2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 978-1-6654-9596-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-NIER55298.2022.9793523

Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers

Michael Schröder
TU Wien
Vienna, Austria
michael.schroeder@tuwien.ac.at

Jürgen Cito
TU Wien and Meta Platforms, Inc.
Vienna, Austria
juergen.cito@tuwien.ac.at

ABSTRACT

Ad hoc parsers are everywhere: they appear any time a string is split, looped over, interpreted, transformed, or otherwise processed. Every ad hoc parser gives rise to a language: the possibly infinite set of input strings that the program accepts without going wrong. Any language can be described by a formal grammar: a finite set of rules that can generate all strings of that language. But programmers do not write grammars for ad hoc parsers—even though they would be eminently useful. Grammars can serve as documentation, aid program comprehension, generate test inputs, and allow reasoning about language-theoretic security. We propose an automatic grammar inference system for ad hoc parsers that would enable all of these use cases, in addition to opening up new possibilities in mining software repositories and bi-directional parser synthesis.

ACM Reference Format:
Michael Schröder and Jürgen Cito. 2022. Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers. In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510455.3512787

1 INTRODUCTION

Parsing is one of the fundamental activities in software engineering. Following Grune and Jacobs [22], we take parsing to mean "the process of structuring a linear representation in accordance with a given grammar," an activity so common that pretty much every program performs some kind of parsing at one point or another. Academically, parsing has been studied since the very early days of computer science [27] and *formal language theory*, which has its origin in linguistics [8], provides the foundation for an impressive amount of both theoretical results [25] and practical applications [22]. As part of every-day programming, *regular expressions* [53] are probably the biggest and most widely known success story of applied formal language theory. But apart from regexes, only a small minority of programs, mainly compilers and some protocol implementations, make explicit reference to the formal-theoretic underpinnings of parsing, documenting grammars of their input languages and making use of formalized parsing techniques such as parser generators [30, 45] or parser combinator frameworks [34]. The vast majority of parsing code in software today is *ad hoc*.

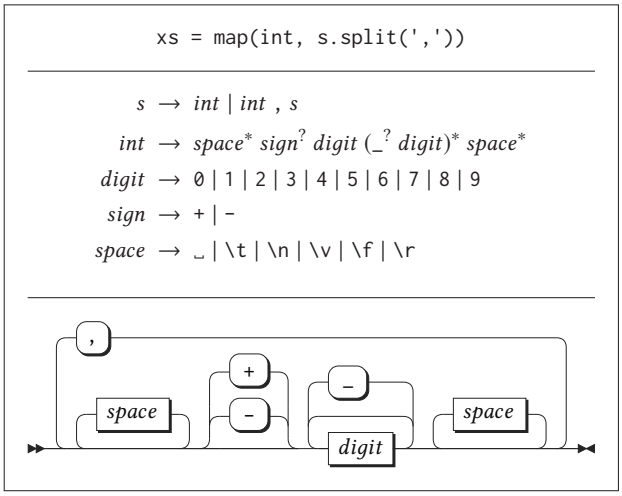

Figure 1: An ad hoc parser and its grammar.[1]

The Python expression in Figure 1 is a typical example of an *ad hoc parser*. It transforms a string `s` into a list of integers `xs`. First, the `split` function breaks `s` into its comma-separated substrings, then the `map` function applies the `int` constructor to all substrings, turning each into a proper integer value. This parser does not use any particular parsing techniques or frameworks, just ordinary functions manipulating strings and transforming values. A programmer writing this expression would most likely not think about the fact that they are writing a parser. Splitting a comma-separated list of values, just like extracting a command-line argument, reading a timestamp, or any other minor programming task involving strings, barely registers as parsing. Commonly, this kind of parsing code is deeply entangled with application logic—a phenomenon known as *shotgun parsing* [42].

Figure 1 also includes a complete grammar for this parser (assuming the semantics of Python 3.9). It is not a particularly complex grammar, but it is perhaps still surprising. Even an experienced Python programmer might be unaware, for example, that the `int` constructor, in addition to allowing an optional leading `+` or `-` sign, also permits leading zeroes, strips surrounding whitespace, and ignores single `_` characters that are used for grouping digits. Looking at the grammar, we can see that the strings `"12,304"` and `"+01_2,3_0_4␣"` will both be accepted by the parser, while the

[1] The notation used here denotes `terminals` with typewriter font and uses the common operators *, +, and ? for zero-or-more, one-or-more, and optional occurrences, respectively. The vertical bar | separates alternative productions, without precedence. Parentheses are used for grouping in the usual way.

ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9224-2/22/05…$15.00
https://doi.org/10.1145/3510455.3512787

empty input "" will crash the program. That the parser's language excludes the empty string is obvious from the grammar, but might be difficult to work out from looking at the code alone.[2]

A grammar certainly reveals a great deal about a rather deceptively simple looking expression, yet no programmer would actually write it down. Grammars share the same fate as most other forms of specification: they are hard to write, can be hard to read, and seem hardly worth the trouble—especially for ad hoc parsers. *If we are not building whole houses, why should we draw blueprints?* [33]

But there is a form of specification, one wildly more successful than grammars, that we can draw inspiration from: *types*. Formal grammars are similar to types, in that a parser without a grammar is very much like a function without a type signature. Types have one significant advantage over grammars, however: most type systems offer a form of *type inference*, allowing programmers to omit type annotations because they can be automatically recovered from the surrounding context.[3] If we could infer grammars like we can infer types, we could reap all the rewards of having a complete specification of our program's input language, without burdening the programmer with the full weight of formal language theory.

In this work, we sketch a possible path towards inferring grammars for ad hoc parsers by combining methods found in refinement types and string constraint solving. We describe future possibilities where grammar inference enables, among other things, better program comprehension by explicitly documenting a program's input space, and bi-directional parser synthesis that helps developers refine and secure their input validation.

2 THE NEED FOR GRAMMARS

Before we delineate how to statically infer grammars, we want to briefly motivate why every (ad hoc) parser would greatly benefit from having a known grammar.

Documentation. A formal grammar is the ideal documentation for a parser, because it provides a high-level perspective that focuses on the *data* as opposed to the code. It allows the programmer to grasp the input language as is, without being distracted by the mechanics of the implementation. There exist numerous notations for grammars, each suitable for different languages and in different contexts: regular expressions [53], Chomsky normal form [9], Augmented Backus-Naur Form (ABNF) [12], parsing expression grammars (PEGs) [16], etc. Graphic representations, like finite state machines [25] or railroad diagrams [7] (see Figure 1), can be particularly helpful in understanding abstract data and align with developers' appreciation of sketches and diagrams [5].

Program Comprehension. It is known that providing alternative representations for a programming task can increase program comprehension [15, 18]. The example in Figure 1 demonstrates how a grammar can elucidate the corresponding ad hoc parsing code, revealing otherwise hidden features and potentially bugs or security issues. Because a grammar is also a *generating device*, it is possible to construct any sentence of its language in a finite number of steps—manually or in an automated fashion. Generating concrete examples of possible inputs further helps in understanding parsing code, and can be invaluable during testing and debugging.

Fuzzing. We can test programs by bombarding them with (systematically generated) random inputs and seeing if anything breaks. This is known as fuzz testing, or fuzzing [39, 56]. Generating good fuzz inputs is not easy, because in order to penetrate into deep program states, one generally needs valid or near-valid inputs, meaning inputs that pass at least the various syntactic checks and transformations—i.e. ad hoc parsers—scattered throughout a typical program. One promising approach is *grammar-based fuzzing* [4, 24], where valid inputs are specified with the help of language grammars.

Language-Theoretic Reasoning. As formal descriptions of input languages, grammars allow us to reason about various language-theoretic properties, such as computability bounds. The *language-theoretic security* (LANGSEC) community[4] regards such reasoning as vital in assuring the correctness and safety of input handling routines. For example, if an input language is recursively enumerable, we can never guarantee that its parser behaves safely (i.e. halts) on inputs that are not in the language, because the parser must be equivalent to a Turing machine. Thus, input languages should be minimally powerful, and their parsers should match them in computational power [49]. Ad hoc parsers open themselves up to attack, because it is not clear what languages they implement, or if they implement them correctly, and variations among implementations are easily overlooked [50]. Grammars can help assure us that our input languages have favorable properties and that their parsers are implemented correctly.

Automatic Parser Generation. A *parser generator* is a tool that synthesizes a parser from a given grammar. Examples include Yacc [30], ANTLR [45], and OMeta [54]. These tools are common in certain areas, such as compilers, and are usually invoked during program build time, generating parsing code that is linked with the rest of the program. The great advantage of starting with a grammar and letting the parser implementation be generated automatically is a high assurance of correctness, as well as easier maintainability.

3 TOWARD GRAMMAR INFERENCE

We hope to realize automatic grammar inference based on the following intuition: Any parser is essentially a *machine* in the formal sense—it is a recognizer for its input language.

3.1 Background: Languages & Machines

Formally, a *language* L is a possibly infinite set of sentences over a finite alphabet Σ. We can define languages very abstractly, as in $L = \{a^n b^n \mid n > 0\}$, a language over the alphabet $\Sigma = \{a, b\}$ that consists of all sentences with at least one a followed by the same number of bs. Usually, however, we define languages via generative devices called *grammars* or recognizing devices called *machines*.

A grammar $G = (V, \Sigma, P, S)$ is a finite description of a language and consists of a set of *variables* (or *nonterminals*) V; a *terminal* alphabet Σ; a set of *productions* P, which are rules of the form $\alpha \to \beta$ where α and β are from V and/or Σ; and a *start symbol* $S \in V$. By

[2] The `split` function, when applied to an empty string, returns a singleton list also containing an empty string (rather than an empty list, as one might assume). The `int` constructor, applied to this empty string via `map`, will then throw a runtime exception.
[3] For a good introduction to type inference and its history, see [37, § 4].
[4] https://langsec.org

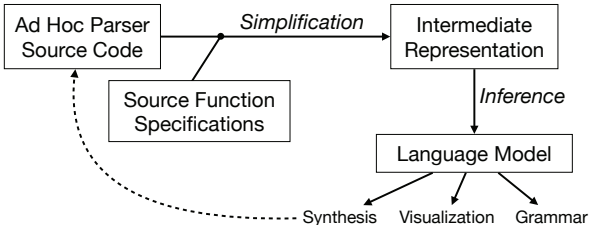

Figure 2: Sketch of our grammar inference system.

starting with S and applying a finite number of productions from P, we can generate sentences over Σ. The language $L(G)$ is the set of all sentences that can be generated by G. By putting various constraints on the *form* of a grammar, such as whether the left-hand side of a production can only include variables, or the right-hand side has to include at least one terminal symbol, and so on, we can limit the grammar's expressiveness, constraining the *family* of languages a grammar of this form can produce. The famous Chomsky hierarchy [9] partitions languages/grammars into four increasingly expressive levels: *regular*, *context-free*, *context-sensitive*, and *recursively enumerable*. Numerous additional language families and types of grammars have been discovered, within and beyond the classic hierarchy: *attribute grammars* [32], *boolean grammars* [43, 44], the *mildly context-sensitive* and *sub-regular* languages [28], *parsing expression grammars* (PEGs) [16], to name just a few.

A machine M, unlike a grammar, does not produce sentences but consumes them. Taking some sentence as input and moving through a finite number of internal states, it arrives at some halting configuration if and only if the sentence is part of the language $L(M)$. If the sentence is not part of the language, the machine either runs forever or gets stuck in a non-accepting state. Just like with grammars, the way that a machine is constructed determines its expressiveness. There is a natural correspondence between languages, grammars, and machines: regular languages correspond to *finite state machines*, which simply move from one internal state to another based on the next input character; context-free languages correspond to finite state machines equipped with a pushdown stack, also known as *pushdown automata*; context-sensitive languages correspond to *linearly bounded automata*, in essence Turing machines with a finite tape; and finally the recursively enumerable languages correspond to the well-known unbounded *Turing machines*. As with grammars, there are numerous additional and alternative constructions between and beyond these classic ones.

3.2 Intuition: Parsers are Embedded Machines

A parser, like the Python snippet in Figure 1, which expects a string from some language L_1 as input, can be seen as a machine M_1 recognizing that language, so that $L_1 = L(M_1)$. This machine is however embedded within the more powerful machine M_0, the general-purpose programming language that the parser itself is written in. Any real world parser will do more than just recognize a language: it will allocate and transform data types, throw exceptions or handle parse errors, or perform side effects unrelated to the parsing process itself. Nevertheless, the control flow at the core of a parser will, in our experience, closely match that of the (hypothetical) machine M_1. While it is entirely possible that a parser written in a Turing-complete programming language exhibits exactly those traits that make it equivalent to a Turing machine, even though it might be parsing a "lesser" language, we think this to be very unlikely. In almost all practical situations, ad hoc parsing code will not significantly exceed the "power-level" of the language it is parsing. For example, unless it has been especially constructed to be confounding, the loops present in a parser will invariably be bounded by at most some linear factor of the length of its input, which corresponds to the expressiveness of a context-sensitive language. Thus, we think it is feasible to transform ad hoc parsers into equivalent machines whose languages can be statically inferred.

3.3 Vision: Automatic Grammar Inference

Figure 2 shows a sketch of the end-to-end grammar inference system that we envision. In the first step, ad hoc parsing code is transformed from a Turing-complete source language (e.g. Python) into an intermediate representation (IR) that is essentially a domain-specific language for parsing. This transformation can be seen as a simplification: it removes syntactic sugar, makes control flow explicit, and throws away all parts of the source code that are not related to parsing. During this step, known string processing functions are translated into one or more equivalent functions of the IR that precisely model the semantics of the source. To illustrate, let us consider a slightly extended version of the example from Figure 1:

```
1 def vector_length(s):
2   [x,y,z] = map(int, s.split(','))
3   return math.sqrt(x**2 + y**2 + z**3)
```

The simplification results in roughly the following IR:

```
1 let parse = λ(s : String {⋆}).
2   let v1 = split_py "," s in
3   let xs = map int_py v1 in
4   let v2 = length xs in
5   let v3 = equals 3 v2 in
6   assert v3
```

Note that this function does not actually return anything. The goal here is not to run it and obtain a result, but to fill the hole ($\star$) in its input type by inferring the appropriate string constraints. To this end, the functions $\texttt{split}_{\texttt{py}}$ and $\texttt{int}_{\texttt{py}}$ precisely model their Python counterparts and refine their input and output types by imposing the constraints resulting from their modeled string processing behavior. Note also how a remnant of the pattern match `[x,y,z]` from the source is present in form of an (indirect) constraint on the length of the string (lines 4–6 in the IR).

Inferring the type of `parse` and solving its string constraints results in a model of the original ad hoc parser's input language. To make the resulting grammar traceable to the originating code, the model also contains rich source location information, which has to be threaded through both the simplification and inference steps. In a final step, the language model can then be used to generate the desired textual, visual, and interactive grammar representations.

4 RELATED WORK

Grammatical Inference. A related but different problem to our goal of finding a grammar given a parser is to find a grammar given

a set of sentences. This is known as *grammatical inference* or *grammar induction*. Early results in computational linguistics quickly established fundamental limits of what could be achieved: it was shown that not even regular languages can be identified given only positive examples [20]. Nevertheless, with applications ranging from speech recognition to computational biology, grammatical inference is an active and vibrant field [13, 14].

Fuzzing. A big problem in grammar-based fuzzing (cf. § 2) is obtaining accurate grammars or language models. Black-box approaches try to infer a language model by poking the program with seed inputs and monitoring its runtime behavior [6, 19]. This has some theoretical limits [2, 3] and the amount of necessary poking (i.e. membership queries) grows exponentially with the size of the grammar. White-box approaches make use of the program code and can thus use more sophisticated analysis techniques, e.g. taint tracking to monitor data flow between variables [26] or tracking dynamic control flow and observing character accesses of input strings [21]. These approaches rely on dynamic execution, but can produce fairly accurate and human-readable grammars, at least in test settings. They can not, however, provide any guarantees of correctness, and thus it is not possible to determine how accurate the resulting grammars really are. In our vision, grammars are *statically* inferred from source code and are always *sound*. By not relying on dynamic execution of whole programs, grammars can be extracted from individual functions or even partial programs, and it is not necessary to generate seed inputs to bootstrap inference.

String Constraint Solving. String constraints are relations defined over string variables and arise out of program statements that manipulate strings, e.g. concatenation or substring replacement. Reasoning about strings requires solving combinatorial problems involving such constraints, which is difficult to do both efficiently and completely, and a large number of approaches have been developed [1, 52]. Our problem of grammar inference is in some ways the inverse: instead of wanting to model all possible strings a function can return or express, we want to model all possible strings a function can accept (without throwing an error or getting stuck).

5 NEW POSSIBILITIES

The end-to-end grammar inference system we envision (§ 3) will not only let us enjoy all the benefits that formal grammars provide in general (§ 2), it also enables some exciting new possibilities.

Interactive Documentation. A grammar that is automatically inferred will always be up-to-date—a significant advantage over manually written documentation, which tends to quickly drift from the object it documents [35]. Furthermore, an inferred grammar could be closely linked directly to the underlying source code, making productions traceable to their origins. One can imagine an interactive environment where hovering over parts of a grammar highlights the corresponding pieces of code—or even allows changing them by manipulating the high-level representation.

Bi-directional Parser Synthesis. Combining grammar inference with parser generation enables a framework of bi-directional parser synthesis. In the most basic case, starting from an existing complete parser implementation, the synthesizer can be used to generate different implementations according to certain criteria, e.g. performance or code style, by transformation via the inferred grammar—a specialized type of semantic program transformation [11]. If the initial parser is incomplete, a bi-directional parser synthesizer can be used for *program sketching* [36, 46, 51], wherein an initial implementation (a "sketch") is the basis of an initial grammar which can be manipulated by the user on a high level—perhaps graphically—to then in turn synthesize a completed or refined implementation. If the sketch-synth loop can be sufficiently shortened, it can be the basis for a direct manipulation bi-directional programming system [10, 40], although based on transformations of the (specification of) inputs to the program rather than its outputs.

Mining & Learning. An inferred grammar abstracts over the underlying concrete parser implementation and can be viewed as an equivalence class, allowing us to group together different parser implementations with similar semantics.[5] This opens up new possibilities in mining software repositories, such as grammar-enhanced semantic code search [17, 41, 47] or detecting code clones [31, 55] of ad hoc parsers. By automatically inferring grammars for each code change, it also becomes possible to learn how (implicit) input specifications evolve over time, enabling a type of grammar-aware semantic change tracking [23, 48]. Augmenting the code review process with current as well as historical grammar information would allow developers to be alerted when a code change introduces a perhaps unexpected change in input grammar.

6 FUTURE PLANS

We want to build the grammar inference system described in this paper and apply it in real-world situations. We plan to realize our vision in a series of upcoming works:

- We are currently conducting a mining study of ad hoc parsers in the wild, collecting common coding patterns in order to determine the possible scope of our system.
- We are currently investigating the use of refinement types [29] in combination with string constraint solving to realize inference of a language model from a simplified parsing IR. While we have seen initial success with smaller examples, we need to expand to more kinds of parsers to understand our scope and limitations.
- To ensure the validity of our approach, both simplification and inference need to be proven sound. We plan on supplying machine-checked proofs for both of these steps.
- To ensure the effectiveness of our approach, we plan on evaluating the system on a corpus of curated ad hoc parser samples from the real world. We have begun collection of a suitable dataset.
- We plan on conducting a large-scale mining study of inferred grammars, to demonstrate the usefulness of our system to applications of code mining and learning.
- We plan on conducting a number of user studies on grammar comprehension in order to determine the benefits and drawbacks of different textual and visual grammar representations.

We are excited about the prospects of automated grammar inference and invite the community to collaborate with us to realize our vision of "grammars for free".

[5] While there are a number of theoretical bounds regarding the decidability of properties about grammars, it is in fact possible to efficiently decide equivalence for many types of grammars encountered in practice [38].

REFERENCES

[1] Roberto Amadini. 2021. A Survey on String Constraint Solving. arXiv:2002.02376 [cs.AI]

[2] Dana Angluin. 1987. Queries and Concept Learning. *Mach. Learn.* 2, 4 (1987), 319–342.

[3] Dana Angluin and Michael Kharitonov. 1995. When Won't Membership Queries Help? *J. Comput. System Sci.* 50, 2 (1995), 336–355.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.*

[5] Sebastian Baltes and Stephan Diehl. 2014. Sketches and Diagrams in Practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. 530–541.

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. *SIGPLAN Not.* 52, 6 (June 2017), 95–110.

[7] Lisa M Braz. 1990. Visual syntax diagrams for programming language statements. *ACM SIGDOC Asterisk Journal of Computer Documentation* 14, 4 (1990), 23–27.

[8] Noam Chomsky. 1957. *Syntactic Structures.* Mouton & Co. 117 pages.

[9] Noam Chomsky. 1959. On Certain Formal Properties of Grammars. *Information and Control* 2, 2 (1959), 137–167.

[10] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. 341–354.

[11] Patrick Cousot and Radhia Cousot. 2002. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. 178–190.

[12] D. Crocker and P. Overell. 2008. *Augmented BNF for Syntax Specifications: ABNF.* STD 68. RFC Editor.

[13] Colin de la Higuera. 2005. A bibliographical study of grammatical inference. *Pattern Recognition* 38, 9 (2005), 1332–1348. Grammatical Inference.

[14] Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press.

[15] M Fitter and TRG Green. 1979. When do diagrams make good computer languages? *International Journal of man-machine studies* 11, 2 (1979), 235–261.

[16] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. 111–122.

[17] Isabel García-Contreras, José F. Morales, and Manuel V. Hermenegildo. 2016. Semantic code browsing. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 721–737.

[18] David J. Gilmore and Thomas R. G. Green. 1984. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* 21, 1 (1984), 31–48.

[19] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn & Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 50–59.

[20] E Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474.

[21] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. 172–183.

[22] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing Techniques* (2nd ed.). Springer, New York, NY.

[23] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding Code Change Understanding with Semantic Change Impact Analysis. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 202–212.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

[25] John Hopcroft and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

[26] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. 720–725.

[27] Edgar T. Irons. 1983. A Syntax Directed Compiler for ALGOL 60. *Commun. ACM* 26, 1 (Jan. 1983), 14–16.

[28] Gerhard Jäger and James Rogers. 2012. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences* 367, 1598 (2012), 1956–1970.

[29] Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. (2020). arXiv:2010.07763 [cs.PL]

[30] Stephen C Johnson and Ravi Sethi. 1990. Yacc: A Parser Generator. *UNIX Vol. II: Research System* (1990), 347–374.

[31] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. 485–495.

[32] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[33] Leslie Lamport. 2015. Who builds a house without drawing blueprints? *Commun. ACM* 58, 4 (2015), 38–41.

[34] Daan Leijen and Erik Meijer. 2001. Parsec: Direct style monadic parser combinators for the real world. (2001).

[35] T.C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (2003), 35–39.

[36] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (Aug. 2020), 29 pages.

[37] David MacQueen, Robert Harper, and John Reppy. 2020. The History of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (June 2020), 100 pages.

[38] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating Grammar Comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. 183–200.

[39] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. arXiv:1812.00140 [cs.CR]

[40] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages.

[41] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-Based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. 997–1016.

[42] Falcon Darkstar Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 45–52.

[43] Alexander Okhotin. 2004. Boolean grammars. *Information and Computation* 194, 1 (2004), 19–48.

[44] Alexander Okhotin. 2013. Conjunctive and Boolean grammars: the true general case of the context-free grammars. *Computer Science Review* 9 (2013), 27–59.

[45] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[46] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. 522–538.

[47] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. 1066–1082.

[48] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. 2004. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 188–197.

[49] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. 2013. Security Applications of Formal Language Theory. *IEEE Systems Journal* 7, 3 (2013), 489–500.

[50] Joern Schneeweisz. 2020. *How to exploit parser differentials.* Retrieved July 16, 2021 from https://about.gitlab.com/blog/2020/03/30/how-to-exploit-parser-differentials/

[51] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. UC Berkeley.

[52] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. 620–635.

[53] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[54] Alessandro Warth and Ian Piumarta. 2007. OMeta: An Object-Oriented Language for Pattern Matching. In *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada) *(DLS '07)*. 11–19.

[55] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural Detection of Semantic Code Clones via Tree-Based Convolution. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 70–80.

[56] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book.* CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-03-12 11:41:11+01:00.