92] and the other half are related to code recommendation solutions [65]. For example [53] uses a probabilistic context free grammar to model the abstract syntax tree for C compiler and uses it to produce code that maps to a valid AST and therefore can compile. Figure 7 shows an example of AST used and also the code generated by their approach.

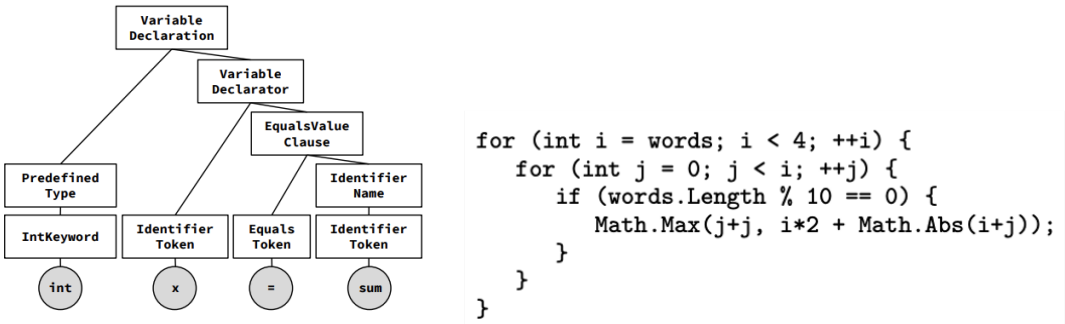


Fig. 7. [53] uses a specific PCFG to generated code complying with C AST. a) Example AST b) Generated Output

Other than code generation or code recommendation, a few studies in this category target transforming code from one compilable form to another. For example, [38] investigates the application of phrase-based statistical

machine translation to the problem of translating programs written in different programming languages and environments.

4.3 Across Techniques

As mentioned earlier, two major categories of techniques that are used in machine programming: generalization techniques try to generalize from a few examples and generate programs that can be applied to similar inputs whereas specialization techniques try to learn generic models from large datasets and can be applied to a wider set of inputs. Most generalization techniques are built on top of deterministic methods whereas specialization ones use statistical methods. However, an interesting observation based on our survey is that even within generalization techniques, the ones that include statistical components are trending up. Specifically neural programming synthesis that uses a neural network to refine the solution space dominates among the generalization-based studies. Another observation is that, among specialization techniques, the majority of techniques are based on syntactic models. The reason for this could be due the challenges existing in extracting shallow and deep semantic information from the provided input. The last observation is on graph-based methods that although they are not being used as much as other techniques, they are being considered in the most recent studies. We think this is due to the recent advances in graph based methods in machine learning.

4.3.1 Generalization Techniques. Neural programming synthesis (NPS) is the dominating technique across generalization techniques in the surveyed studies. As shown in the figure 8, NPS techniques apply machine learning to refine the solution space derived by applying generalization techniques. For example [44] studies the problem of efficiently predicting a correct program from a large set of programs induced from a few input-output examples. This is an important problem for making Programming by Example systems usable so that users do not need to provide too many examples to learn the desired program. They first use version-space algebra to derive the solution space and then use a supervised machine learning method for learning a hierarchical ranking function to efficiently predict a correct program.

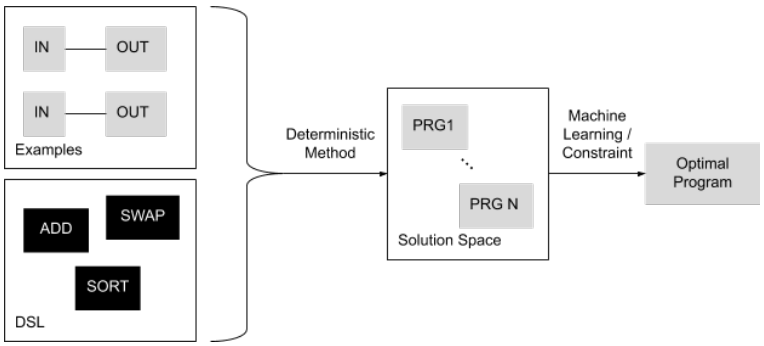


Fig. 8. Generalization Techniques

Like other generalization techniques, NPS uses examples in the overwhelming majority of studies to define the solution space and uses training to refine the solutions and select the optimal program. Majority of the approaches generate programs that can execute to produce the expected results.

Another widely used generalization techniques is inductive logic programming (ILP). ILP has been defined as the intersection of inductive learning and logic programming **??**. From inductive machine learning, ILP inherits its goal: to develop tools and techniques to induce hypotheses from observations (examples) and to synthesize new knowledge from experience. From logic programming, ILP inherits its representational formalism and semantic

orientation. In inductive logic programming, given a dataset, a set of starting view definitions, and a target predicate, we can infer the view definition of the target predicate. For example Figure 9 shows an example where given a few dataset on family relationships we can use ILP to infer new rules defining father and mother.

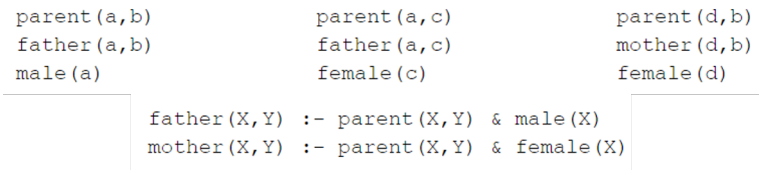


Fig. 9. An example of using ILP using input datasets to infer logical rules.

Other generalization techniques include version-based algebra, inductive logic programming, SAT/SMT solvers, compositional synthesis, and genetic programming. As shown in Figure 10, [48] provides a technique that combines elements from compositional and statistical methods to synthesize a program by composing partial programs.

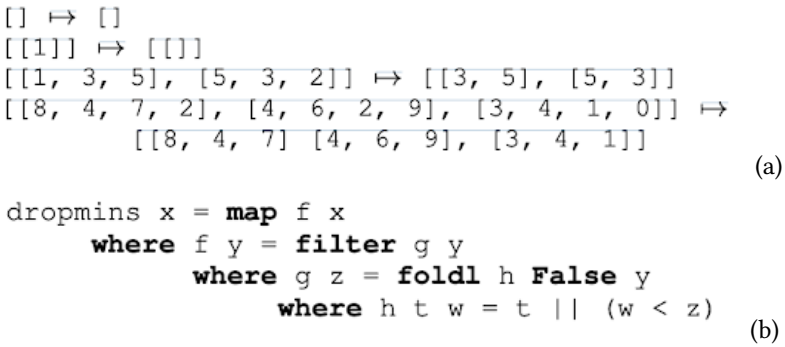


Fig. 10. The Generated Code from Compositional Technique in [48] Given the Provided Input. a) Provided examples where the minimum number in each set is dropped. b) A program that is composed of several sub-programs within the defines DSL including map, where fold1, False, and logical and basic mathematical operations.

4.3.2 Specialization Techniques. Statistic syntactic techniques dominate within the specialization category. Although semantic information provides much more information than syntactic ones, extracting such information requires new effective techniques like graphs and contextual models that have been studied less frequently [73]. Therefore most techniques rely on syntactic information that can be easily extracted from source code. Another reason to favor syntactic techniques is that they are created by using syntactic representations like AST as the training features [73] and therefore are able to produce code that is compilable which could be required in most practical applications. Some studies try to use both syntactic and semantic models to bring the positive benefits of both sides. Such approaches tend to generate code that is both compilable and meaningful [75].

Almost half of the studies we have surveyed that are based on syntactic techniques use the trained model for code generation [54] yet several of them use it with code recommendations solutions [52]. As shown in Figure above these approaches rely on a huge dataset of natural language description (NLD) to code mappings. They first process this raw dataset to produce key phrases (KP) to AST parings and use them as training features.

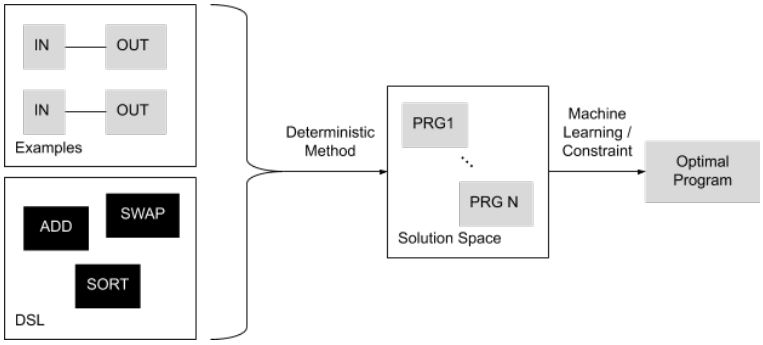


Fig. 11. Specialization Techniques

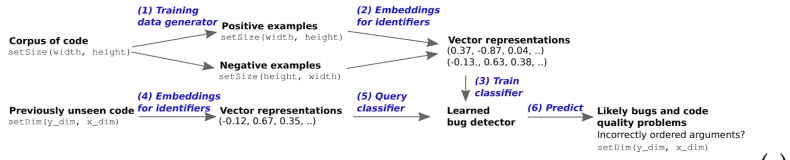
Finally a machine learning approach like deep learning is used to generate a model that can be used for code generation. Whereas generalization techniques use examples and produce executable programs most of the time, specialization techniques take source code or natural language as input and produce human-verifiable or compilable programs. This observation perfectly aligns with our expectation that we stated earlier as the trade off between accuracy and size of the generated code. generalization techniques are more accurate whereas specialization techniques can produce larger sizes of code. Finally all the studies we have surveyed in this category use training to train their models.

The next mostly used specialization technique is semantic statistical. Statistically speaking, while in the majority of cases, semantic techniques are used to solve either code generation [44] or code recommendation [75] problems, compared to synthetic ones, they do less so instead they attempt to solve problems not tackled by the syntactic techniques including code translation [38] and defect detection. They tend to use source code more often than natural language description as their input and produce code that is less frequently compilable compared to syntactic techniques. This makes sense since these techniques focus less on syntax and more on meaning of the information contained in the input.

In the studies that we have surveyed we have seen semantics being used in two forms: shallow semantics and deep semantics. As an example for the former case, [71] presents a learning approach to name-based bug detection, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them. They formulate bug detection as a binary classification problem and train a classifier that distinguishes correct from incorrect code. This is shown in Figure 12.

As an example of using deep semantics [21] presents a model that integrates components which write, execute, and assess code to perform a stochastic search over the semantic space of possible programs. They use a tool called REPL which immediately executes partially written programs, exposing their semantics. The REPL addresses a fundamental challenge of program synthesis: tiny changes in syntax can lead to huge changes in semantics. By conditioning the search solely on the execution states rather than the program syntax, the search is performed entirely in the semantic space.

Other specialization techniques are graph based. Although there are not many such techniques, more scholars have been looking at them most recently. Graph-based techniques exist in both syntactic and semantic forms. For example [20] presents a syntactic graph learning-based approach to detect and fix a broad range of bugs in JavaScript programs. Given a buggy program modeled by a graph structure it makes a sequence of predictions including the position of bug nodes and corresponding graph edits to produce a fix. [34] presents a new graphical structure to capture semantics from code using a semantic graph called program-derived semantic graph (PSG). The principle behind the PSG is to provide a single structure that can capture program semantics at many levels



(a)

```
var p = new Promise ();
if (promises === null || promises.length === 0) {
  p. done (error, result )
}
else {
  promises [0]( error, result ).then( function(res, err) {
     p. done (res, err );
  });
}
```

(b)

Fig. 12. Understanding shallow semantics, DeepBug can detect the bug in passing the parameters in the wrong order to "promise.done". Syntactic approaches are not able to detect such bugs. [71]. a) Overview of the DeepBug Approach. b) A program that is composed of several sub-programs within the defines DSL including map, where fold1, False, and logical and basic mathematical operations.

of granularity. Thus, the PSG is hierarchical in nature. As shown in the figure 13, this allows PSG to understand whether two blocks of code with different syntax have similar semantics.

4.4 Across Solutions

Machine programming can solve many different problems that need to generate code without relying on human beings. Two major factors aiding machine programming to provide these solutions include:

Utilizing the knowledge available in existing public source code repositories which has made it possible to create learning-based techniques mapping code to its various specifications. The semantic Information already available within the source code or represented as the state of the application when it is executed.

4.4.1 Code Generation. Majority of the studies that we have surveyed use the mentioned factors to tackle code generation. Code generation is the cornerstone of machine learning since if it is completely solved, other problems will automatically disappear. Almost half of such studies use natural language description (NLD) as the input followed by examples and source code. We have already reviewed several cases of using either NLD or examples to generate code.

Like the general trend, most of the code generation solutions rely on neural programming synthesis (NPS) followed by syntactic statistical based techniques. As mentioned earlier NPS is used when the provided input is an example whereas syntactic statistical technique is used when the input is NLD. Majority of the code generation solutions use training as the process which can either be a step in NPS to select the best program or are used to create the statistical model.

Machine learning is used quite extensively when it comes to code generation. With specialization techniques, most of the time machine learning is used to create a mapping between NLD features (either syntactic or semantic) and code segments, however, some researchers use ML in other fashions. For example, as shown in Figure 14, [5] uses machine learning to map from input-output examples to program attributes which is then used to select a search procedure that searches program space.

Implementation 1

```
0 signed int recursive_power(signed int x, unsigned int y)
1 {
2    if (y == 0)
3       return 1;
4    else if (y % 2 == 0)
5       return recursive_power(x, y / 2) *
           recursive_power(x, y / 2);
6    else
7       return x * recursive_power(x, y / 2) *
           recursive_power(x, y / 2);
8 }
```

Implementation 2

```
0 signed int iterative_power(signed int x, unsigned int y)
1 {
2    signed int val = 1;
3    while (y > 0) {
4       val *= x;
5       y -= 1;
6    }
7    return val;
8 }
```

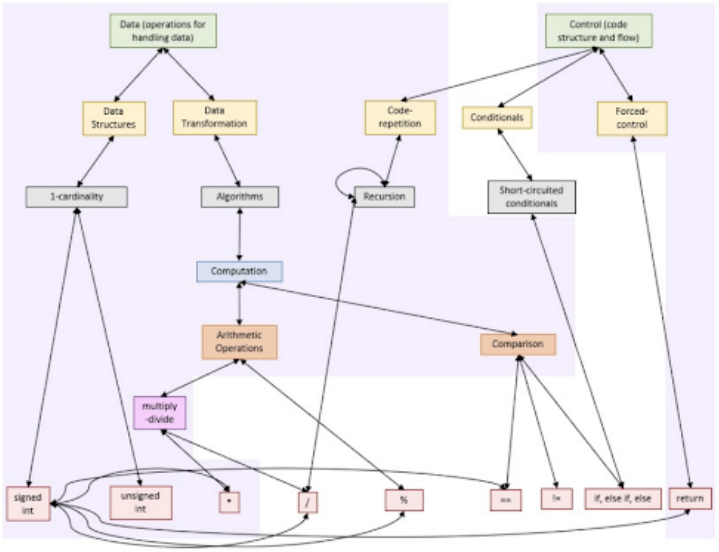


Fig. 13. Two different implementations of exponential function (x^y). The first one is recursive whereas the second one is iterative. The shaded area shows the overlap in the PSG for both approaches which is about 70%. Other approaches fall short of providing such a high overlap [34].

4.4.2 Code Recommendation. The second mostly studied solution is code recommendation. Code recommendation is a much easier problem to tackle than code generation due to two reasons: first, having access to a large

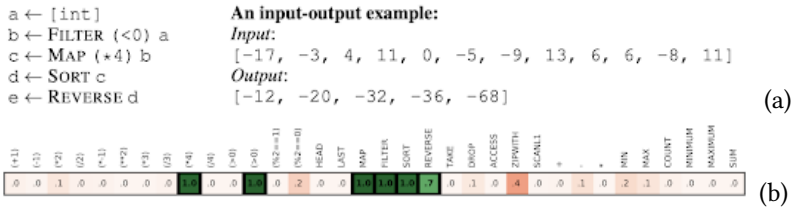


Fig. 14. DeepCoder tries to solve generated code for input/output examples using neural programming synthesis augmented by search techniques. a) Input provided and the generate code b) The predictions of a trained neural network made from input-output examples for the program shown in (a).

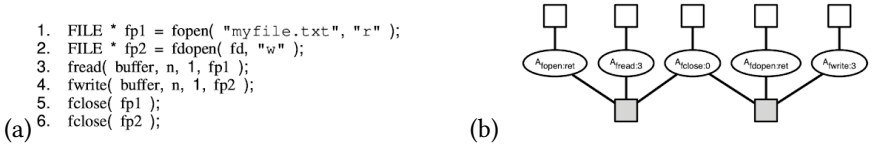


Fig. 15. AFG for a sample program that maps both check factors and beliefs to variables used in the code. a) A program with potential bugs if pointers are not de-allocated before functions returns. b) Factor graph model for the code. Circular nodes correspond to variables and square nodes to factors. The shaded factors indicate check factors, while the top row depicts factors modeling prior beliefs.

repository of source code, most of the approaches build statistical models that can predict what would be the next statement given a partial code block. Second, the result can be verified easily either by syntax check or human-verification.

Most of the code recommendation studies use source code as their input. All of these studies use machine learning to train a model which is most of the time based on specialization techniques that use either a syntactic model and or a semantic model. Given that the suggested word or statement is applied to a partial program none of these solutions generate an executable code but they can be validated by humans or checked for a valid syntax using compilation.

In order to make sure the recommended statement is syntactically correct, some studies incorporate syntactic components into their models. For example [53] combines probabilistic context free grammars (PCFG) with AST-based syntax.

4.4.3 Other Solutions. Other solutions that have been studied less include patch prescription [89] defect detection [82], and code translation [38]. For example, [43] uses an interesting approach to detect defects without required explicit rules. It uses an annotation factor graph (AFG) shown in Figure 15 to specify specifications. There are two types of factors in AFG: first, check factors that extract behavior from the program itself. The second set of factors are used to model arbitrary domain-specific knowledge. This includes prior beliefs about the relative frequency of certain specifications, knowledge about suggestive naming conventions.

4.5 Across Processes

Given the input, output, and technique, the process specifies how the technique is applied to transform the input to the output. The studies that we have surveyed use a training-based process predominantly to apply the technique. This is due to the recent progress in using machine learning to extract and use information and patterns available in the vast amounts of data. Such Training based processes have taken over rule-based processes that

aim to compile an encompassing set of rules that can be applied to the majority of the cases. However, given that the most used type of input, NLD, is not structured, rule-based processes are not able to scale as training-based processes do.

In addition to training-based and rule-based processes that try to build the output in one shot, a few other studies use processes use multiple iterations [12] or interact with the user to collect information that can reduce or eliminate gaps in the provided input. The interactive processes are mostly used with example-driven code generation. For example [46] uses the changes in the application state that result from the user's demonstrated actions, and learns the general program that maps from one application state to the next. [25] presents a a both iterative and interactive process called CodeHint that helps programmers write difficult statements. Figure 16 shows an input and the corresponding output for CodeHint. In this example the output is generated in three iterations: it first queries the debugger for all the variables in scope, evaluating them so it knows their dynamic types. Second, it combines simple expressions into more complicated ones according to the Java grammar. Third, it combines the candidates to produce larger expressions. In the last iteration, CodeHint uses its probabilistic model to avoid searching some additional expressions. If CodeHint had not found any results that satisfied the user's intent at this point, it notifies the user and asks if it should continue for another iteration.

```
1 final JComponent tree = makeTree();
2 tree.addMouseListener(new MouseAdapter() {
3   public void mousePressed(MouseEvent e) {
4     int x = e.getX(), y = e.getY();
5     Object o = null;
6     // Get the menu bar or the clicked element.
7   }
8 });
```

(a)

```
((JFrame)SwingUtilities.getWindowAncestor(jtree))
  .getJMenuBar()
((JFrame)tree.getTopLevelAncestor()).getJMenuBar()
((JFrame)SwingUtilities.getRoot(tree)).getJMenuBar()
```

(b)

Fig. 16. CodeHint uses and iterative and interactive process helping programmers write difficult statements a) Input provided containing a hint of what code is expected in line 5 b) The generated code by CodeHint.

5 CONCLUSION AND FURTHER STUDIES

In this survey we provided a taxonomy that offer a new way of dissecting research on machine programming techniques. The taxonomy can answer the why, what, and how questions about code generation:

(1) What form of input is used to explain what the intent is and what will be generated as the output?
(2) How the transformation is performed?
(3) Why the transformation is important and what problem it can solve?

While our taxonomy sheds light on how the existing studies can be categorized and explained to address the essential questions posited above, it can also be used to detect existing gaps in the five dimensional space of {input, output, techniques, process, solution} and can be used to show us some new paths to consider in future studies.

On the input dimension, there is a lack of research in the following areas:

(1) Considering natural description that contains ambiguities. Most of available studies consider a description which is either completely understood or clarified by supplemented examples. However, such assumptions are generally invalid when we consider real-world software projects where requirements are often contain ambiguities that need to be detected and eliminated before machine programming can take place.
(2) Considering descriptions that are inter-related. The existing studies only consider a sentence or a paragraph the contain all the required information. However, in reality we are faced with a set of inter-related requirements that should be processed and mapped using a data structure than can represent such dependencies. Such a representation can help with eliminating ambiguities by relying on novel NLP techniques.
(3) The descriptions used to describe the code are often based on first order terms and meanings that can be easily mapped to programming instructions or a set of defined DSL. However, in reality the words used to define the requirements assume a set of underlying context. The future research can consider using contextual information to help with refining requirements.

On the output dimension, the existing studies only consider generating a code fragment, however, studies that consider generating code for larger scopes like a module within an app or an entire app is completely missing. Note that the we cannot simply assume that once we are able to generated code at the fragment level we can use a divide-and-conquer approach to build complete applications. This is because an application is more than a sum of its code fragments. Specifically an application depends on contextual dependencies that can only become available when we consider all the requirements together.

On the technique dimension, we think the graph-based approaches are the only candidate that can represents requirements beyond a simple refined sentence. Recently there has been some studies that look as using graph based techniques in machine programming, however, the number of such studies is much less than non-graph approaches.

On the process dimension, there is a lack of interactive approaches. Assuming that all the information is readily available in the provided description is an unrealistic simplifying assumption. The advanced in NLP and conversational AI can be used to detect ambiguities and create a dialogue with a human analyst to refine the provided description.

Finally, on the solution dimension, we see a wide open area for research around generating code for and end-to-end application and also systems that contain several applications. The gap in this area is possibly due to lack of required research and tools in the other dimensions but expanding research in those dimension can open the door to research that can enable us to generate end-to-end applications and systems.

6 AUTHORS AND AFFILIATIONS

7 CITATIONS AND BIBLIOGRAPHIES

```
\bibliographystyle{ACM-Reference-Format}
\bibliography{bibfile}
```

8 ACKNOWLEDGMENTS

9 APPENDICES

```
\appendix
```

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] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International conference on machine learning*. 2123–2132.

[3] Edward B Allen and Taghi M Khoshgoftaar. 1999. Measuring coupling and cohesion: An information-theory approach. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*. IEEE, 119–127.

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[6] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[7] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. 1993. Software complexity and maintenance costs. *Commun. ACM* 36, 11 (1993), 81–95.

[8] Islam Beltagy and Chris Quirk. 2016. Improved semantic parsers for if-then statements. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 726–736.

[9] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.

[10] Daniel M Berry and Erik Kamsties. 2004. Ambiguity in requirements specification. In *Perspectives on software requirements*. Springer, 7–44.

[11] James M Boyle and Monagur N Muralidharan. 1984. Program reusability through program transformation. *IEEE Transactions on Software Engineering* 5 (1984), 574–588.

[12] Forrest Briggs and Melissa O'neill. 2006. Functional genetic programming with combinators. In *Proceedings of the Third Asian-Pacific workshop on Genetic Programming, ASPGP*. 110–127.

[13] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).

[14] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 213–222.

[15] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

[16] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. 2015. Attention-based models for speech recognition. In *Advances in neural information processing systems*. 577–585.

[17] Brian Cremeans, Marcos Emanuel Carranza, Krishna Surya, Mats Agerstam, and Justin Gottschlich. 2019. Intent-based machine programming. US Patent App. 16/455,125.

[18] Fabian de Bruijn and Hans L Dekkers. 2010. Ambiguity in natural language software requirements: A case study. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 233–247.

[19] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469* (2017).

[20] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2019. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*.

[21] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*. 9169–9178.

[22] Dawson R Engler. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. *ACM SIGPLAN Notices* 31, 5 (1996), 160–170.

[23] Alessio Ferrari and Andrea Esuli. 2019. An NLP approach for cross-domain ambiguity detection in requirements engineering. *Automated Software Engineering* 26, 3 (2019), 559–598.

[24] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

[25] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*. 653–663.

[26] Parikshit Gopalan, Phokion G Kolaitis, Elitza N Maneva, and Christos H Papadimitriou. 2006. The connectivity of boolean satisfiability: Computational and structural dichotomies. In *International Colloquium on Automata, Languages, and Programming*. Springer, 346–357.

[27] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[28] Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL meets ML. In *Asian Symposium on Programming Languages and Systems*. Springer, 3–20.

[29] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
[30] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 27–38.
[31] Warren Harrison. 1992. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering* 18, 11 (1992), 1025–1029.
[32] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025* (2018).
[33] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.
[34] Roshni G Iyer, Yizhou Sun, Wei Wang, and Justin Gottschlich. 2020. Software Language Comprehension using a Program-Derived Semantic Graph. *arXiv preprint arXiv:2004.00768* (2020).
[35] 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.
[36] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018).
[37] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.
[38] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 173–184.
[39] Vlado Keselj. 2009. Speech and Language Processing Daniel Jurafsky and James H. Martin (Stanford University and University of Colorado at Boulder) Pearson Prentice Hall, 2009, xxxi+ 988 pp; hardbound, ISBN 978-0-13-187321-6, 115.00.
[40] Emanuel Kitzelmann. 2009. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*. Springer, 50–73.
[41] John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4, 2 (1994), 87–112.
[42] John R Koza and James P Rice. 1992. Automatic programming of robots using genetic programming. In *AAAI*, Vol. 92. Citeseer, 194–207.
[43] Ted Kremenek, Andrew Y Ng, and Dawson R Engler. 2007. A Factor Graph Model for Software Bug Finding.. In *IJCAI*. 2510–2516.
[44] Nate Kushman and Regina Barzilay. 2013. 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*. 826–836.
[45] John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001).
[46] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1-2 (2003), 111–156.
[47] Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. 2013. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1294–1303.
[48] Percy Liang, Michael I Jordan, and Dan Klein. 2010. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 639–646.
[49] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
[50] Lowell Lindstrom and Ron Jeffries. 2004. Extreme programming and agile software development methodologies. *Information systems management* 21, 3 (2004), 41–52.
[51] 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).
[52] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.
[53] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *International Conference on Machine Learning*. 649–657.
[54] Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating Programming by Example and Natural Language Programming.. In *AAAI*.
[55] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. 1999. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st international conference on Software engineering*. 44–53.
[56] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *International Conference on Machine Learning*. 187–195.
[57] Sushmita Mitra and Yoichi Hayashi. 2000. Neuro-fuzzy rule generation: survey in soft computing framework. *IEEE transactions on neural networks* 11, 3 (2000), 748–768.

[58] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211* (2015).

[59] Stephen Muggleton. 1992. *Inductive logic programming*. Number 38. Morgan Kaufmann.

[60] Stephen Muggleton, Cao Feng, et al. 1992. Efficient induction of logic programs. *Inductive logic programming* 38 (1992), 281–298.

[61] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2017. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698* (2017).

[62] Brad Myers, Jim Hollan, Isabel Cruz, Steve Bryson, Dick Bulterman, Tiziana Catarci, Wayne Citrin, Ephraim Glinert, Jonathan Grudin, and Yannis Ioannidis. 1996. Strategic directions in human-computer interaction. *ACM Computing Surveys (CSUR)* 28, 4 (1996), 794–809.

[63] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834* (2015).

[64] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 858–868.

[65] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 69–79.

[66] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[67] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[68] 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).

[69] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310.

[70] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[71] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[72] Oscar Pulido-Prieto and Ulises Juárez-Martínez. 2017. A survey of naturalistic programming technologies. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–35.

[73] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[74] Nihar Ranjan, Kaushal Mundada, Kunal Phaltane, and Saim Ahmad. 2016. A Survey on Techniques in NLP. *International Journal of Computer Applications* 134, 8 (2016), 6–9.

[75] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[76] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2014. Programming by example using least general generalizations. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*. Citeseer.

[77] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[78] Buchi RICHARD et al. 1962. On a Decision Method in Restricted Second Order Arithmetic. In *Proc. of the International Congress on Logic, Method and Philosophy of Science, 1962*. Stanford University Press.

[79] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

[80] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*. Springer, 593–607.

[81] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*. Springer, 398–414.

[82] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[83] Sandra A Slaughter, Donald E Harter, and Mayuram S Krishnan. 1998. Evaluating the cost of software quality. *Commun. ACM* 41, 8 (1998), 67–73.

[84] Phillip D Summers. 1977. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)* 24, 1 (1977), 161–175.

[85] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability modulo recursive programs. In *International Static Analysis Symposium*. Springer, 298–315.

[86] Wolfgang Thomas. 2008. Solution of Church's Problem: A tutorial. *New Perspectives on Games and interaction* 5, 23 (2008), 3.

[87] Philip R Thrift. 1991. Fuzzy Logic Synthesis with Genetic Algorithms.. In *ICGA*. 509–513.
[88] Ellen M Voorhees et al. 1999. The TREC-8 question answering track report. In *Trec*, Vol. 99. 77–82.
[89] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.
[90] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 334–345.
[91] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).
[92] John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*. 1050–1055.