Programming in the Age of Intelligent Machines

Breandan Considine

August 13, 2021

1 Introduction

Since the invention of modern computers in the mid 20th century, computer programming has undergone a number of paradigm shifts. From the rise of functional programming, to dynamic, object-oriented, and type-level programming, to the availability of myriad tools and frameworks – its practitioners have witnessed a veritable Renaissance in the art of computer programming. With each of these paradigm shifts, programmers have realized new conceptual frameworks for reasoning and expressing their ideas more clearly and concisely.

Over the last few years, another paradigm shift has been set in motion, with significant implications for how we think about and write programs in the coming century. By most measures, computers have grown steadily more intelligent and capable of assisting programmers with mentally taxing chores. For example, intelligent programming tools (IPTs) powered by neural language models have this year helped over 10 million unique human beings program computers. As IPTs help digitally illiterate communities to discover their innate aptitude for computer programming, this population will continue to rise.

Computer programming is a uniquely creative exercise among the range of human activities. It channels our innate linguistic, logical, imaginative, and social abilities to bring abstract ideas into reality, and ultimately, gives humans the freedom to create new realities of their own design. In collaboration with other humans and the increasing participation of IPTs, vast and elaborate virtual worlds have been manufactured, where the majority of humankind now chooses to spend their daily lives. With the expanding opportunities these new digital frontiers promise, their population too will continue to grow.

Today IPTs share an equal role in shaping many aspects of computer programming, from knowledge discovery to API design, and program synthesis to validation and verification. However, this balance is shifting beneath our feet. Once its creators, programmers are now primarily consumers of information provided by an IPT, and increasingly rely on them to perform their daily work. With the unique opportunities and risks this partnership presents, what division of labor should exist between humans and our new coding collaborators? This is the question we have set out to understand in the following literature review.

2 Neural Language Models for Source Code

Programming researchers have long held an interest in using intelligent tools to help them write programs [10]. Due to fundamental limitations in data and processing power, many of these ambitions had not come to pass until the last few years, thanks to the availability of *big code* [2], the development of differentiable programming libraries for gradient-based learning [7], and attention-based language models [31], among other technical achievements. Armed with this new repertoire, programming researchers have revisited their interest in IPTs. Naturally, one of the first applications considered was code completion.

Following their initial success in natural language, rapid progress continues to be made in the application and specialization of transformers to source code, as well as industrial transfer where this technology is now trained and deployed on millions of programmers worldwide [11]. Given a natural language description of an incomplete method, these models are capable of inferring programmer intent and completing multiline code snippets. Progress is expected to improve.

The problem comes down to a question of grammar induction. Based on empirical results, fixed-precision transformers (e.g. GPT-2, BERT) are thought capable of recognizing the class of counter languages [9], i.e. somewhere between context-free and context-sensitive, although this characterization requires a more careful theoretical analysis. For source code typically stored on GitHub, this class would appear to suffice – models trained on such datasets are currently capable of sketching rudimentary programs and boilerplate code templates, however more complex fragments require additional oversight.

An important shortcoming of imitation learning is the question of data provenance and validation: even if the training data is syntactically well-formed, constraints on the class of valid programs are ill-posed. As a consequence, a large fragment of languages generated may be syntactically valid but semantically unsound, i.e. may throw undesired runtime errors, or appear to work at first, but are in fact broken in a subtle manner. Like most language models of its kind, performance is highly sensitive to the dataset quality, as common errors in the training data are prone to be inherited and reproduced by an IPT.

The vast majority of modern programming consists of writing ceremonial boilerplate, tasks for which neural language models are well-suited. A tremendous amount of human labor is spent on such chores, and reallocating those resources towards more intellectually stimulating tasks may encourage a larger demographic to become programmers who would otherwise lack the patience or interest. By removing these barriers to entry, programmers can more quickly arrive at the rewarding parts of program design and implementation.

Nevertheless, mere imitation is a somewhat dissatisfying approximation to programming from a computer-scientific perspective – lacking in some essential aspect the qualities its practitioners aspire to fulfill. Helpful though it may be for tedious chores, the art of programming is not reading gigabytes of code and minimizing a cross-entropy loss. Programming requires imagination, creativity, problem-solving – qualities which cannot be conferred by simply scaling existing language models with more data and parameters. What could be missing?

3 Knowledge Discovery and Neural Code Search

For many decades, computer scientists have pondered the nature of search. Search is an indispensable tool in the working programmer's repertoire and goes to the heart of many fundamental problems in artificial intelligence, from classical to statistical optimization, and information retrieval to computational linguistics. Programming itself can be seen as a kind of search-based optimization problem [5], consistent with its original mathematical interpretation, e.g. linear or stochastic programming. Coupled with a grammatical template, one could imagine searching through the space of valid programs to produce a higher-order function satisfying some criteria. Indeed, this exact setup is routinely-studied in the annual syntax-guided synthesis (SyGuS) competition [4].

Returning to our earlier question of, "What else could human programmers be doing besides imitation learning?", one plausible answer could be trial and error. Given some program specification and a computational budget, a naïve strategy could be to simply evaluate as many programs from a dataset of candidate solutions as possible within the allotted budget. Many programmers do in fact practice this style of copy-paste programming as evidenced by duplicate code studies [23], a problem known to adversely bias machine learning models, and which must be corrected for during data curation [1].

For nearly all practical programming languages, the space of valid programs can hardly be enumerated, never mind evaluated in a reasonable amount of time. A more refined strategy is needed: for example, we could select a small set of reusable building blocks, then compose and evaluate partial programs using an execution-guided scheme [12, 33]. By interacting with an interpreter, we may be able to arrive at a solution via incremental improvement. As with most dynamic programming algorithms, the problem comes down to a question of substructure: if modification invalidates prior work, search can become exponential or worse.

For example, many useful programs belong to the class of context free languages. Sampling is possible using a probabilistic grammar, but at what cost? The number of distinct parse trees grows super-exponentially with height. Various strategies have been designed to inhibit this growth, but even with judicious pruning, the topology of many languages does not admit an efficient beam search, or the cost required may be prohibitive. Yet humans are able to solve many computationally hard search problems with deceptive ease. How?

One possibility is that humans possess more computational resources than might we give them credit for, and a similarly-enriched neural-guided search would be equally capable. Another hypothesis is that we are not *searching* for programs as such, or at least thinking about this process in the wrong terms. When someone is painting a portrait or writing a novel, we do not call this search. Likewise, programming is not always about searching for an answer, but finding the right question to ask, of understanding, exploring and visualizing the design space, for which there is no specification. The program acquires a meaning of its own as the medium for a dialog between human and machine. In the following sections, we explore two contrasting models for this dialog, one where the human is the teacher (§4), and one where the IPT is the teacher (§5).

4 Automatic and Declarative Programming

In The Art of Computer Programming [22], Donald Knuth memorably writes, "Programs are meant to be read by humans and only incidentally for computers to execute." Taking this perspective, one may be tempted to ask, "Why must programming languages be so difficult that we need IPTs to write down our ideas in the first place?" If we consider programming a matter of communicating human intent to machines, language designers should take great pains to simplify the language so that users may convey their intent in an effortless manner, then harness machine intelligence in the service of fulfilling their intent, rather than force them to specify how to accomplish it in painstaking detail. Known as declarative programming, this approach has been well-studied in languages like SQL, Prolog and miniKanren, among others.

The idea of what is today known as declarative programming can be traced back to the 1940s when researchers started applying the tools of constraint programming to what is today known as operations research. In this early work, programmers would first state their intent as a minimization problem on a system of inequalities, such as a linear or quadratic program. The Soviets used this technique to study many problems related to optimal transport and economic planning. It also turns out to be useful as a means of defining metrics on graphs and probability distributions.

Our primary interest in declarative programming is twofold: (1) it is the basis of the first successful attempt to popularize gradient-based learning [7], and (2) is a practical framework for realizing the once-scorned [14] but now increasingly plausible [11] idea of natural language programming.

Not only can constraint programming produce numerical solutions, but the same tools can be applied to programming in a more general manner. By considering different algebras, we can compute different properties objects. These things are called algebraic path problems and have many useful applications in combinatorial optimization.

It is possible to implement many fundamental computer science algorithms in a much simpler way as iterated matrix multiplication on a semiring algebra. A commutative monoid $(S, \bullet, ①)$ is a set S with a binary operator $\bullet : S \times S \to S$ which has the following properties:

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c} \qquad \frac{a \bullet ①}{a} \qquad \frac{a \bullet b}{b \bullet a}$$

Associativity *Neutrality* *Commutativity*

A semiring algebra, denoted $(S, \oplus, \otimes, ⓪, ①)$, is a set together with two binary operators $\oplus, \otimes : S \times S \to S$ such that $(S, \oplus, ⓪)$ is a commutative monoid and $(S, \otimes, ①)$ is a monoid. It has the following additional properties:

$$\frac{a \otimes (b \oplus c)}{(a \otimes b) \oplus (a \otimes c)} \qquad \frac{(a \oplus b) \otimes c}{(a \otimes c) \oplus (b \otimes c)} \qquad \frac{a \otimes ⓪}{⓪}$$

Distributivity *Annihilation*

It is possible solve a rich family of search problems using iterated matrix multiplication on semirings. Known as *propagation* or *message passing*, this procedure consists of two steps: *aggregate* and *update*. Let δ_{st} denote some distance metric on a path between two vertices s and t in a graph. To obtain δ_{st}, one may run the following procedure on a desired path algebra until convergence (a few examples are provided for reference):

$$\delta_{st} = \overbrace{\bigoplus_{P \in P^*_{st}} \underbrace{\bigotimes_{e \in P} W_e}_{\text{Aggregate}}}^{\text{Update}}$$

S	$\oplus$	$\otimes$	⓪	①	Path
$\mathbb{R} \cup \{\infty\}$	min	$+$	∞	0	Shortest
$\mathbb{R} \cup \{\infty\}$	max	$+$	$-\infty$	0	Longest
$\mathbb{R} \cup \{\infty\}$	max	min	0	∞	Widest

Many dynamic programming algorithms, including Bellman-Ford, Floyd-Warshall, Dijkstra, belief, constraint, error and expectation propagation, Markov chains, and others can all be neatly expressed as message passing on a semiring algebra. We refer the curious reader to [16, 6] for a more complete summary of the algebraic path problem and its many wonderful applications.

For example, one could use ADTs to represent higher-order functions, i.e. for program synthesis if they so desired.

The problem with automatic programming is that the only means of feedback to the user is by outputting a solution, or failing that, producing lots of heat. It would be nice to have a more interactive mechanism.

5 Computer-Aided Reasoning Tools

The programmer is not the teacher, the programmer is actually the student.

When programmers ask for an intelligent programming tool, what they really want is not a genie who grants wishes without question, but a teacher who rapidly gives them feedback about the implications of their design choices within the confines of a well-defined language. This is one advantage of a type system: not only does it provide highly precise feedback, but also allows us to sample a space of semantically constrained programs. Thus the interaction is bidirectional - the user provides a partial program sketch, and asks the type system to infer the missing thing.

6 Future Directions of Computer Programming

Procedural knowledge is knowledge designed or discovered to facilitate a process. Unlike *relational knowledge* which may be bidirectional, procedural knowledge has a specific direction or goal. The methods presented in this literature review have broad applications to parsing and indexing procedural knowledge. Given a sequence of *paths*, or *traces* representing discrete steps in a goal-directed process with intermediate results, our research seeks to provide a mechanism for

estimating procedural similarity, to synthesize or recommend semantically similar procedures under feasible reorderings. Specifically, our work focuses on the domain of *programming* knowledge.

Code shares many statistical properties in common with natural language and can be studied using natural language processing [18]. Unlike natural language, all syntactically-valid code has an unambiguous grammar. Furthermore, code primarily consists of *functions* or *procedures* intended to operate a machine. The underlying programming model may be relational, object-oriented or functional, but all mainstream programming languages support some form of procedure that accepts inputs and produces outputs.

All procedures in a statically-typed programming language have a *type signature*, for example, `getHomePhone: Person → PhoneNumber`. This represents a *contract* between the procedure and its caller: if the procedure `getHomePhone` is called with a `Person`, it will return a `PhoneNumber`. All other inputs and outputs are forbidden by the type system. While often beneficial to consider more narrow definitions such as soundness or correctness, for procedural similarity these assumptions are unnecessary.

Application programming interfaces (APIs) are a set of interfaces which describe available ways of structuring computation to achieve a set of related programming tasks. We call the graph of all possible ways to compose an API the *API surface*. One traverses the API surface by composing accessible procedures in a *call graph*. Thus, we can view the API as kind of a *procedural knowledge base* representing common data transformations and how to compose them. In practice, how to achieve some desired goal is often far from obvious, requiring a large amount of documentation to explain.

Many consumers of popular APIs publish code and documentation in open source repositories, a largely untapped source of knowledge for programming tools. Our work seeks to find ways of linking knowledge contained in open source repositories to help users locate examples and compose software applications. In the following section, we propose three applications of procedural pattern recognition for documentation alignment (§ 7.1), code search (§ 7.2), program repair (§7.2), and eDSL generation (§ 7.2).

6.1 Documentation alignment

Documentation is an indispensable resource for software developers learning to use a new API or programming language. Maintainers of popular software projects often publish web-based developer documents, typically in markup languages such as HTML or Markdown. These documents contain a mixture of natural language sentences, code snippets, and hyperlinks to related documents and source code files. All of these artifacts hold rich semantic information: the markup graph describes the text in relation to other entities in the document hierarchy, while the link graph describes relationships between relevant documents or artifacts in a software repository.

Consider the typical workflow of a software developer who is seeking information about an unfamiliar API. To effectively locate relevant documentation, she

must first copy a specific fragment of text (e.g. a function name, error message, or identifier) from a development environment into a search engine, providing relevant contextual information. The query must be descriptive enough to retrieve relevant documents with high probability, while omitting extraneous information (e.g. user-defined tokens) unlikely to occur outside the scope of the developer's personal environment or project.

Prior work in information retrieval for software development investigated recommending API documentation [29] and Q&A content [30] to developers. Similar work in natural language processing has studied the relationship between comments and source code entities [19, 26] strictly within source code. Examples of cross-domain entity linking in the source-to-doc (S2D) and doc-to-source (D2S) setting are scarce, however these results indicate alignment between natural language and software artifacts may be feasible.

Our work seeks to facilitate procedural knowledge discovery by enriching lexical queries with semantic information extracted from a programming environment, and prioritizing semantically relevant software artifacts among a set of matching search results. Broadly, the tools we have proposed in this literature review can be used to study both source code and procedural knowledge graphs, however due to the paucity of cross-domain entity links between code and natural language artifacts, reasoning about cross-domain relations will require developing new approaches to feature engineering, unsupervised learning and entity alignment in the low-data regime.

Such an application, if successful, would allow developers to more quickly and easily locate semantically or contextually relevant code samples in API documentation and open source repositories. The same tools could also help authors maintain a consistent set of API documentation and usage examples across a large codebase, a persistent obstacle when evolving any API.

6.2 Code search

Given a learned similarity metric between procedures, one straightforward application is code search. Prior work in this area has explored type-directed [20], learning-based [17] and semantic search [27] techniques. These techniques all use a fixed, or synthetic ordering over search results. For a given hole context, there are often many valid completions within an API or codebase. Given a corpus of procedures in their surrounding typing environment, is it possible to estimate a probability distribution on a shared embedding between contexts and results, and measure the likelihood that a given search result occurs in an empty location? This requires:

1. Efficiently searching a corpus for a well-typed pattern
2. Ranking the matching search results by semantic alignment
3. Incorporating information into user's context (e.g. variable renaming)

Given a cursor and the surrounding context (e.g. in an IDE or editor), such a tool would need to search a database for the most similar contexts, extract common snippets to estimate their *concordance* or *agreement* with the surrounding code context, then adapt the foreign code snippet into to the user's context. External contexts including public code samples or API-documentation (e.g. fixes or repairs for compiler error messages) could help the user to complete some unfinished piece of code. Some open research questions include:

1. **Semantic segmentation**: How do we slice or truncate context?
2. **Graph search**: What kernels enable fast subgraph detection?
3. **Context ranking**: What features best measure contextual similarity?
4. **Refactoring**: How to integrate a selected result into the user's code?

What is a language model? Disregarding meaning, a language model is no more than a statistical model of symbolic information. Natural languages differ from mechanical languages in a few key ways. Natural languages are linear: almost all human languages have a fixed maximum recursion depth governed by biological and cognitive factors. Mechanical languages, by contrast, and are typically nonlinear due to their metalinguistic properties. Thus, natural language models cannot represent a mechanical language without significantly constratining their expressiveness.

One might argue – since natural language models are effectively linear – a language model with a large enough working memory, given enough data, should be able to model language fragments whose description length fits inside their working memory. Indeed, recent literature has shown that natural language models are surprisingly adept at modeling idioms in source code. However, we would expect such models to struggle with fragments whose information

complexity exceeds its working memory (e.g. trees and graphs whose average MDL stretch the limit).

What is the difference between working memory, i.e. reasoning, and long-term memory, i.e. knowledge? Long term memory is information stored in the learned parameters, i.e. the topological structure of the network – as a model *learns*, this information is passively encoded in the topological structure. Long-term mememories must be conserved by the data distribution. Distributional shift tends to erase past memories, a phenonomena known as catastrophic forgetting.

In contrast, short-term memories are dynamical patterns of *activity* which must be actively consereved by the network topology. This second mechanism, called working memory, is a dynamical process with a much smaller information footprint, but allows a network to reason about incoming signals and adapt to previously unseen scenarios without memorization. To be passed on to successive layers, working memories must be actively consereved by the network topology. Both learning and reasoning are a forms of message passing over a graphical structure: learning shapes the graph topology, and reasoning propagates signals through it.

Models which cannot fit a language into working memory must learn maximal-length framgents and use long-term memory to fill in the gaps – i.e. by memorizing transitions between fragments in long-term memory. If this is the case, we would expect to find transitions that are locally consistient, but globally nonsensical. For instance, can source code models learn balanced parenthesis? Balanced intermingled parenethesis? `{([])}` Other algebraic datastructures (e.g. bush, imbalanced trees, etc.)? What are the limit languages that can be learned by natural langauge models of source code (e.g. Transformers)?

Early work in program learning realized the importance of graph-based representations [3], however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal [15], graphical [32] or other user interface to explore the space of valid programs, however do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph [21] from source code, but still require an explicit parser to form the initial graph and adaption to settings where style, terminology and document structure vary remains a challenge.

Imagine a newly-hired developer, who has programming experience, but no prior knowledge about a closed-source project. She receives access to the team's Git repository and is assigned her first ticket: Fix test broken by `0fb98be`. After locating the commit and becoming familiar with the code, she queries StackOverflow, discovers a relevant solution, copies the code into the project, makes a few edits, presses run, and the test passes.

Humans are adept information foraging agents. We can quickly find relevant information in a large corpus by recognizing and following textual landmarks. Software projects are composed of a variety of semi-structured documents containing many clues where relevant information may be found. In this work, we learn a grammar from a model trained on software artifacts like source code and

documentation, in order to facilitate common programming tasks such as code search, completion, or knowledge discovery.

Early work in program learning realized the importance of graph-based representations [3], however explicit graph construction requires extensive feature-engineering. More recent work in program synthesis has explored incorporating a terminal [15], graphical [32] or other user interface to explore the space of valid programs, however do not consider the scope or variety of artifacts in a software project. Others have shown the feasibility of learning a local graph [21] from source code, but still require an explicit parser to form the initial graph and adaption to settings where style, terminology and document structure vary remains a challenge.

Prior work in the code search literature explores the text-to-code [**?**] setting, where queries are typically considered to be a short sequence composed by the user, or code-to-code [**?**] setting where the query takes the form of a code snippet. Model performance typically evaluated using mean reciprocal rank (MRR), mean average precision (MAP), normalized discounted cumulative gain (NDCG), precision and recall at top N-results (P/R@N), or similar metrics. Although some [**?**] do incorporate other features from the local document, none however, consider the query in the context of a broader project. The ability to align contextual features from the surrounding project, we argue, is essential to delivering semantically relevant search results. Pior studies of natural langauge in source code have been undertaken [**?**, **?**, 11], which characterize the families of computational languages that neural langauge models can recognize in practice.

The proposed method presents an interesting engineering challenge. To work for code completion, it would need to have relatively low latency for a pleasant user experience. While the model could be trained on a large corpus offline, the contextual embedding would need to be periodically recomputed as the surrounding typing environment changes in the editor. Furthermore, the search and retrieval speed would need keystroke latency to be effective.

6.3 DSL generation

Most knowledge starts with pen on paper. It is generally possible for developers to translate well-structured pseudocode into code. However, there are many valid translations – the same algorithm implemented in the same language by different authors is seldom written the same way. Instead of translating these ideas from scratch, it may be possible to encode just a few axioms and enough knowledge to derive a family of algorithms, reusing existing optimizations written by other developers and selecting the most appropriate procedure for computing a desired quantity, e.g. using a sufficiently expressive logic system, and a database of simple facts and relations.

Knowledge systems or *ontologies* are a collection of related facts designed or discovered by human beings. For example, we can treat mathematical knowledge as a kind of symbolic rewrite system. This has been successfully operationalized in Theano [8] and other DSLs. It is possible to view constructive mathematics libraries like Metamath [24], Rubi [28], Arend, KMath [25] and Kotlin∇ [13] as

also working towards this high-level goal.

In knowledge engineering, approximate equality is known as entity *alignment* or *matching*. With an approximate matching algorithm, we could accurately detect near-duplicates in a large codebase, find similar documentation snippets, retrieve contextually or semantically relevant code samples to assist developers writing unfamiliar code, and search for bugs in code or fixes from an external knowledge base to repair them. All of these tasks require some notion of semantic similarity between procedures.

7 Conclusion

8 Acknowledgements

The author wishes to thank his advisors Jin Guo and Xujie Si, for providing several helpful comments on this proposal, Disha Shrivastava and David Yu-Tung Hui, for various interesting discussions.

References

[1] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

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

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[5] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

[6] John S Baras and George Theodorakopoulos. Path problems in networks. *Synthesis Lectures on Communication Networks*, 3(1):1–77, 2010.

[7] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[8] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pages 1–7. Austin, TX, 2010.

[9] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

[10] M Bras and Y Toussaint. Artificial intelligence tools for software engineering: Processing natural language requirements. *WIT Transactions on Information and Communication Technologies*, 2, 1993.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

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

[13] Breandan Considine, Michalis Famelis, and Liam Paull. Kotlin∇: A Shape-Safe eDSL for Differentiable Programming. `https://github.com/breandan/kotlingrad`, 2019.

[14] Edsger W Dijkstra. On the foolishness of natural language programming. In *Program construction*, pages 51–53. Springer, 1979.

[15] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. *arXiv preprint arXiv:1906.04604*, 2019.

[16] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[17] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[18] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[19] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[20] Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[21] Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

[22] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[23] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[24] Norman Megill. Metamath. In *The Seventeen Provers of the World*, pages 88–95. Springer, 2006.

[25] Alexander Nozik. Kotlin language for science and kmath library. In *AIP Conference Proceedings*, volume 2163, page 040004. AIP Publishing LLC, 2019.

[26] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. Associating natural language comment and source code entities. In *AAAI*, 2020.

[27] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *PLDI*, pages 1066–1082, 2020.

[28] Albert D Rich and David J Jeffrey. A knowledge repository for indefinite integration based on transformation rules. In *International Conference on Intelligent Computer Mathematics*, pages 480–485. Springer, 2009.

[29] Martin P Robillard and Yam B Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[30] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 392–403, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[32] Homer Walke, R Kenny Jones, and Daniel Ritchie. Learning to infer shape programs using latent execution self training. *arXiv preprint arXiv:2011.13045*, 2020.

[33] Chenglong Wang, Po-Sen Huang, Alex Polozov, Marc Brockschmidt, and Rishabh Singh. Execution-guided neural program decoding. In *ICML workshop on Neural Abstract Machines and Program Induction v2 (NAMPI)*, 2018.