# A Survey of Machine Programming Techniques

In this survey, we review and classify the research on machine programming. We are specifically interested in the research that takes a description of user intent and generates software. We introduce a taxonomy with five dimensions and use it to classify the existing research. We also review the research within each category.

**ACM Reference Format:**
. 2021. A Survey of Machine Programming Techniques. 1, 1 (September 2021), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Code generation - also known as automatic programming - comprises semi-automated or automated approaches for the synthesis of software from non-software artifacts [7]. Earlier methods have mainly required structured, rigorous, reasonably complete, and unambiguous descriptions of user requirements, needs, and/or intent [1] as the input. However, by relying on advances in machine learning, recent code generation approaches have used less structured, at times free-form, partial, and often ambiguous descriptions as the input. Researchers have thus started referring to such practices by the term "machine programming" [27]. We define a machine programming approach as a set (I, O, T) denoting input required by the approach, output produced by it, and the specific technique it employs to convert the input to the output.

We note that program synthesis is a related term widely used to describe a specific class of code generation techniques. Gulwani defines program synthesis 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 [32]. The main difference between machine programming and program synthesis is the complexity of the approach. The input to a program synthesis technique primarily consists of numerical [39] or string [29] input-output examples. Its output is a generated code fragment that can work on all provided examples, contains a small set of instructions (e.g., if statements, loops, assignments), and typically does not comprise more than a few code lines. By contrast, machine programming considers more complex input like natural language description [59], pseudocode [104], or even images [79] as the input. The generated code by machine programming is also more complex and can consist of several complete methods containing all the available instructions in a typical high-level programming language [38].

While machine programming's focus is on generating code based on some description of the user intent, in general, there are three directions that the related research has taken: (1) taking non-code artifacts and turning them into code or improving existing code, (2) converting code to non-code artifacts, (3) translating code from one language to another. While the first category's primary output is code, the other two directions can use the code as their input or output or both. For example, there are machine programming applications in converting comments or examples to code [59] in the first category. There are studies on automated bug finding and fixing

[1] We will refer to the user requirements, needs, and intent simply as "intent" in this paper for simplicity.

Author's address:


XXXX-XXXX/2021/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[47] [4], or documentation [36] in the second category. In the third category, one approach shows how to translate code from one language to another without degrading its readability [42].

This paper is a survey of machine programming techniques. We specifically focus on approaches that take non-code artifacts, turn them into code or augment and improve existing code - i.e., the first of the three directions mentioned earlier. This decision was motivated by two reasons: first, the widely acknowledged benefits gained by reduced time and cost and increased quality when human involvement is minimized [90]; second, the sophisticated nature of approaches used to substitute human participation beyond rule-based automation [5]. The second and third categories, while useful, are outside our scope. The second category - which is also known as code-to-comment conversion by the natural language processing community - deals with techniques that translate lines of code into comments using a combination of NLP and machine learning techniques [36]. The third category has been researched much less extensively than the other two since rule-based techniques "mechanically" perform code-to-code translation without requiring any special sort of intelligence.

Although machine programming has seen a recent increase in attention from software engineering and machine learning communities, it has been studied in various incarnations for decades since the 1960s. The earliest work that can be considered machine programming dates to over 60 years ago when Church introduced the problem of synthesizing an electronic circuit to realize a mathematical function. It subsequently became the origin of research on combinatorial logic [96]. This was followed by the inductive programming approaches such as first-order logic that could generate formal language programs in the 1960s [85]. In the 1970s, inductive programming addressed learning of typically declarative or functional logic, and often recursive programs from incomplete specifications [93]. The predominant focus in the 1980s was on automation using patterns, and reusability [11]. The 1990s witnessed the rise of fuzzy logic to capture uncertainties and the onset of using AI to generate code [97]. In the 2000s, the focus was on model-driven approaches [60] and inductive programming synthesis [44]. Since 2010, with dramatic advances in machine learning, many researchers have started using neural networks to translate input-output examples or natural language descriptions of the desired software into code [1].

This survey's major contribution is a taxonomy to classify machine programming research that focuses on code generation. The taxonomy comprises five principal dimensions. First to understand the benefits and implementation of a machine programming approach, it identifies (1) *application* (what problem is being solved using the approach), and (2) *process* (how the technique is implemented and applied). To understand the approach itself, it identifies its (3) *types of input* (a description of desired software capability), (4) *types of output* (some form of generated code), and (5) *technique* (the specific manner in which the input is converted into the output). We have used these five dimensions to categorize the existing research and identify areas that deserve more attention. Our survey elaborates each dimension and provides representative examples from existing approaches.

It is worth noting that this is not the first survey in this field. For example, there are surveys on more traditional techniques, such as inductive programming [44] and fuzzy logic [63]. In addition, two surveys on this topic have recently emerged. These surveys focus on more recent techniques used in code generation natural-language-based approaches and program synthesis [1, 32]. By contrast, our goal is to provide a more comprehensive study of machine programming techniques and to properly contextualize the recent advances within the larger body of work in this area. Furthermore, these two recent surveys offer categorizations of existing research with relatively narrow foci. For example, Allamanis [1] offers a taxonomy based on the type of models used to generate or represent the code, which comprises only one dimension within our taxonomy. As another example, Gulwani [32] offers a survey based on examples as program inputs and executable code as the output, comprising a subset of the input and output types included in our taxonomy. Our survey offers a broader categorization resulting in a new comprehensive taxonomy with five dimensions. Using this more expansive view, however, has resulted in some unavoidable overlap with other surveys.

The remainder of the paper is organized as follows. Roadmap will be included here.

## 2 EARLIER SURVEYS

Recent research on machine programming has taken two directions: generalization and specialization. The former starts from a specific set of constraints and generates a program satisfying these constraints. We refer to these approaches as *generalization* approaches since they generalize a particular set of examples or constraints to produce a program. Approaches from the latter direction first generate a probabilistic model on the text to program relationship and then apply it to the provided description of user intent. We refer to these approaches as specialization since they specialize in the general probabilistic model to produce a program.

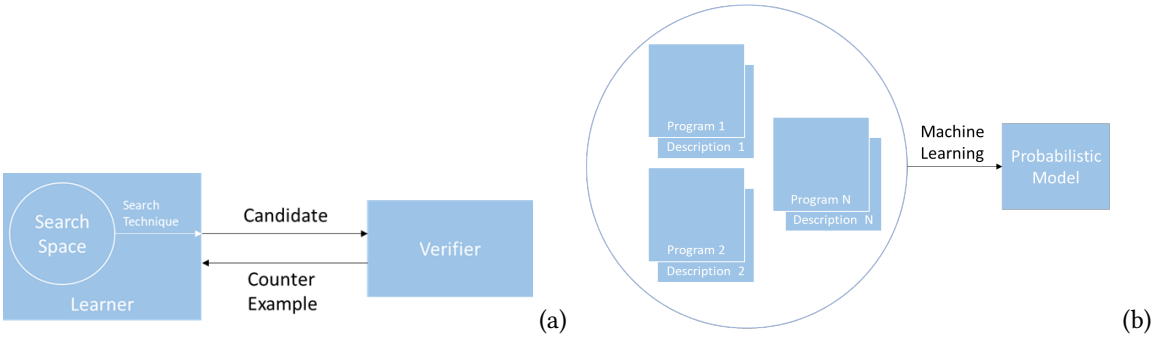



Fig. 1. Machine programming research directions. a) Generalization b) Specialization

While older research has been focusing on generating code using rule-based techniques (for example see **??** or **??**, two recent surveys in the area each focus on one of these two directions. The survey by Gulwani et al. [32] follows the first direction. It takes a formal-theory approach that is most commonly used in programming by example and program synthesis research. The other survey by Allamanis et al., [1], follows the second direction and takes a naturalistic approach. The most crucial difference between these two approaches is that the former does not consider an a priori model that the generated code should follow. It takes a set of constraints expressed in logical expression and grammar - also called domain-specific language - that specifies what programming statements are allowed to be used. It then searches over all possible combinations of the grammar to find a program that satisfies the constraint. The second approach, in contrast, assumes a model for the generated code. This model is learned by applying machine learning techniques to build a probabilistic model on how programming statements are most likely aligned. It will then use an informal description of the code to generate the program.

### 2.1 Survey of Program Synthesis

Following the generalization direction, Gulwani et al. [32] survey different program synthesis approaches. This survey is motivated by the challenges involved in understanding user intent by an algorithm that can generate code to realize that intent and techniques used to turn that into code. Several challenges contribute to this complexity, as the authors suggest. The first challenge is "the existence of a large program space." In its most general formulation, program synthesis is undecidable. Thus almost all successful synthesis approaches perform some kind of search over the program space. This search itself is a hard combinatorial problem. The second challenge is in synthesizing and interpreting user intent—different methods for expressing user intent range from formal logical specifications to informal natural-language descriptions or input-output examples.

The first area this survey focuses on is user intent. The user intent can be expressed in various forms, including logical specification, examples, traces, natural language, partial programs, or even related programs.

The second focus area is the search space can be over imperative or functional programs (with possible restrictions on the control structure or the operator set). The program space can be restricted to a subset of an existing programming language (general-purpose or domain-specific) or a specifically designed domain-specific language.

The last area studied in this survey is the search technique that can be based on enumerative search, deduction, constraint solving, statistical methods, or some combination of these.

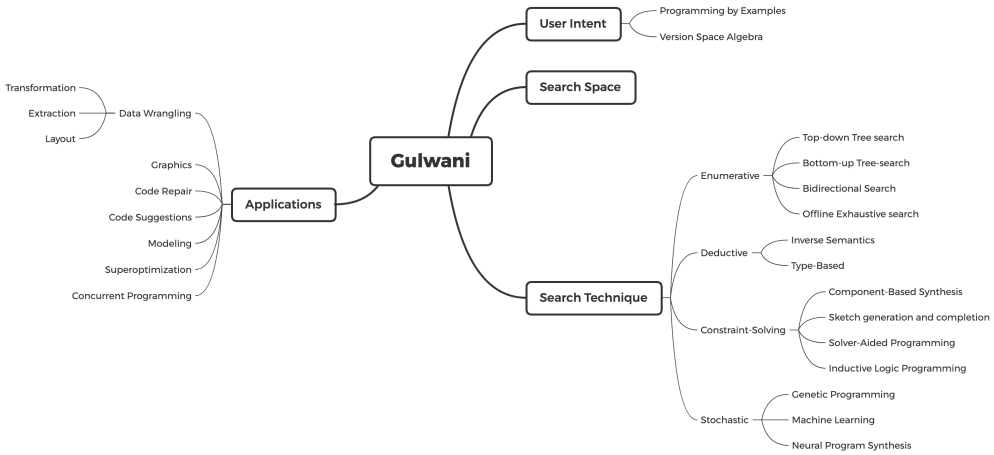



Fig. 2. Taxonomy derived from Gulwani's Survey

While this survey offers valuable insights on program synthesis based on constraints or examples, it also reveals somewhat inadequacy of such inputs to truly capture the user intent for more practical purposes. Other inputs such as natural language are a better fit for a certain class of tasks such as spreadsheet queries **??** and smartphone scripts **??**. Finding more elegant ways to express the user intent is one of the areas that machine programming research has shown more interest recently **??**. Machine programming approaches have also been proposed to express intent using various inputs such as examples, demonstrations, natural language, keywords, and sketches **??**.

Another area to consider for further investigation is around what we call specialization direction. The techniques studied in this survey have mostly leveraged the use of logical methods. While these techniques are good at leveraging the various operators' semantic knowledge or properties, they fall into scalability challenges. As we will see in this and other surveys, there have recently been very impressive advances in using deep learning methods to build scalable models.

## 2.2 Survey of Machine Learning for Big Code and Naturalness

Following the specification direction, Allamanis et al. [1] try to classify different models used in Big Code. The motivation behind this survey is the availability of billions of tokens of code and millions of instances of meta-data, such as changes, bug fixes, and code review - known as "Big Code." Availability of big code suggests, as the authors suggest - a new, data-driven approach to developing software tools. One can mainly use statistical distributional properties estimated over large and representative software corpora to design and create development tools or generate code.

This survey builds on top of what the authors call the naturalness hypothesis. This hypothesis holds that, because coding is an act of communication, one might expect large code corpora to have rich patterns, similar to natural language, thus allowing software engineering tools to exploit probabilistic ML models.

Given the ubiquity of the big code and assuming the naturalness hypnosis's correctness, this survey focuses on studying the available models built based on the code's characteristics. It categorizes models used in code generation into three types: code-generating models, representational models, and pattern mining models. These probabilistic models are created based on some key programming language characteristics - similar to the language models like BERT or GPT used in NLP to model natural language.

The first category is based on syntactic code characteristics like AST. These models specify how the code is written and can be used to generate code - this is where our surveys have the most commonalities.

The second category is based on code attributes like the types of variables. These models are used to infer a representation of the source code and, in practice, can be used to built tools like IntelliSense.

The last class is based on some semantic meaning of the code, like patterns for specific code defects. These models are used to extract semantic information from the source code like documentation or anomaly detection.

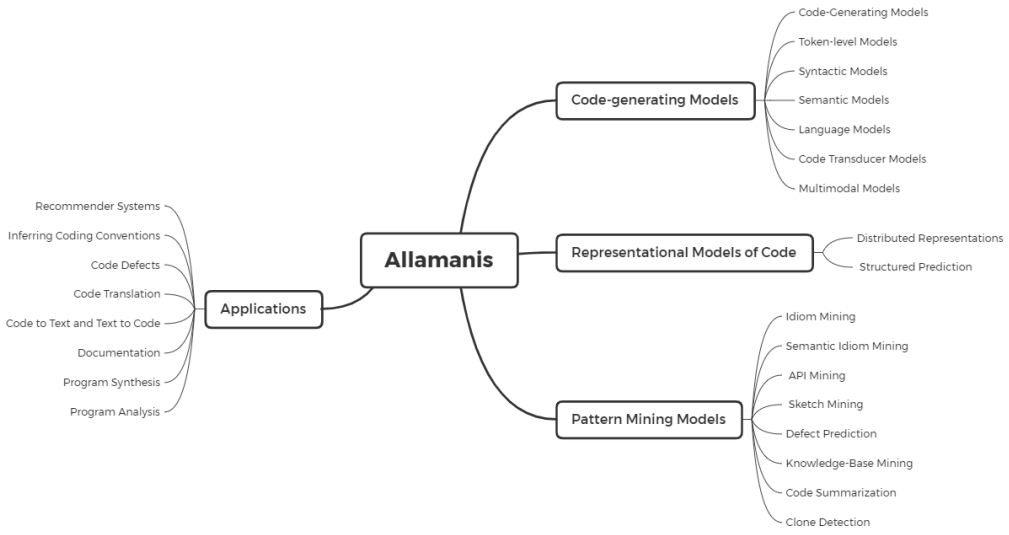



Fig. 3. Taxonomy derived from Allamanis's Survey

While understanding different types of models is a crucial part of what we call the technique within a machine programming approach, it is very important to know what type of models can be applied to specific types of input and/or to generate a specific type of output. In the generalization direction, the answer is clear, the input is always a set of constraints or examples and the output is a piece of code that satisfies the input. However, when it comes to specification direction, models can be applied to different types of input and can produce different types of outputs. Therefore it is important to be more specific about what combination across these three dimensions are valid and if so what specific problem they can solve. We spend more effort in our survey to make this connection.

## 2.3 Other Surveys

Another -less techniques-centric, more-language-centric - survey by [77] called “A Survey of Naturalistic Programming Technologies” studies programming languages and code generating tools based on their naturalistic characteristics. It focuses on the program’s structural differences and uses such characteristics as whether it contains English-like expressiveness or indirect inferences.

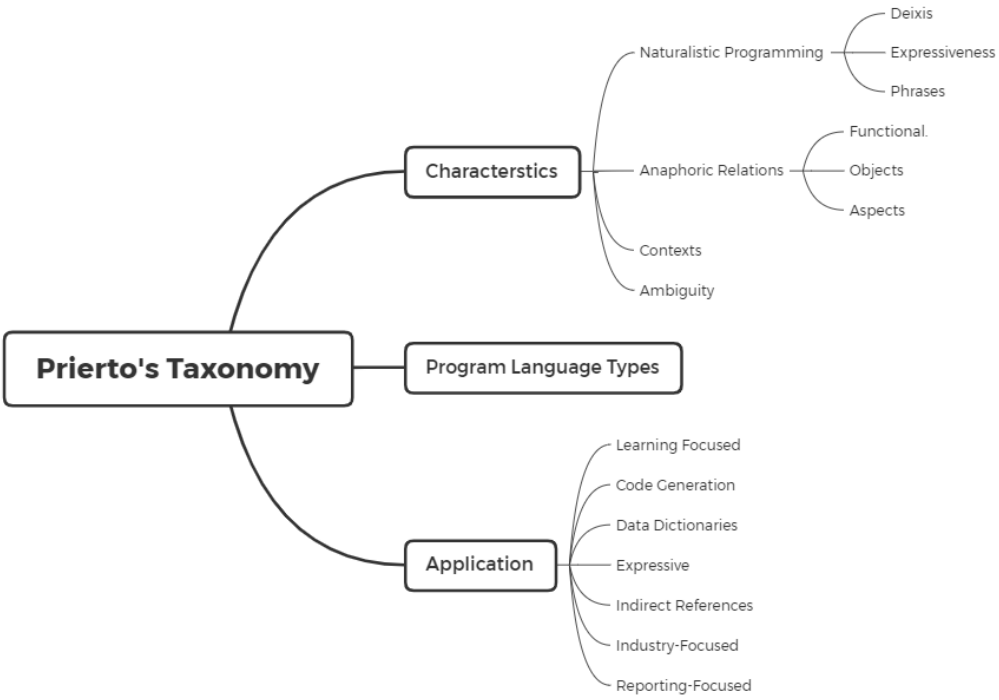



Fig. 4. Taxonomy derived from Pietro’ Survey

While these surveys have influenced our views about machine programming and helped us gain a detailed view on both generalization and specification directions, we felt that an overarching study that brings both of these views under the same umbrella was missing. We have therefore tried to fill up this gap in this survey. In this new survey, rather than focusing on how code is generated in a specific way, we offer a broader view of machine programming approaches across all possible combinations of inputs, outputs, and techniques and also look at how they are used to solve a specific problem.

# 3 TAXONOMY

Our taxonomy is designed with an end-to-end machine programming approach in mind. Therefore, it strives to offer a valuable tool to help with studying, understanding, and building on the research on this topic. As shown in Figure 5, the taxonomy provides five dimensions to support this objective: (1) *application* (what problem is being solved using the approach), (2) *technique* (the specific manner in which the input is converted into the

output), (3) *process* (how the technique is implemented and applied), (4) *types of input* (a description of desired software capability), and (5) *types of output* (some form of generated code).

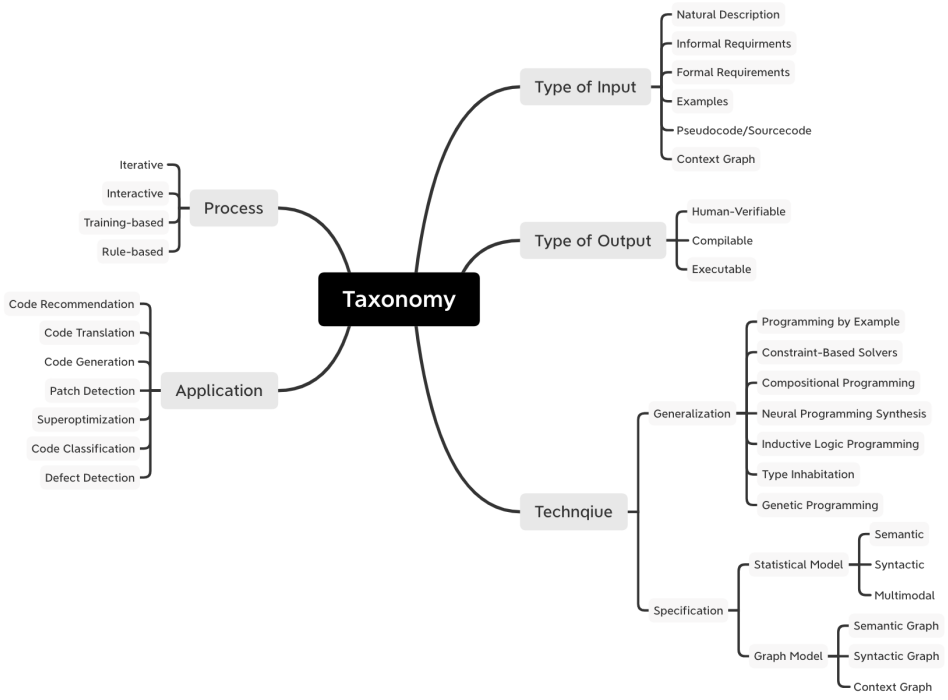



Fig. 5. Taxonomy

## 3.1 Application

The first dimension in our taxonomy, application, explains the machine programming problem the approach aims to solve. We can categorize the applications into two general groups regarding whether the used approach generates (1) new code or (2) an improved or alternative version of existing code. Code generation, code translation, and code recommendation belong to the first group, whereas defect detection, patch prescription, code classification, and super-optimization belong to the second group. It is noteworthy that non-machine programming approaches are traditionally used to perform these applications in the second group; however, the machine programming approaches can outperform traditional ones.

We use specific metrics to compare the gain obtained by the MP approaches in the following sections. These metrics include efficiency, compilability, verifiability, accuracy, and comprehensibility. These metrics are defined below:

- Efficiency is the amount of computing resources or time used to produce the outcome.
- Compilability is the number of errors or warnings once the compiler builds the code.
- Verifiability is the percentage of user intent realized by the code.
- Accuracy measures the ratio of falsely detected items to the total number of instances.
- Comprehensibility is a qualitative metric showing how human-readable the code is.

*3.1.1 Code Generation.* In this application, we would like to generate code that can satisfy that intent described user intent. Machine programming can increase efficiency by either replacing or aiding human developers with tools that can convert from any form of human-understandable description of the code - which are listed under the types of input - into code.

Code generation is the most challenging task in machine programming. The challenge in code generation is the result of two factors: containing ambiguity and dependency on context. For instance, consider the case where the input is the following 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 relatively simple sentence, it would not be possible to use any elementary intelligent technique to turn the sentence into executable code. The technique used to generate code from this requirement should detect and somehow refine the ambiguity in the word "fast." It also needs to understand what "driving" means. If so, it can relate "driving fast" to the "speed" and generate code to potentially use the device sensors provided to obtain a measurable value. Due to these challenges, the code generated by machine programming can be less compilable and verifiable compared to the code written by a human programmer.

*3.1.2 Code Translation.* Code translation takes the code in one programming language and translates it into another programming language. Note that there is a mechanical way of doing this task using a pair of compilers/decompilers in a limited fashion. For example, one could use available commercial tools to convert from C to Java. While such tools can save a lot of time, the generated code is not guaranteed to compile or be comprehensible. Most of such tools are incapable of translating segments that do not have similar constructs in both languages. For example, in the C to Java case, C LINQ statements or conditional compilations do not have any equivalent in Java and cannot be converted. Another limitation is translating UI-related code, which is rarely possible to convert due to significant differences in how various languages can be used to implement UI.

A machine programming approach can eliminate some of these limitations. For instance, it can be trained to produce a more "well-written" code; or use semantics rather than syntax to overcome cases where equivalent constructs are missing. Overall, MP approaches can increase compilability and comprehensibility.

Using a machine programming approach increases the risk of producing code that is not verifiable. This means that even though the original code could be verified to satisfy the user intent, the translated code might not.

*3.1.3 Defect Detection.* There are two ways that a machine programming approach can detect bugs. The first way studied more often in program synthesis research is based on static analysis, and abstraction [48]. Here, the condition for having a violation is explicitly provided by a human. The technique's job is to find code that can violate the condition. The second research in the machine learning community uses models that can detect anomalies by looking at existing patterns in code without requiring an explicit failure condition.

Being able to prevent bugs can reduce the cost of software development a lot. Fixing a bug detected during development is less expensive than fixing the bug discovered by manual QA and far less costly than fixing a bug caught in production [90]. Defect detection can generate unit testing code based on the expected output used to detect defects during development.

Each way of detecting bugs has its specific challenges. Like other program synthesis approaches, the former could be very resource-intensive and slow during the detection process. In contrast, similar to other specialization approaches, the latter requires much more resources and time during model development. While MP approaches can improve the number of true positives - incorrect code detected as a bug - they can also increase the number of false positives - correct code detected as a bug.

*3.1.4 Patch Prescription.* Patch prescription generates code that fixes known bugs in the provided code. Software patches change a system to address security flaws, fix the implementation of an algorithm, add features, boost code maintenance, improve portability, etc. Beyond these immediate effects, patches are a valuable source of

information for software researchers and developers. [**?** ] 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 [99].

Fixing a defect can have an immediate effect on improving the code functionality. However, there could be a long-term increase in the noise in software patches impacting their understanding, automated analysis, and use for tasks such as change prediction [**?** ]. The situation could become even worse when using a machine programming approach since such approaches pay less attention to non-functional aspects of applying the patch. These approaches can result in introducing more noise in the code.

*3.1.5 Code Classification.* Code Classification specifies a class for a given block of code. The classification is essential in many applications, such as selecting the best place to execute the code (CPU or GPU) [37].

While MP approaches can increase the efficiency in cases that they make a correct classification, they can have the inverse effect when a mistake is made (for example, if a program using heavy tensor calculation is classified as CPU-friendly.) Hence it is crucial that the existing approaches take the cost and benefit of both sides into account.

*3.1.6 Code Recommendation.* Code recommenders provide suggestions to complete a partially written code. Note that this can go well beyond the IntelliSense tools code's syntax.

While syntax-based recommenders are essential in today's programming, they can go beyond reminding what methods or properties are applicable at the current place of the cursor. However, MP-based recommenders can utilize semantics like other code in the present context, name of the methods or variables, or overall coding style of the programmer to recommend segments of code or argument completion in cases that a method has several overloads.

Compared to other applications, code recommendation has fewer challenges since more local structured information is available about the user intent - the surrounding code and context - therefore, commercial tools have been recently introduced to assist programmers in major IDEs - for an example see IntelliCode in Visual Studio [62].

*3.1.7 Superoptimization.* Code super-optimization is the task of transforming any given program to a more efficient version while preserving its input-output behavior. Note that this can go well beyond using compilers. While modern compilers implement a large set of rewrite rules and can achieve impressive speed-ups, they fail to offer any guarantee of optimality, thus leaving room for further improvement [15].

Superoptimization - being part of generalization approaches - relies typically on brute force methods, which can be very costly given the exponential size of the search space [74]. Finding ways to speed up the search algorithm is a challenge that has been under extensive research in recent years.

## 3.2 Techniques

Machine programming techniques are at the core of the research in machine programming. They contain the algorithms that turn the input into output. There are two types of techniques in 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.

*3.2.1 Generalization Techniques.* Generalization techniques try to find an implementation that can produce an expected result given some examples or constraints. They can be implemented as simple brute-force methods that search the entire space or sophisticated ones containing machine learning, composition-based, and evolving components as described in the following paragraphs.

**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 [83]. 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 significant 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 everyday tasks in the target domain) and tractability (for the synthesis algorithm to learn correct programs efficiently).

**Constraint-based 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, SMT's language 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 to discover a function that satisfies a given specification. Whereas traditional synthesis algorithms perform an expensive combinatorial search over all possible programs [39], new approaches take advantage of other approaches like machine learning to reduce this search space and improve the efficiency of SMT solvers [40].

**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 [65].

**Neural Programming Induction**: Most of the generalization problems are NP-hard. To tackle this, some researchers have leveraged neural networks that can either reduce the search space [66] or learn to select the best implementation from a set of candidates [31].

**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 [84]

**Type Inhabitation**: Type inhabitation - also know as type-driven program synthesis - uses type specification to find 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 an 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 essential consideration in type inhabitation. For example, [33] introduces a concise representation for type judgments that merges types into equivalence classes to reduce the search space.

**Genetic 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 [46].

*3.2.2 Specialization Techniques.* Specialization techniques assume a priori knowledge about the structure of the output. These models are built using state-of-the-art machine learning techniques trained over a large amount of existing code. At first glance, this may sound similar to the language models used in NLP. However, there is a significant challenge in producing programming language models. Programming languages - in contrast to natural language - follow specific syntactic and semantic rules to be compilable. An extensive amount of research has been devoted to finding models that incorporate such constraints.

**Statistical Models**: The statistical models use a priori knowledge in the source code and use it to either predicts 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 make predictions irrespective of the overall code's syntax and hence can produce code that could fail to compile [10]. Syntactic encoding models include a constraint that the generated code is required to be syntactically correct [101]. A third class jointly models source code and natural language to generate syntactically valid source code or meaningful natural language describing the code [2].

**Graph Models**: Statistical models have been successfully applied to capture programming patterns to support code completion and suggestion. However, they face challenges in capturing the patterns at higher levels of abstraction. These challenges seem to be due to the mismatch between such models' sequential nature and the nested 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 [69]. Another approach to create graph-based models is to use graph neural networks. In such models, both code and the neural network model are represented as graphs. The resulting graphs are then used to make code predictions [13].

*3.2.3 Measuring the Effectiveness of the Techniques.* The quality of any generated text can be measured in terms of its fluency and adequacy [91]; however, the quality of code depends on two additional metrics: 1) compilability 2) verifiability.

The traditional metrics used to evaluate machine programming models 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. 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. [43]
(3) **Perplexity** is a measure of how well a probability distribution or probability model predicts a sample [81].
(4) **Mean 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 [98].
(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 [72].
(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 [8].
(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 [55].

The traditional NLP metrics mentioned above do not suffice to measure the quality of the code. Therefore researchers have looked at new metrics that can capture them. For example, [53], offers a metric based on the generated code's quality and efficiency that encompasses six different generated source code properties.

## 3.3 Types of Input

Input is the expression of the desired intent of the software to be implemented. There are several ways to express the intent, which can be categorized based on the three following dimensions:

**Expressive versus exact**: The input can be expressed in a freeform manner where a human expresses her intent using natural language, drawings, examples, or a combination of all without following the consistent structure throughout. However, the traditional way of expressing software specifications follows a semi-structured style where requirements with specific sentence structure are used to convey the user intent. For example, many agile software processes recommend using this structure: "As a [user role] when [condition] I would like to see [outcome]." In the extreme case of formalism, boolean or mathematical expressions or a combination of both are used to express user intent for program synthesis. For example, this expression can be used to ask a synthesizer to find the max of two numbers: f(x,y) x  f(x,y)  y  f(x,y)  x,y [3].

**Brief versus comprehensive**: Human beings are excellent at discovering patterns based on few examples. For example, given only one pair of text transformations, showing a full name is abbreviated to initials. A human can apply it to a new character without requiring an explicit explanation of the task. Traditional ways of automating such tasks, however, required specifying many rules to describe such transformations. MP approaches are capable of acting similar to humans, i.e., to learn patterns based on only a few examples.

*3.3.1 Natural Description.* The input is in the shape of natural language without any restrictions. Natural language is the most expressive verbal way to utter the user's intent, allowing the users to use abstract, ambiguous, or incomplete ways to describe what they intend. This way of expressing intent, however, is a significant challenge in machine programming. The NLP community has exhaustively researched these challenges under the information extraction (IE) domain. Besides the NLP challenge, there are others specific to machine programming. These challenges include the required context knowledge, high level of ambiguity, and implicit referential meaning in the natural descriptions.

*3.3.2 Textual Requirements.* Textual requirements are written in natural language by a human analyst. Most of the software built in the industry is based on some sort of requirement. Recruitment can vary in terms of the level of details included, but overall they describe the functional - positive or negative - and nonfunctional scenarios that the software should implement.

Compared to natural descriptions, requirements are more structured and, therefore, easier to understand by a machine programming approach. However, given the still existing free form way of expressing information, IE is still a challenge - to a lesser extent than the natural description. Mapping relevant requirements to eliminate ambiguities is another challenge for MP. For example, given two requirements: "Driver should set the alarm when driving fast" and "The system should use the available public transport data to determine the maximum speed," there is no ambiguity if an MP approach can map them. Another difficulty arises when mentioning different roles in requirements that sometimes could refer to the same user. For example, in the driver example, a related requirement might say, "user should not see more than three alarms within one minute." Here user refers to the driver, but this referential relationship is not easy to detect for most MP techniques.

*3.3.3 Formal Requirements.* A formal software specification is expressed in a language whose vocabulary, syntax, and semantics are formally defined. A formal semantic definition means that the specification languages cannot

rely on natural language; they must be based on logical and mathematical statements. Using formal requirements to define the intent is prevalent in program synthesis research.

Formal requirements are less expressive than language-driven inputs, but they are far more exact. Therefore, they can be used in cases where no variations are accepted around the user intent. One prominent way to present such specifications is the language used to specify instances of the syntax-guided synthesis (SyGuS) problem. An instance of a SyGuS problem specifies the vocabularies, theories, and base types that are used to specify (a) semantic constraints on the function to be synthesized and (b) the definition of the function to be synthesized itself. [80]. The challenge in using formal requirements is the complexity existing in defining them. For example, defining the requirement to synthesize a linear equation in Sygus can take several lines.

*3.3.4 Examples.* Programming by examples (PBE) is a sub-field of program synthesis, where the specification comes in the form of input-output examples. Examples are the least expressive and most brief way of explaining the user intent. They are primarily used in text manipulation of mathematical relationships that are mainly used in spreadsheet applications.

Given the ease of providing examples, commercial tools are available that can take a few examples of what the user wants to do and generalize a pattern that can be applied to the rest of the data. For example, FlashFill is a data tool in Microsoft Excel that can combine, extract or transform data based on a few examples [30]. Examples, however, are not scalable to define larger or more complex intents. Therefore, they cannot be used beyond short text or relatively simple numerical applications.

*3.3.5 Pseudo-code and Source-code.* Pseudo-code provides a high-level structure of code but does not include all the necessary details. Source code or pseudocode can be used as the input to a machine-programming approach used to recommend missing code or improve an existing code. There are different ways of defining pseudocode. One variation that gained popularity in the 2000s is sketching, which expresses the high-level structure of implementation but leaves holes in place of the low-level details [92]. For example, the user intent to reversely order a list can be expressed as: "list reverseEfficient(list l) list nl = new list(); while( ) ," where  specifies the required details to be generated by the underlying technique. Source Code itself can be used as the input to MP to generate code in other languages.

It is easy to put together pseudocode by technical people. It can speed up code development by leaving some part of the code up to the MP approach. However, pseudo-code is not easy to write by non-technical people. It is also not a good way to express complex intent.

*3.3.6 Context Graph.* The context graph illustrates how the application state changes based on the user intent. 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, several users, each with several intents. These intents work together to define the context graph for the application. Recently there have been efforts to use a graph representation as to the input to the code generation techniques, anticipating that such representation would have a fundamental impact on the scalability and accuracy of the code generation techniques [70].

A context graph can relate requirements provided for an application and helps the MP technique to extract information and automatically disambiguate the intent or user roles specified in the requirements. Building the context graph, however, could be a challenging task. It requires an AI-based technique to process the list of requirements to construct the graph.

## 3.4 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.

*3.4.1 Human Verifiable Code.* Some of the techniques generate code 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. While non-compilable code requires manual post-generation work to be useful, it can reduce the amount of repetitive work significantly. It can also be used as an interim outcome, passed into more rigorous techniques to fix compilation issues. The challenge in this type of generated code is in the mechanism used to measure its goodness. NLP metrics can be used to offer such an evaluation but, in some cases, might not provide a very objective criterion.

*3.4.2 Compilable Code.* Some techniques generate code with a valid syntax - i.e., it can be mapped to a valid AST by a compiler. However, the code might not be invokable. For example, it can contain a run time error that causes it to crash. Another possibility is that it does not produce the expected outcome when it runs. compilable code requires less manual work to be useful, h. However, generating compilable code imposes significant challenges in MP techniques that have been the subject of research most recently.

*3.4.3 Executable Code.* Other techniques generate code that, once invoked, provides the expected result. This is the output produced by the generalization techniques and some of the specialization ones. This type of output is the best outcome for its usefulness and allows for evaluations/ This type of output is the most challenging type of code to produce since it needs both syntactic and semantic validity. The verification itself can be challenging when the input is not provided as examples or formal requirements.

## 3.5 Processes

An aspect of machine programming studied less by the research community is how the techniques are derived and applied. It would make sense to ignore the process for smaller programs like a code segment, a bug fix, or a unit test since the techniques introduced earlier would generate them in one shot. However, when considering applications and systems as the output, 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.

*3.5.1 Iterative.* In iterative processes, the output is generated in multiple iterations. Each iteration can extract some information from the input or create a portion of the output [17].The generated output can be verified for correctness, and discovered issues can be looped backed and used as an input for subsequent iterations. This is similar to how reinforcement learning works, where the system can continuously learn from past mistakes and take corrective actions to achieve a better result [16]. In An MP iterative process, the number of iterations and what should be covered in each iteration should be specified in real-time based on what information has been provided in previous iterations, which could be a challenging task. Making a decision when to stop iterating is another challenge in such processes.

*3.5.2 Interactive.* In interactive approaches, the user is asked to provide the input throughout the process, not just at the beginning. Each interaction can result in reducing gaps in information existing in the input [28]. The interactive processes are more appropriate for larger outputs like applications or systems. In such scenarios, users can better understand their intent as they see the implementation of their previously expressed intents - this is known as IKIWISI (I know it when I see it in software engineering.)

An interactive process is usually implemented as a conversational AI system. The agent can memorize the past state and use it to find the gaps, translate the gaps into appropriate questions, and process the response

provided by a human. This requires a complex natural language processing system which includes state-of-the-art information extraction [103], attention mechanisms to utilize history bahdanau2014neural, and specialized language models [78].

*3.5.3 Training-based.* Training-based approaches are prevalent when the underlying technique uses machine learning. In such processes, 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. Training-based processes are easy to implement. Having access to enough training data, one can use both supervised and unsupervised methods to generate models that can predict the following line of code given the provided source code [95]. Training-based processes require a large amount of training data to create the model; however, datasets that can map natural languages to code are not prevalent. Producing such datasets requires designing elaborate rules for tagging and laborious efforts to apply them. These models can be applied to smaller code fragments, but we need to rely on iterative and interactive processes when for larger systems.

*3.5.4 Rule-based.* Most generalization techniques like PBE rely on rule-based techniques like version algebra or enumeration to generate programs. These techniques, while calculation-intensive, can produce all possible programs satisfying a set of constraints. Rule-based processes can outperform learning-based approaches in terms of accuracy when applied to smaller deterministic programs. However, using such techniques on larger stochastic programs like applications that consist of user interactions, complex user intent, and high-level properties like soundness, responsiveness, and user-friendliness requires specific theories on such programs that do not exist.

## 4 EVALUATING EXISTING APPROACHES

This section reviews the current 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 with several possible intents, each applicable at a given state of the application. For larger applications, there can exist more users, each with several intents. User intent representation has a fundamental impact on all three metrics that we defined earlier to differentiate the machine programming techniques. Researchers have used various types of inputs and their combinations to represent the intent. Each kind of input provides one or more forms of the information listed below:

**Syntactic Form**: includes 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 for its position in the tree.

**Shallow-Semantic Form**: provides a limited amount of information on what the features mean beyond its syntactic information. For example, the name given to a variable or method offers insights into what its function is intended to be.

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

*4.1.1 Natural Language Description as Input.* Using natural language to describe input has been receiving increasing attention due to natural language processing (NLP) advances. In the studies that we have surveyed, it comes at the top rank to describe the user intent. Natural language can capture much more information than examples, and it is much easier for non-technical people to produce than 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

| 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 |
| Pseudo/Source code | Yes | Yes | Yes |
| Context Graph | Yes | Yes | Yes |

Table 1. information form provided by each type of input

information. The second one, a systematic problem, is the complexity involved in disambiguating the natural language. Techniques exist 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 [56], [39], [24], [12]. For example [56] applies a complex neural network based on the attention model introduced by [18] to convert code in Hearthstone card games. As shown in Figure 6, they use specific components to extract information from different areas of text.

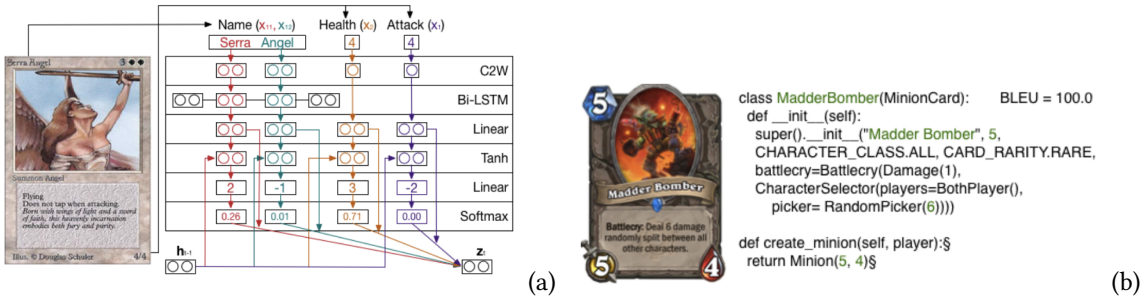



Fig. 6. [56] uses text input provided in Hearthstone cards to generate code. a) The model used to convert text to code. b) An example input and output.

However, there are several studies that generate code that can only be compiled but fall short in generating the expected output [101], [2], [79], [41] or would not compile at all but are close to the code that could have been implemented by a human [68], [19], [34], [87]. Researchers have used both generalization and specialization techniques to transform natural language descriptions into code. Among generalization techniques, inductive logic programming [24], [19], [41] and neural programming synthesis [68], [73], [86] techniques are used most often. Statistical syntactic model is the prominent technique among the specialization ones [34], [52], [64], [59]. 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 a generalization technique applied to natural language description, [68] 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. On the other hand, as an exciting specialization technique based on natural language description [52] 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, translate the English specification into a specification tree, and then translate it into a C++ input parser.

*4.1.2 Pseudo/Source Code as input.* Pseudo/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], [42], [57], shallow semantics via the name assigned to variables and methods [47], [76], 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 in 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 [42], [41] and in the latter it helps a programmer to complete their partially written code [35], [69], [82], [21]. For example, as shown in Figure 7, [82] uses a CRF-based approach [50] to assign proper names and types to in input JavaScript programs.

```
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. 7. [82] 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 the actual intent. Finally, most of the approaches that use source code as input rely on a training process to create their model. Training is more convenient 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 [100], [58]. The other reason for this is the abundance of available public source code that can be used for creating statistical models [9].

*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 [54], [24] or simple text manipulation tasks [51], [84], however, they normally fall short to process exhaustive requirements that are required for larger applications. As an example in [84],

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 examples' transitions. Figure 8 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(
    CharTok('G'),
    Interval(NumChar, 1, 5)),
  ConcatTok(
    CharTok('G),
    ConcatTok(Interval(NumChar, 4), UpdateChar))
  )
)
```

(b)

Fig. 8. [84] 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 the majority of them use statistical generalization technique like neural programming synthesis to search for the solution [54], [17], [88], [61]. Some use deterministic generalization techniques like version space algebra [28], [75] to find all possible programs satisfying the examples and then use ranking algorithms to select the most appropriate one - which could be the shortest or the fastest. When examples are used as the input, the majority of the approaches are used to generate executable code. Producing code that is not executable is not acceptable here since the provided input and output examples leave no ambiguity in what the generated code is intended to do.

Some researcher have used examples as a complement to other input types like source code or natural language. For example, [39] synthesizes a program iteratively from a library of existing code components and use examples to rank the synthesized programs. [84] proposes an approach that combines examples and natural language to generate code used for synthesizing algorithms 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 constraint-based solvers to find programs that satisfy them [22], [94]. 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 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 - in our view - essential to produce software at the scope of a standalone application or systems, Still it has not been the focus of machine programming research so far.

## 4.2 Across the Types of Output

The majority of the studies we have surveyed aim to produce code that can execute, which is of no surprise since machine programming's ultimate goal is to generate fully-functional applications that can execute to carry out the user intent. However, when input is neither precise enough nor easy to extract the existing information from, producing executable code becomes challenging. Therefore many studies suffice to either generate code that is

compilable Without a high accuracy in producing expected results [101], [42], [79] or even produce code that contain syntactic errors [37], [45], [67].

*4.2.1 Executable Output.* The majority of the surveyed studies generates 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 [88], [61]. Among generalization techniques, neural programming synthesis is the major technique that produces executable code [21], [73]. 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 a large number of candidates [88]. Other approaches like version space algebra normally use computationally heavy operations and heuristics that are less effective in refining the candidate solutions [28]. As an example [73] presents a generalization technique that learns to generate a program incrementally without the need for a detailed search. Once trained, this approach can automatically construct computer programs consistent with any set of input-output examples provided at test time. Figure 9 provides a few examples that this approach can handle. Note that, as opposed to PBE shown earlier, being a neural network, this approach can only predict the output without offering an explicit source code.

| Input | Output |
|---|---|
| [CPT-00359] | [CPT-00350] |
| [CPT-00340] | [CPT-00340] |
| [CPT-114563] | [CPT-114563] |
| [CPT-1AB02] | [CPT-1AB02] |
| [CPT-00360] | [CPT-00360] |

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

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

Fig. 9. [73] uses a neural programming synthesis technique to produce outputs following the patterns observed in provided input examples.

Although most of the approaches generating executable output measure their validity by showing they can produce the expected outcome for a given input, a few studies use a different measure. For example, [71] 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 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 machine learning approach's ability to generate valid code and the size of the program it produces. Whereas generalization techniques focus on the former factor, specialization techniques tend to create larger applications [64]. They often generate non-executable nor code; however, it is very close to what could have been written by a human programmer. These approaches use a training-based process to create a model to predict code given the described user intent. They usually train their model over a vast repository of public code [9] and, therefore, can produce code for almost any provided input. However, due to their statistical nature, they do not necessarily generate an entirely correct code. For example,

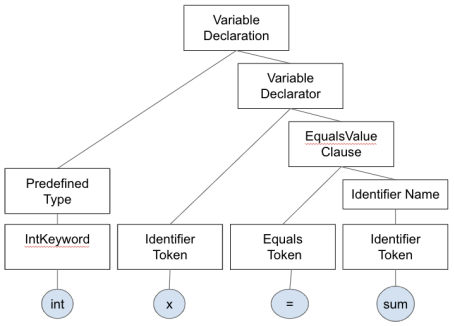
[64] uses a recurrent neural network to generate code from the user intention. Although the generated output is very close to the expectation, it contains several errors, as shown in Figure 10.

```
#include<stdio.h>
int main () {
  int n,i,j,sz[100].max=0.ci=0;
  scanf("%d", &n);
  for (i=0;i<n;i++) {
    scanf("%d", &sz[i]);
    if (sz[i]>max) {
      max=sz[i];
    }
  }
  for (i=0;i<n;i++) {
    if(sz[i]>ci&&sz[i]<max) {
      ci=sz[i];
    }
  }
  printf("%d\n%d",max,ci);
  return 0;
}
```

```
#include<stdio.h>
void main () {
  int x,a[100],i,max1.max2;
  scanf("%d", &n);
  for (i=0;i<=n-1;i++) {
    scanf("%d", &a[i]);
    if (a[i]>max1)
      max2=a[i];
  }
  for (i=1;i<n;i++) {
    if(a[i]>max&&a[i]==max1)
      max2=a[i];
  }
  printf("%d\n%d",max1,max2);
  return 0;
}
```

Fig. 10. [64] 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 balance the executability and size of the output. They produce compilable code but do not necessarily produce the expected outcome. An overwhelming majority of such studies use specialization techniques and rely on syntactic models to produce code that is syntactically correct [58], [42]. Almost half of the studies in this category are related to code generation solutions [102] and the other half are related to code recommendation solutions [70]. For example, [58] uses a probabilistic context-free grammar to model the abstract syntax tree for the C compiler and uses it to produce code that maps to a valid AST and therefore can compile. Figure 11 shows an example of AST used and the code generated by their approach.




```
for (int i=words; i<4; i++) {
  for (int j=0; j<i; ++j) {
    if (words.Length % 10 == 0) {
      Math.Max(j+j, I*2 + Max.Abs(i+j));
    }
  }
}
```

Fig. 11. [58] 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, [42] 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 are used in machine programming: generalization techniques try to generalize from a few examples and generate programs that can be applied to similar inputs. Specialization techniques, however, try to learn generic models from large datasets and can be applied to a broader 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, including statistical components, is increasing. Specifically, neural programming synthesis that uses a neural network to refine the solution space dominates generalization-based studies. Another observation is that, among specialization techniques, the majority of techniques are based on syntactic models. This could be due to the challenges existing in extracting shallow and deep semantic information from the provided input. The last observation on graph-based techniques is that although they are not being used as much as other techniques, they are 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 12, NPS techniques apply machine learning to refine the solution space derived using generalization techniques. For example, [49] studies the problem of efficiently predicting a correct program from a large set of programs induced from a few input-output examples. It is essential to make programming by example techniques work without requiring to have access to too many examples. Hence, these techniques first use an analytic method like version-base algebra to define the solution space and then use a supervised machine learning method to obtain a hierarchical ranking function and predict a correct program efficiently.

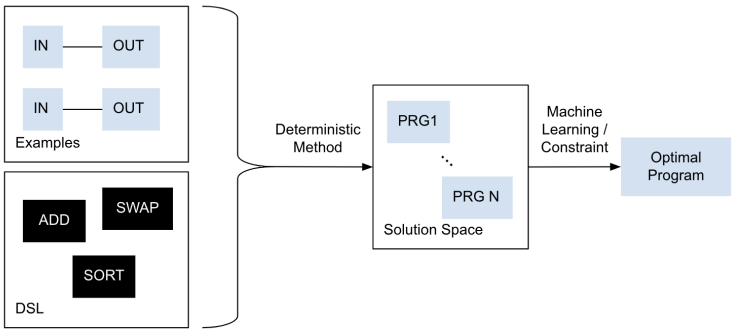



Fig. 12. Generalization Techniques

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

Another widely used generalization technique is inductive logic programming (ILP). ILP is the intersection of inductive learning and logic programming **??**. ILP inherits its goal from inductive machine learning: developing

tools and techniques to induce hypotheses from observations (examples) and 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 target predicate's view definition. Figure 13 shows an example of a few datasets on family relationships; we can use ILP to infer new rules defining father and mother.

```
parent(a,b)     parent(a,c)     parent(d,b)
father(a,b)     father(a,c)     mother(d,b)
male(a)         female(c)       female(d)

   father(X,Y)  :- parent(X,Y) & male(X)
   mother(X,Y) :- parent(X,Y) & female(X)
```

Fig. 13. 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 14, [54] provides a technique that combines elements from compositional and statistical methods to synthesize a program by composing partial programs.

```
[] → []
[[1]] → [[]]
[[1,3,5], [5,3,2]] → [[3,5], [5,3]]
[[8,4,7,2], [4,6,2,9], [3,4,1,0]] → [[8,4,7], [4,6,9], [3,4,1]]
```

(a)

```
dropmins x = map f x
  where f y = filter g y
    where g z = foldl h False y
      where h t w = t || (w < z)
```

(b)

Fig. 14. The generated code from compositional technique in [54] given the provided input. a) Provided examples where the minimum number in each set is dropped. b) A program composed of several sub-programs within the defined DSL, including map, where, fold1, False, 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 data requires new effective techniques like graphs and contextual models, which have been studied less frequently [79]. Therefore most techniques rely on syntactic information, which they can easily extract from source code. Another reason to favor syntactic techniques is that being driven by syntactic representations like AST [79] they can produce compilable code. Some studies try to use both syntactic and semantic models. Such approaches tend to generate code that is both compilable and meaningful [82].

Almost half of the studies we have surveyed use syntactic techniques to train models for code generation [59]. However, several are focused on code recommendations solutions [57]. As shown in Figure 15, these

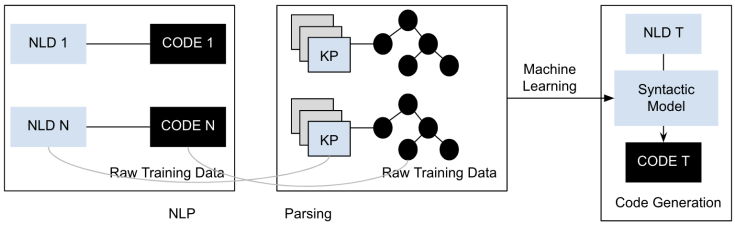



Fig. 15. Specialization Techniques

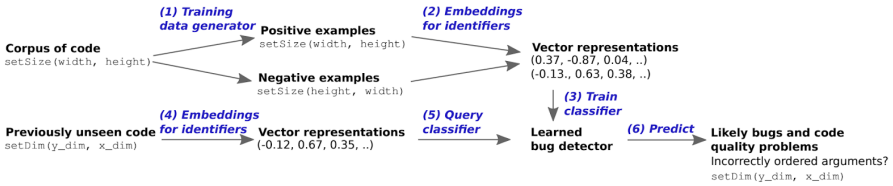
approaches rely on a massive dataset of natural language description (NLD) to code mappings. They first process this raw dataset to produce key phrases to AST parings and use them as training features. Finally, a machine learning approach like deep learning is used to generate a model that can be used for code generation. Whereas generalization techniques use examples and produce executable programs, specialization techniques take pseudo/source code or natural language as input and produce human-verifiable programs. This observation perfectly aligns with our expectation that we stated earlier as the trade-off between the generated code's accuracy and size. 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 a training-based process.

The next most used specialization technique is semantic statistical. While in the majority of cases, semantic techniques are used to solve code generation [49] or code recommendation [82] problems, they attempt less to solve code translation [42] and defect detection. They tend to use pseudo/source code more often than natural language description as their input and produce compilable code less frequently compared to syntactic techniques. This makes sense since these techniques focus less on syntax and more on the meaning of the input's information.

In the studies that we have surveyed, we have seen semantics being used as both shallow semantics and deep semantics. As an example for the former case, [76] 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 16.

As an example of using deep semantics, [21] presents a model that integrates components that 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 runs partially written programs, exposing their semantics. The REPL addresses a fundamental challenge of program synthesis: tiny syntax changes can lead to considerable changes in semantics changes. 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, some researchers have studied them more 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 bug nodes' position and corresponding graph edits to produce a fix. [37] presents a new graphical structure to capture semantics from code using a semantic graph called program-derived semantic graph (PSG). The PSG provides a single structure that can capture program semantics at many levels of granularity. As shown in the figure 17, this allows PSG to understand whether two blocks of code with different syntax have similar semantics.




(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. 16. 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. [76]. a) Overview of the DeepBug Approach. b) A program with a bug that is detected by DeepBug.

**Implementation 1**

```
signed int recursive_power(signed int x, unsigned int y) {
   if (y == 0)
      return 1;
   else if (y % 2 == 0)
      return recursive_power(x, y/2) * recursive_power(x, y / 2);
   else
      return x * recursive_power(x, y/2) * recursive_power(x, y / 2);
}
```

**Implementation 2**

```
signed int iteative_power(signed int x, unsigned int y) {
   signed int val = 1;
   while (y > 0) {
      val *= x;
      y -= 1;
   }
   return val;
}
```

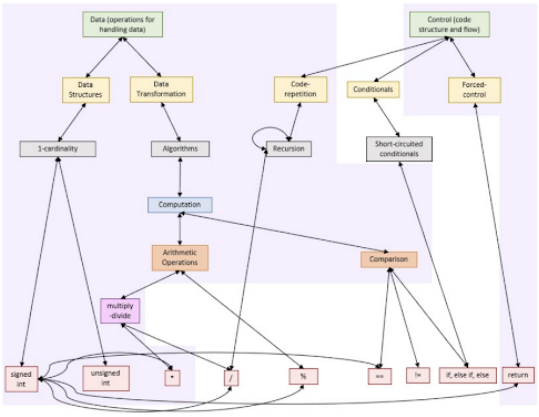



Fig. 17. Two different implementations of exponential function ($x^y$). The first one is recursive whereas the second one is iterative. The shaded area shows the overlap in the PSG for both approaches which is about 70%. Other approaches fall short of providing such a high overlap [37].

## 4.4 Across Solutions

While, in the general, machine programming is used to generate code, it can be applied to solve many different problems, including - but not limited to - code recommendation, bug detection, and patch prescription.

*4.4.1 Code Generation.* The majority of the studies that we have surveyed use the mentioned try 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 pseudo/source code. We have already reviewed several cases of using either NLD or examples to generate code.

As a general trend, most code generation solutions rely on neural programming synthesis (NPS) followed by syntactic statistical-based techniques. NPS is used when the provided input is an example, whereas the syntactic statistical technique is used when the input is NLD. The majority of the code generation solutions use training to 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, 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 18, [6] uses machine learning to map from input-output examples to program attributes which are then used to select a search procedure that searches program space.

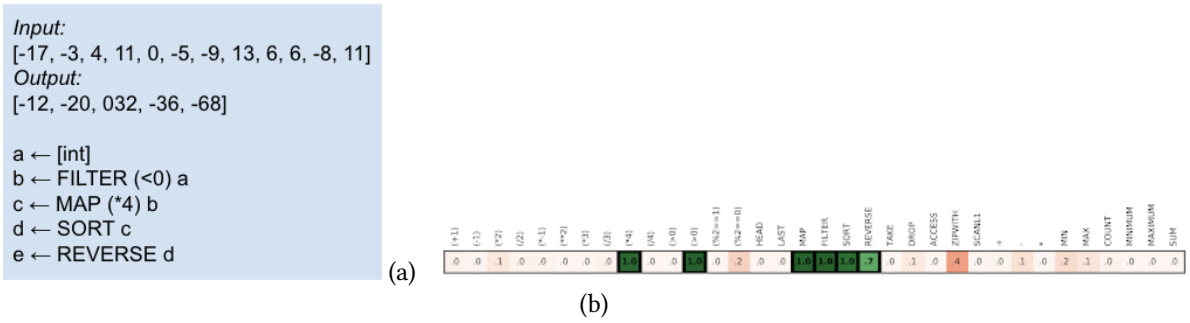



Fig. 18. DeepCoder tries to solve generated code for input/output examples using neural programming synthesis augmented by search techniques. a) Input provided and the generate code b) The predictions of a trained neural network made from input-output examples for the program shown in (a).

*4.4.2 Code Recommendation.* The second most studied solution is code recommendation. Code recommendation is an easier problem to tackle than code generation due to two reasons. First, most approaches can easily use the existing large repository of source code to build statistical predictive models. Second, the result can be verified easily either by syntax checking or human-verification.

Most of the code recommendation studies use pseudo/source code as their input. 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 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. Still, they can be validated by humans or checked for a valid syntax using compilation.

Some studies incorporate syntactic components into their models to make sure the recommended statement is syntactically correct. For example [58] 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 [99] defect detection [89], and code translation [42]. For example, [47] uses an interesting approach to detect defects without explicit rules. It uses an annotation factor graph (AFG) shown in Figure 19 to specify specifications. There are two types

(a)

```
FILE * fp1 = fopen(“myfile,txt”, “r”);
FILE * fp2 = fopen(fd, “w”);
fread(buffer, n, 1, fp1);
fwrite(buffer, n, 1, fp2);
fclose(fp1);
fclose(fp2);
```

(b)

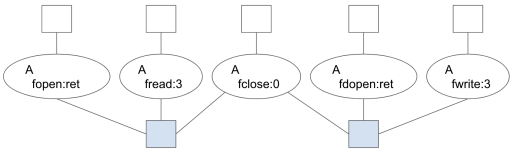



Fig. 19. AFG for a sample program that maps both check factors and beliefs to variables used in the code. a) A program with potential bugs if pointers are not de-allocated before functions returns. b) Factor graph model for the code. Circular nodes correspond to variables and square nodes to factors. The shaded factors indicate check factors, while the top row depicts factors modeling prior beliefs.

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 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 most cases. However, given that the most used type of input, NLD, is not structured, rule-based processes cannot 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 with 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, [51] 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 both iterative and interactive process called CodeHint that helps programmers write difficult statements. Figure 20 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 Java grammar. Third, it combines the candidates to produce larger expressions. In the last iteration, CodeHint uses its probabilistic model to avoid searching for other 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.

## 5 CONCLUSION AND FURTHER STUDIES

This survey provided a taxonomy that offers 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 can we use to explain the user intent and what will be generated as the output?
(2) How is the transformation performed?
(3) Why is the transformation important, and what problem can it 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 current gaps in the five-dimensional space of

```
final JComponent tree = makeTree();
tree.addMouseLocation(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
  int x = e.getX(), y = e.getY();
  Object o = null;
  // Get the menu bar or the clicked element
```

(a)

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

(b)

Fig. 20. 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.

{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 the available studies consider a description that is either completely understood or clarified by supplemented examples. However, such assumptions are generally invalid when considering real-world software projects where requirements often contain ambiguities that need to be detected and eliminated before machine programming can occur.
(2) Considering descriptions that are inter-related. The existing studies only consider a sentence or a paragraph that contains all the required information. However, in reality, we face a set of interrelated requirements that should be processed and mapped using a data structure that 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 is 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 determine the requirements assume a set of the underlying contexts. 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 evaluate generating code for larger scopes like a module within an app or an entire app are entirely missing. Note that we cannot simply assume that we can use a divide-and-conquer approach to build complete applications once we can generate code at the fragment level. 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 considering all the requirements together.

We think graph-based approaches are the only candidate representing requirements beyond a simple, refined sentence on the technique dimension. Recently there have been some studies that look at 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 advances 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 an end-to-end application and systems containing several applications. The gap in this area is potentially due to the lack of required research and tools in the other dimensions.

## 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] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

[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] Imran Sarwar Bajwa, M Imran Siddique, and M Abbas Choudhary. 2006. Rule based Production Systems for Automatic Code Generation in Java. In *2006 1st International Conference on Digital Information Management*. IEEE, 300–305.

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

[7] Robert Balzer. 1985. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering* 11 (1985), 1257–1268.

[8] 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.

[9] 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.

[10] 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.

[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] Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. 2016. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787* (2016).

[16] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276* (2018).

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

[18] 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.

[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] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 69–80.
[28] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[29] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.
[30] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.
[31] Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL meets ML. In *Asian Symposium on Programming Languages and Systems*. Springer, 3–20.
[32] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
[33] 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.
[34] 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).
[35] 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.
[36] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.
[37] 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).
[38] Jinseong Jeon, Xiaokang Qiu, Jeffrey S Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 934–937.
[39] 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.
[40] 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).
[41] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.
[42] 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.
[43] 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.
[44] Emanuel Kitzelmann. 2009. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*. Springer, 50–73.
[45] John R Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and computing* 4, 2 (1994), 87–112.
[46] John R Koza and James P Rice. 1992. Automatic programming of robots using genetic programming. In *AAAI*, Vol. 92. Citeseer, 194–207.
[47] Ted Kremenek, Andrew Y Ng, and Dawson R Engler. 2007. A Factor Graph Model for Software Bug Finding.. In *IJCAI*. 2510–2516.
[48] Daniel Kroening, Alex Groce, and Edmund Clarke. 2004. Counterexample guided abstraction refinement via program execution. In *International Conference on Formal Engineering Methods*. Springer, 224–238.
[49] 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.
[50] John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001).

[51] 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.
[52] 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.
[53] Zhen Li, Ying Jiang, Xiao Jiang Zhang, and Hai Yan Xu. 2020. The Metric for Automatic Code Generation. *Procedia Computer Science* 166 (2020), 279–286.
[54] 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.
[55] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.
[56] 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).
[57] 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.
[58] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *International Conference on Machine Learning*. 649–657.
[59] Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating Programming by Example and Natural Language Programming.. In *AAAI*.
[60] 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.
[61] 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.
[62] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[63] 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.
[64] 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).
[65] Stephen Muggleton. 1992. *Inductive logic programming*. Number 38. Morgan Kaufmann.
[66] Stephen Muggleton, Cao Feng, et al. 1992. Efficient induction of logic programs. *Inductive logic programming* 38 (1992), 281–298.
[67] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2017. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698* (2017).
[68] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834* (2015).
[69] 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.
[70] 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.
[71] 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.
[72] 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.
[73] 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).
[74] 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.
[75] 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.
[76] 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.
[77] Oscar Pulido-Prieto and Ulises Juárez-Martínez. 2017. A survey of naturalistic programming technologies. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–35.
[78] Chen Qu, Liu Yang, Minghui Qiu, W Bruce Croft, Yongfeng Zhang, and Mohit Iyyer. 2019. BERT with history answer embedding for conversational question answering. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1133–1136.

[79] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).
[80] Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. 2019. The SyGuS Language Standard Version 2.0.
[81] 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.
[82] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.
[83] 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.
[84] 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*.
[85] 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.
[86] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.
[87] 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.
[88] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*. Springer, 398–414.
[89] 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.
[90] Sandra A Slaughter, Donald E Harter, and Mayuram S Krishnan. 1998. Evaluating the cost of software quality. *Commun. ACM* 41, 8 (1998), 67–73.
[91] Matthew Snover, Nitin Madnani, Bonnie Dorr, and Richard Schwartz. 2009. Fluency, adequacy, or HTER? Exploring different human judgments with a tunable MT metric. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*. 259–268.
[92] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Citeseer.
[93] Phillip D Summers. 1977. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)* 24, 1 (1977), 161–175.
[94] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability modulo recursive programs. In *International Static Analysis Symposium*. Springer, 298–315.
[95] Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. 2020. Fast and Memory-Efficient Neural Code Completion. *arXiv preprint arXiv:2004.13651* (2020).
[96] Wolfgang Thomas. 2008. Solution of Church's Problem: A tutorial. *New Perspectives on Games and interaction* 5, 23 (2008), 3.
[97] Philip R Thrift. 1991. Fuzzy Logic Synthesis with Genetic Algorithms.. In *ICGA*. 509–513.
[98] Ellen M Voorhees et al. 1999. The TREC-8 question answering track report. In *Trec*, Vol. 99. 77–82.
[99] 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.
[100] 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.
[101] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).
[102] 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.
[103] Jiansong Zhang and Nora M El-Gohary. 2016. Semantic NLP-based information extraction from construction regulatory documents for automated compliance checking. *Journal of Computing in Civil Engineering* 30, 2 (2016), 04015014.
[104] Ruiqi Zhong, Mitchell Stern, and Dan Klein. 2020. Semantic scaffolds for pseudocode-to-code generation. *arXiv preprint arXiv:2005.05927* (2020).