The Linguistics of Programming

Colin S. Gordon
Department of Computer Science
Drexel University
Philadelphia, USA
csgordon@drexel.edu

Abstract

Research in programming languages and software engineering are broadly concerned with the study of aspects of computer programs: their syntactic structure, the relationship between form and meaning (semantics), empirical properties of how they are constructed and deployed, and more. We could equally well apply this description to the range of ways in which linguistics studies the form, meaning, and use of natural language. We argue that despite some notable examples of PL and SE research drawing on ideas from natural language processing, there are still a wealth of concepts, techniques, and conceptual framings originating in linguistics which would be of use to PL and SE research. Moreover we show that beyond mere parallels, there are cases where linguistics research has complementary methodologies, may help explain or predict study outcomes, or offer new perspectives on established research areas in PL and SE. Broadly, we argue that researchers across PL and SE are investigating close cousins of problems actively studied for years by linguists, and familiarity with linguistics research seems likely to bear fruit for many PL and SE researchers.

Keywords: programming languages, software engineering, linguistics

ACM Reference Format:
Colin S. Gordon. 2024. The Linguistics of Programming. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3689492.3689806

1 Introduction

Programming language and software engineering research are centered around computer programs: their construction process (both social and technical), how they are expressed, what guarantees we can make of them, how they interact with additional artifacts (such as comments, design documents, and natural language specifications), how people comprehend them, and what common properties the languages used to build them either have or could have (by design) and how those interact with what a program expresses (a computational solution).

Linguistics is concerned with analogous ideas for human languages, rather than programming: what are commonalities across languages, how do those differences interact with language use, what bounds can be established on how languages might evolve to express (or leave implicit) certain information, how language use might vary according to use case or social context, how to systematically and precisely model natural languages, and more.

While the two fields have clear differences — in particular, studying engineered languages versus naturally evolving languages — linguistics offers a wealth of ideas, perspectives, and established analytical tools (both formal and informal) which we believe can help shed light on problems of interest to researchers in programming languages and software engineering.

This essay seeks to give a broad overview of pieces of linguistics with clear relevance to problems actively studied in computer science. Out of neccessity this means our discussions will only scratch the surface of the field and cannot include all possible points of connection; our priority is to demonstrate a number of places where connections may occur, and provide entry points to the literature for those interested in problems like those we highlight, or those who see relevance between the linguistic concepts we point out and other problems.

2 Parallels Between Linguistics and PL/SE

Linguistics is an enormous area, neither strictly science nor strictly humanity, but, as the meme goes, a secret third thing (i.e., both). It generally covers such topics as syntax (roughly, grammar); semantics (roughly, literal meaning); pragmatics (how and why language is used certain ways rather than others in a given situation, and how words that literally mean one thing can acquire additional meanings in context); discourse analysis (study of ongoing linguistic interactions); morphology (the relationship between word forms and meanings, such as the relation between word form changes and meaning change); orthography (the study of writing systems); phonetics and phonology (the study of how speech

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1215-9/24/10.
https://doi.org/10.1145/3689492.3689806

sounds are generated and sensed, and how they change situationally or over time); psycholinguistics (how people acquire and use language); and neurolinguistics (how language is represented and processed in the human brain); and interfaces between these areas. Virtually every area of linguistics has at least a rough parallel in existing PL and SE literature.

We are not the only ones to notice parallels between programming languages and linguistics. Adamczyk [2] explores a bit of the history of the word "language" applied to computer programs, and engages in some philosophical musing and thought experiments based on some of Stephen Pinker's ideas about language.[1] Recently Noble and Biddle [130] published an extended argument that programming languages are, in a meaningful sense, languages in the sense of those studied by linguists, in that they are critically linguistic communications between various parties (from programmers to other programmers or to machines, depending on context), regardless of the differences that obviously exist. Their argument is philosophical, calling out numerous similarities between programming languages and human languages. However, they leave the question of what to do with these similarities up to the reader. This essay's life began before the publication of that paper, but could be seen as a kind of follow-up providing more fine-grained connections: our goal is to point out concrete ideas and techniques from linguistics that have potential value to the study and creation of programming languages and the processes of software engineering.

Distinguishing NLP from Linguistics. In this paper we seek to make a distinction between natural language processing (NLP) on one hand, and linguistics (including computational linguistics) on the other. To be sure, there is no hard and fast dividing line between the two and they share a wealth of background knowledge and techniques, but there are marked differences in the primary goals of each area. In the sense we use the terms, NLP is focused quite literally on processing natural language text for end use cases such as extracting information, question answering, statistical language modeling for sequence prediction, etc. Linguistics is interested in understanding "how language works" (and why) in essentially all contexts. Beyond the task-driven vs. curiosity-driven framings (which are not mutually exclusive!), these differences in goals result in different value systems for which questions to investigate and which techniques are appropriate. At times these value systems agree, and other times they do not. These differences primarily manifest in NLP work frequently discounting fundamental assumptions or established results from linguistics, in almost any case where outcomes on a particular metric appear to be better using a technique that doesn't account for them. For example, most uses of large language models (LLMs) for chat or text processing do not attempt to explicitly model the meaning of language; these LLMs underlie the best-known results on a wide array of tasks, in both NLP and software engineering, but fail to capture basic concepts like simple boolean reasoning [50, 134, 179], do not demonstrate certain kinds of linguistic reasoning essential to human language use [147], and are provably unable to learn essential concepts like universal quantification [7]. For many applications, this may not matter, so NLP continues using these tools widely.[2] Conversely, while (computational) linguistics is not agnostic with regards to computational efficiency and scaling laws,[3] explanatory power is generally prioritized over computational convenience (many grammar formalisms are Turing-complete [26, 27, 139]), and often more attention is paid to the construct validity of measurements, rather than over-using simple task metrics like BLEU scores [135].[4] In this essay, we will emphasize drawing upon work focused primarily on how and why language works.

On Large Language Models. A natural question for this essay in 2024 might be: how much will we discuss the *large language models* (LLMs) underlying systems like ChatGPT and a plethora of other tools? The answer is, almost none. Our goal is to encourage researchers in programming languages and software engineering to study linguistics broadly, to bring a wealth of new relevant ideas and tools to bear on our work. We don't need to say anything about LLMs for that to happen — it has already happened! In general, PL and SE researchers have a long history of adopting the latest statistical and probabilistic models where useful and appropriate. For similar reasons, we say little about neurolinguistics, as this connection is already starting to be explored [138, 167, 168].

LLMs are only a small slice of linguistics, whose import in linguistics writ large seems distorted to those of us in computer science, as we are closest to the computational linguists, whose work in the past 15 years *has* been largely overwhelmed by the influence of neural language models.

[1] He notes that those ideas are not settled interpretations in linguistics, citing Tomasello [178]; however, this is an understatement: the consensus across linguistics has long been that most of Pinker's ideas about language in that book are incorrect, and largely based on selectively ignoring contrary evidence [78, 116, 152, 178] [79, Ch. 10].

[2] This is not to suggest linguistics is completely disinterested in LLMs; on the contrary they are being investigated with care as approximate models of *some specific aspects* of human language use, such as surprisal [65] (relevant to ways in which using non-standard language affects how long it takes humans to comprehend text), and even language acquisition [38, 185] (though this is fraught with significant methodological pitfalls [102]).

[3] For example, one justification for studying mildly-context-sensitive grammar formalisms [93, 94, 106] is that they can be parsed in polynomial time, which seems like a reasonable rough upper bound on what humans can process online during linguistic interactions [164], compared to simply requiring parsing to be decidable.

[4] BLEU was a heuristic scoring mechanism proposed for evaluating accuracy of machine translation, premised among other things on input and output having similar length, making it inappropriate for use in tasks that involve summarization as well — including scoring of comment generation [44, 165, 169] — in addition to concerns about fitness for its original purpose [24].

However, linguistics is far, *far* broader than that. Any impression that all of linguistics has been overtaken by LLMs is similar to a layperson who believes that developers have basically stopped programming and are simply using ChatGPT to generate most of their code — understandable based on their most common source of information (for us, NLP researchers in our departments, for them, popular press and marketing material), but based on skewed and highly incomplete sources of information. Our emphasis in this essay is to argue for the rich body of *explanatory* models from linguistics, which are largely orthogonal to the LLMs already widely adopted in our fields.

A Plan for Discussion. The rest of this essay is structured as a tour of just a few select areas of parallels between linguistics and PL and SE research. We will discuss examples that deal with applying linguistics knowledge to both humans and to text. That text may be either natural language text *about* programs, or program text *itself* (source code). We will work from higher-level concerns about human factors and what research hypotheses we consider, towards lower-level concerns of methodological questions and technical devices suitable for incorporation into tools. It is our hope that the breadth of discussion, from the human to socio-technical to purely technical, will demonstrate that regardless of which slices of programming language and software engineering research we engage with, there is a wealth of established linguistics knowledge just waiting to be applied.

3 Human Factors in Language and Tool Design

Software engineering has long made significant use of ideas connected to HCI and psychology to study how developers work, and consequently, how tools might better support these natural working processes [30, 58, 100, 111, 112]. However, we are not aware of any direct influence from linguistics on work related to the human factors of programming despite likely relevance when considering text-mediated interactions. Well-established linguistics concepts articulate how humans often recognize subtle meanings in natural language prompts and instructions which can affect their performance on tasks in studies. And decades of research studying how humans learn natural languages seems to have direct application to how humans learn programming languages.

3.1 Pragmatics

The area of *pragmatics* within linguistics is the study of how language choices are made with an eye towards communicative intents: cases where an utterance[5] (or perhaps, program) is structured to not only convey some fact or knowledge (or functionality), but also to emphasize some aspect of what is uttered as particularly salient, or more generally to communicate more than literal meaning (e.g., to imply something by choice of wording without saying it explicitly). This latter and most general framing is called *implicature* [70]: the enrichment of literal meaning with what was likely intended by the speaker, and specifically intended by the speaker for the recipient to understand — again, without explicitly saying so.

Central themes of pragmatics emphasize the idea that communication is geared towards a specific purpose, and towards a specific audience. Most work in pragmatics assumes that the speaker (or author) and hearer (or reader) have certain common background knowledge that both are aware of, and are cooperating towards a certain communicative goal. Under these assumptions, it is commonly believed[6] that speakers say only as much as is necessary to make their point, and no more, in part because they expect recipients to be able to *infer* certain implications called *implicatures* [70, 89].

Under the assumption that humans who behave in these ways with non-technical communication carry at least some of these behaviors into technical settings, this has clear relevance to any analysis of program-related text (as in Section 5) or any other natural-language artefact associated with software — including how study participants or users interpret instructions or specifications.

Classic introductory examples include:

- *Conversational Implicature*: Indirectly answering "Did you get the code working?" with "I've spent the last week debugging" does not directly literally answer the question, or logically imply success or failure at the task. But it strongly suggests the time was spent debugging without success, and this is moreover the likely intended point the speaker wished to make.
- *Deixis*: Deixis deals with contextual reference. Consider the comment "The next line of code…" in the midst of some code. This is a form of *textual deixis*. This refers to a specific line by virtue of where the comment is located in the program text, and moving it to another location would change the comment's referrent (as well as probably making it incorrect). Similar deixis occurs in comments that refer to the current method, class, or project, or to invariants that are expected to hold at specific program locations.
- *Presupposition:*[7] Consider comments along the lines of "This method is called when the Foo component…" which *presupposes* both the existence of a component known as Foo, and that the reader of the comment knows which component that is. In cases where one or both of these are untrue (either because Foo has been

[5] Any linguistic fragment, from an expression to a sentence to a passage of text or more, whether spoken, written, signed, or communicated by other means.

[6] Based on a wide variety of evidence, including corpora; see Section 5.

[7] There is some debate as to whether presupposition is a matter of literal semantics or pragmatics [144, §2.3.1], but pragmatics are generally agreed to play a major role.

removed and the comment is not out of date, or because the reader is a new team member) the meaning is unclear, or effectively undefined, due to *presupposition failure*. Similarly, a classic style of example is to note that the statement "I found my third major bug for the day" presupposes the existence of two other major bugs found earlier in the day.
- *Scalar Implicature*: If a developer writes that using a certain library is "acceptable," this *implicates* that the use of the library is not better than merely acceptable, because the author could have used a stronger descriptor (e.g., "good," "great") but chose not to in order to avoid saying something they believed to be false.

These are demonstrative examples of pragmatic phenomena.

It is well-established in the software engineering literature that concerns beyond mere program correctness factor into choices about how to structure programs [20], comment code [137, 173], or even choose identifiers [51], notably a focus on ensuring others can understand the code. While code is constrained by its primary function (to be executable), as is often noted these practices double the purpose to communicate with humans as well, in a way similar to what pragmatics studies: more is (or at least, can be) communicated by a piece of code than its mere functionality, just as more can be communicated by an utterance than its literal denotation.

Pragmatics also carries some more concrete connections.

Wording in Human Surveys. Awareness of pragmatics is particularly relevant to study design involving human participants. An infamous experiment in behavioral economics [181] has been used to claim that people do not understand conjunctions, specifically that they do not understand that given two sets $X \subseteq Y$ and a random point p, the probability of $p \in Y$ is larger than the probability of $p \in X$, seemingly misunderstanding that the latter implies the former. Of course, the wording used to ask the question is highly relevant, and in the case of the original study, key. A core pragmatic principle is that if someone (e.g., an experimenter) says something that does not quite make sense, then they must in fact mean something slightly different. In the case of this result, the question posed is an example of a *Hurford disjunction* [90]: a disjunction that is infelicitous because one disjunct implies the other. Logically there is no issue here, but if you were asked whether, given the knowledge that Erin liked Python's decorator feature, it was more likely that Erin was a Python programmer or a programmer, you would notice that the question is a bit "off." Obviously the set of all programmers contains the set of Python programmers, so surely your interrogator intended something more nuanced — otherwise it would be a silly question to ask. Perhaps it was intened to ask if it was more likely Erin was a Python programmer or some other kind of non-Python programmer? In that case one might reason that Erin is clearly a programmer, and because of their interest in Python decorators, more likely a Python programmer than a non-Python programmer.

Of course, the reasonableness of that conclusion depends critically on the inference that what the questioner meant was not the silly question that was literally posed.

In the case of this infamous behavioral economics result however, the questioner-intended meaning was the most literal one, where of course $\Pr(Y) \geq \Pr(X)$ when $X \subseteq Y$. But the survey instrument was flawed, because most people will draw inferences like the above when answering [63]. Later work showed that improving the wording or allowing consultation reduces the number of people choosing the "wrong" answer [29, 121], but the effect remains because fundamentally, the situation posed by any rewording of the question remains a Hurford disjunction at its core.

Clearly, we can see the relevance of Hurford disjunctions to survey design in PL and SE research: accidentally posing questions of that form would be confusing to participants, and would likely not measure the intended phenomenon. But of course, this is just one prominent example. Any of the pragmatic phenomena mentioned earlier (or those we did not mention [89]) could lead to similar misunderstandings. In areas where we spend much of our time dealing with strictly logical interpretations of text, it's essential to remember that there is more to natural language than the narrowest logical interpretation, and there is more structure to that gap than mere ambiguity. Awareness of specific pragmatic phenomena (specific to the human language at hand!) can help us avoid confusing participants, and improve the chances of measuring what we intend.

Interpreting English Specifications. Closer to home, pragmatic implicatures involving time seem to be at play in some observed experimental results in human studies in formal methods. Greenman et al. [69] describe a series of experiments analyzing the mistakes made by both novices *and experts* in translating English behavioral descriptions into Linear Temporal Logic (LTL) [142]. Two of the interesting categories of mistakes made (by both novices and experts) in that setting were leaving off a global "always" modality surrounding the LTL translation, and missing "eventually" or "next" modalities in the middle of formulae. These mistakes significantly change the meanings of formulae. LTL specifications are formulae which are evaluated with regard to a sequence of program states or events for a particular execution. The main specification formula is evaluated in the first state, and talking about later states requires using modal operators. So, for example, leaving off a "next" modality, and writing ϕ instead of $\bigcirc\phi$ (to indicate ϕ should be true in the immediately following state) means talking about what is true at the wrong moment in time.

The former appears related to the concept of *generic* sentences, or more precisely the subset also called *habitual* sentences, which describe recurring states of affairs.[8] An LTL formula such as $\phi \Rightarrow \psi$ (ϕ being true implies ψ's truth, a common form to formulae with this mistake in the experiments) indicates that if ϕ is true *right now* (at the moment of formula evaluation), then ψ is true at the same moment. However, most LTL formulae with such shapes occur under an always-modality, e.g., $\Box(\phi \Rightarrow \psi)$, which intuitively says that it is *always* the case that ψ is true when ϕ is true, at the current moment and all future moments. A sentence such as "If the train is arriving its warning lights are on" has readings of both forms — with and without the modal. In normal natural language contexts, most English speakers will interpret it with the habitual reading, as that is the most likely intended meaning. However in teaching students to use formal logic we tend to teach students to suppress inferences of things not explicitly present in the text, and indeed there is no explicit marker of habitual aspect in this example sentence (such as preceding the sentence with "always" or "generally"). However, the intended interpretation here expects participants to formalize this inference which is *not* explicit in the text; learners of LTL therefore must learn that in this domain this implicature should be carried over explicitly from natural contexts to formal contexts.

This is not the only example in their study results. Another classic example of temporal implicature is given by the classic linguistics example "I climbed on my horse and rode off into the sunset." The word "and" here is normally pragmatically enriched by most who read it to mean the speaker first climbed onto the horse, and *immediately after that* rode of into the sunset (on the horse, rather than in a dune buggy). This implicature is frequently explained in linguistics texts as the "and then" interpretation. This also occurs in examples in Greeman et al.'s study. In the coded responses, a number of the responses reflect a possible assumption by the participant that the formal "and" or "implies" implicitly shift the right operand later in time than the execution fragment where the left operand holds, such as responses including terms such as $\mathsf{Engine} \Rightarrow \Box(\neg\mathsf{Engine})$, which is trivially false because the always modality's argument is evaluated in the same state as the hypothesis of the conditional — it literally means that if the engine light is on now, then both right now and forever after the light is off. This was likely intended to be what we would actually write as $\mathsf{Engine} \Rightarrow \bigcirc\Box(\neg\mathsf{Engine})$ (i.e., if the engine light is on now, then *starting in the next state* ($\bigcirc$) the light is off from that point onwards). This is a slightly different kind of error from the above, where intead of neglecting to formalize an expected pragmatic enrichment, the participants assumed the formal logical expression was subject to further pragmatic enrichment.

[8]Generic sentences are broader, including other non-temporal generalizations such as the popular "tigers have stripes" and "birds fly" examples.

Thus pragmatics seems to fully explain some common mistakes, and appears to be a contributing factor to others. This makes it relevant to both understanding the mistakes, and considerations for design of new specification languages.

3.2 Language Acquisition and the Learning and Design of Programming Languages

Pragmatics is not the only linguistics topic of relevance to human factors. Linguistics underwent a major shift in the early-to-mid 20th century towards the science of language learning [128]. This body of work collaborating with neuroscientists and psychologists resulted in voluminous knowledge about how to teach a second language, and how to teach subsequent new languages (which, it turns out, have important differences from acquisition of first and second languages [11, 59, 151, 186]). Critically, this body of work offers models of language acquisition that predict stages of acquisition and common mistakes at different stages.

It is now well-established [145] (and has long been suspected [154] and studied [108]) that natural language aptitude is a significant predictor of success in learning to program. Thus it seems natural to explore the adoption of natural language teaching techniques into how programming in general, or even for specific language features, is taught. Work on programming education appears to have independently discovered echoes of results from the teaching of natural languages, whose literature could have been at least anticipated some of these results based on accepted models of acquisition. This strongly suggests that the existing linguistics literature could also be the source of inspiration, guidance, hypotheses, evaluations, and other criteria directly useful to work on programming education and learning of new programming languages. This idea, broadly construed, is not a new idea: Robertson and Lee [150] suggested this 30 years ago, though left the suggestion as a broad recommendation to emphasize considering several broad themes of import (in 1995) to second language teaching, most of which were not explained in sufficient detail to act on. Here we try to give more specific guidance about parallels and relationships.

Language acquisition is the area of linguistics (overlapping with psychology and other cognitive sciences) focused on how humans learn natural languages. As we will explore, much of this work seems highly relevant to the learning of *programming* languages as well. In this domain, a person's first language is often short-handed as their L1, their second language as L2, and so on, generalized as L*n*. While this phrasing assumes a linear order to language learning, variants cover cases such as a *simultaneous* L1/L2 for people who are raised bilingual (and simultaneous L1/L2/L3, and further, are also possible).

It is not clear whether the first programming language learned is closer to first language acquisition or second language acquisition (or later acquisition, for novice programmers who are already bilingual or multilingual when they

learn their first programming language), or really in between (with unique features to the learning of programming), but there are clear parallels after the first programming language. It is common knowledge (common belief) among computer scientists that once you know a certain programming language, it is usually easiest to learn new languages in the same family (e.g., another dynamically-typed object-oriented language, or another statically typed functional programming language), and bridging to different language families is more difficult — students have an easier time transitioning from Java to C# than from Java to Racket. This mirrors findings among families of natural languages [151].

We will focus somewhat more narrowly on the problems of what makes specific programming languages or features easier or harder to learn, more or less intuitive. Much work has been done in this space [17, 19, 76, 97, 104, 115, 131, 157–159, 172, 180], though we are unaware of direct application of ideas from natural language acquisition to the acquisition of programming languages (though language acquisition work is not fully disjoint from the broader learning sciences). Some of claims that have drawn the most attention in this space are based on studies with clear analogues to natural language scenarios which highlight confounding factors known to be highly problematic in similar linguistics studies; or yield results which are analogues of known results in linguistics. In the former cases, familiarity with linguistics work on language acquisition would suggest relevant confounds which could be controlled for. In the latter case, the studies remain worth carrying out for confirmation (natural and programming languages *do* have differences!), but awareness of results in language acquisition changes the perspective on what the anticipated outcomes should be — explaining why at least one seemingly controversial paper's results are actually quite natural, given that the experiment does not seem to be testing what was intended.

In what follows, we focus heavily on the learning of *syntax*. Naturally we wish for our students to understand more than the syntax of programs in a given language, but also general computational concepts that are not directly connected to language — what a program *denotes*. But an easy-to-overlook aspect of human language acquisition is that some baseline amount of syntax must be learned before language meaning can be addressed [141] (there needs to be a language fragment whose meaning can be learned!). It is well-known both anecdotally and scientifically, that the need to learn syntax is not only a major major barrier to novices learning programming languages, but also the first encountered [42, 43, 80, 172, 175], leading to work on languages specifically designed to account for this [80, 81] and pedagogical consideration of languages that seem to have less complex syntax for basic programs (notably Scheme and dialects [22, 23, 52, 53, 55, 56, 104, 148], though other aspects of the language's semantics and amenability to metaprogramming play a role in those explorations as well). Existing results in applied linguistics predict or explain some of the more surprising findings.

Static Typing. In a paper that drew significant attention at the time, Hanenberg [76] carried out a study which showed no improvement in outcomes for 49 students using a typed dialect of a new programming language, compared to a group using an untyped dialect. Students using the typed variant took slightly longer to get a working (executable, not necessarily finished) parser (the goal project). The quality (number of bugs) between the typed and untyped groups was similar. However, Hanenberg's students were still novice programmers, drawn from a pool that had completed a single Java course. The students were given a tutorial on the typed or untyped variant of the new language, then given 27 hours to implement a parser. This experiment was taken as a failure of static typing to provide benefits.[9] We will discuss how work on *natural* language acquisition predicts this outcome, and moreover provides an explanatory mechanism for this outcome (thus should be used to guide future experiments with similar goals).

Consider the analogous natural language setup: a group of students whose only working language is German are asked to write small passages in either English or French after a short tutorial. Why is this analogous? German is a language with 3 grammatical genders, where nouns are declined differently for every case, each roughly akin to a kind of typing of referents (so roughly, Java, or according to [130], Rust). English and French both have similar word order to German (for basic sentences, all three admit SVO word order). All three languages require agreement in number (singular vs plural) between verbs and subjects. However the English case and declension system is greatly simplified compared to German: English has no grammatical gender, and aside from pronouns, nouns are typically not declined according to case. French has only 2 grammatical genders, and like German requires adjectives to agree with nouns in gender and number. English and French themselves have very similar word order.[10]

In this case, essentially all major theories of second (L2) [160, 182] and further (L3/L*n*) [59, 151] acquisition predict that the initial mental model for a new language is either the full syntax of a known language, or at least defaults to reusing syntactic assumptions of known languages. In the parallel natural language setting above, we would expect L1 German students just learning English or French to produce English or French with similar sentence structure to correspondingly simple German sentences. We would not expect the students to produce English sentences any more quickly than they

[9] Note, however, that this was not the end of this line of research.

[10] For example, both tend to collect verbs together, as in "I want to read the book" or "Je veux lire le livre," contrasted with German's movement of non-primary verbs to the end of the sentence, as in "Ich will das Buch lesen."

would French sentences, because they would essentially still be mentally working in German. For learning a second programming language after learning a typed language as the first, applying any theories of natural language acquisition would predict that novice learners of the new language would *continue to structure their code in a type-based way, even without type-checking feedback*, because those are the starting mental structures that are available.

So in effect, Hanenberg's experiment did not test whether there are benefits to static typing, because *all participants were effectively adhering to the same syntactic criteria.* Instead the experiment tested whether students who knew only Java[11] would initially continue to write Java-like code in a new OO language even without a the demands of a type checker; nothing in the experimental setup would suggest the students were informed of the relaxed syntactic constraints of the dynamically-typed language, or ways to take advantage of the relaxed syntactic constraints. Even if they were, short instruction and limited practice are not enough to change the default mental structurs fluently. So as novices, they most likely did what natural language learners do initially: early-stage learners, when a target language permits grammatical constructions familiar from a known language, tend to closely follow the familiar structures valid in both one or more known languages [151, 186]. This is a familiar experience for those of us who remember the early stages of learning a second, third, or further natural language.

From that point Hanenberg's results make perfect sense. Both groups have similar error rates because they're effectively trying to write the same code, in the same *intended* grammar, regardless of which experimental group they were put in. This critique was raised intuitively by researchers around the time the study was published, but essentially based on "gut feeling." This concern turns out to actually have backing in the linguistics literature.

This also explains why the typed group took longer on average to complete: both groups were trying to write the same code, but only the typed group was told (by the type-checker) when they deviated from their intent and was then forced to correct things. (Interrupting a non-native speaker to get them to correct grammatical mistakes slows their communication by virtue of interrupting them.)

A computer scientist may object at this point: surely the experimental language variants had a parser, and both groups received parser errors! So how is static type checking related to syntax? It is true that there was a parser for both language fragments, but the assumption that type checking is not about syntax neglects that programming language implementations typically take a narrower view of syntax than formal language theory, treating anything beyond an $LR(k)$ grammar for small k as "semantic analysis" in the parlance of influential compiler texts. But of course, many of those "non-parsing" tasks are categorized that way because they are not efficient to implement in classical CFG-based parsing frameworks. However, as we discuss in Section 6, properties such as identifier scoping and even simple typing are problems in the domain of context-sensitive languages, not much beyond the class of mildly-context-sensitive languages that seems to bound human languages [21, 86, 166]. And particularly for novices, both classes of feedback appear as "shapes of code that are allowed or disallowed."

The point here is that decades of research on language instruction predict aspects of Hanenberg's results. Still, while the linguistics literature predicts that this experimental setup would not measure what was intended, this extrapolation we have just described does not automatically hold true — the experiment is still valuable because it provides a confirmed instance of an experiment in programming language acquisition behaving as predicted by natural language acquisition research, suggesting that other predictions from extrapolation of language acquisition work are worth specifically considering and checking. However, the analogy to the linguistic context does change the framing of the experiment: it makes Hanenberg's findings *the expected results* given the experimental setup.

Many of the ideas from the linguistics literature that predict these outcomes are from the 1990s and early 2000s, though much of the specific evidence we are aware of, cited above, stems from notable increase in L3 acquisition work from roughtly 2015 onwards; thus we do not believe the results were necessarily obvious extrapolations from what was known at the time of the study. However, given the connections described above, it seems likely that researchers pursuing empirical studies on syntax in particular, and learning of programming languages in general, would benefit from examining analogous natural language acquisition settings when designing experiments and framing results.

Intuitiveness of Syntax. Another appealing measure of a programming language design besides productivity is whether or not the structure of the language is *intuitive*, as fuzzy a notion as that is. Our colleagues in design and HCI deal with this notion themselves, and it is worth remembering that what counts as intuitive depends heavily on past experience and recent use, and can thus vary widely across individuals. This is another place where decades of linguists' experiences investigating the factors that make language acquisition easier or harder has strong relevance.

Much of the current literature on learning the syntax of programming languages (as typically understood in computer science, excluding type systems) also presents findings as surprising which are arguably predicted by linguistics work on language acquisition. Monolingual learners of a second language (L2) initially transfer expectations about

[11] With the possible exception of some students with prior programming experience.

syntax and lexical entries (words) from their one known language [61]. Bilingual or multilingual learners of an additional (3rd or later, L3/L*n*) transfer grammatical expectations from at least one known language, with strong evidence in favor of mixed transfer from multiple known languages [11, 151, 186], and additional evidence in favor of transfer specifically from the most typologically (structurally) similar language. Multilingual learners who use one language significantly more than others are additionally more likely to transfer from their dominant language [146], though it remains unresolved how this tendency is balanced against closer similarity between a non-dominant known language and the target language. Critically, in any case, which language(s) a learner already knows, and even the order in which they were learned [11] and relative proficiency levels [161] has influence on the initial learning process for any new language.

Adapting this framing to consider what programming language syntax true programming novices would consider easiest to learn yields some natural expectations. Monolingual novices (with no programming experience) would likely consider programming language syntax with grammar and lexical entries (keywords) closest to their known language more intuitive, because they could transfer more expectations from their original language without error. Bilingual or multilingual novice learners (with no programming experience) who predominantly use one language would find programming language with grammar and lexical entries more similar to their dominant language to be more intuitive.

Adapting this to learners who have some prior programming experience naturally becomes more muddled, as it is clear that programming languages have significant differences from natural language (e.g., compilers and interpreters do not engage in pragmatic implicature), but they are still languages with lexical entries and grammatical structures which can be transferred. Theories of L3/L*n* acquisition consistently assume most if not all transfer will be from either the most dominant language or the most typologically (structurally) similar language [151, 186], but for learning a second *programming* language, it seems reasonable that learners would be more likely to transfer from their known programming language than from one of their natural languages. So we can make reasonable extrapolations for learner who may be monolingual or multilingual with regards to natural language, but have basic familiarity with (exactly) one programming language: most (or all) initial transfer would come from the (one) known programming language.

For learners who have already gained significant experience with multiple programming languages, it seems reasonable to project based on interpreting L3/L*n* theories of acquisition with regard to known programming languages. In this case, programming languages with grammar and lexical entries (keywords) more similar to known languages would be more intuitive, with programming languages where those features most resemble the learner's most-used (dominant) programming language being even more intuitive due to greater opportunity for correct transfer from the most familiar context.

In light of this, let us consider a study from Stefik and Siebert [172], widely noted for finding that Java and Perl's syntaxes were not significantly more intuitive to novices than a language with randomly-chosen keywords, while Ruby, Python, and Quorum (a language designed by those authors and collaborators to have more English-like syntax) were deemed more intuitive based on novices' accuracy in intuiting small programs based on examples in one of the languages. These were the take-aways from only two of the paper's experiments (Studies 3 and 4); two other experiments (Studies 1 and 2) asked either first-year CS students or later-stage students to subjectively rate how intuitive the syntaxes of various programming languages were. A common critique of this paper (more precisely of Studies 1 and 2 in the paper) is that what is intuitive is highly subjective, and depends on prior exposure to related ideas. This is a natural observation which is acknowledged in broad terms as a possible confounding factor by Stefik and Siebert. Stefik and Siebert's accuracy evaluation (Studies 3 and 4) does directly address this, by asking non-programmers to write a specified small program based on an example in one of the experimental languages (each participant was only shown examples in a single language), and evaluating how close the responses were to an intended program. A subset of the (extensive) findings addressed that early students in Studies 1 and 2 generally preferred English keywords to punctuation, with the partial exception of preferring a single equals sign for equality comparisons, and that non-programmers had higher error rates with Perl, Java, and a language with randomly generated keywords than with Ruby, Python, or the Quorum language designed by the authors and others (the study is infamous in part because the mean accuracy for participants in the Java group for Study 4 was only marginally better than the random-keyword language). These results are valuable, but presented as at least somewhat surprising. We would argue that these aspects of the results are actually anticipated by extrapolating language acquisition research results to programming languages. (Though again, this experiment was worth performing to confirm that the programming language case behaves consistently with knowledge about natural languages.)

A general trend in Stefik and Siebert's results was that English keywords were favored over punctuation by novices, and languages that make heavier use of punctuation for operators had lower accuracy for non-programmers. This is at least partly explained by the lack of transfer of lexical handling from natural languages and limited transfer from some programming languages for punctuation. In natural languages, punctuation is primarily a marker used for grammatical guidance (such as nesting appositives in parentheses) or prosody

(e.g., pauses), and is not meaning-bearing.[12] But of course, ++ and similar operators in imperative programming languages *are* meaning-bearing! So this is loosely akin to learning a language using a different script,[13] which is known to introduce additional difficulties in language acquisition [5, 13, 125].

Beyond predicting this trend, the linguistics literature suggests refinements to the participant groupings for Studies 1 and 2 (of novice and experienced CS students). As noted above, which language(s) a learner already knows when learning a new language has a significant impact on many aspects of learning [11, 98, 151, 186]. Stefik and Siebert describe their subjects for Studies 1 and 2 as "from a variety of courses in the computer science department, including freshman through junior and senior level courses that are taught in a variety of languages (e.g., C++, Java)" [172]. Their background surveys cover 13 and 15 languages (for their syntax experiments), covering quite a few different programming language families. So while the varied experience levels (years in the CS program) were controlled for by separating the groups, varied background (exposure to various subsets of programming languages prior to the study, presumably learned in varying orders), and unknowns regarding participants' dominant programming languages, treating this breadth of subjects in a single pool seems to be a more likely confounding factor.

This suggests likely follow-up studies to tease apart these subpopulations with refined experimental design. Natural language acquisition studies typically select target populations to specifically control for these factors, gathering participants with specific amounts of experience, and pairing populations. For example, rather than simply asking about the intuitiveness of a certain language syntax from a broad pool, selecting groups of L1-Python/L2-Java participants, L1-Java/L2-Python participants (with appropriate criteria for what counts as an L1 or L2 programming language), and investigating those participants' views of a range of syntaxes. Critically, the languages whose syntaxes were polled should not overlap the set of known languages, though this is likely harder to achieve in participant recruiting for programming languages than with natural languages (and notably, cannot be done solely with participants from a single academic institution in most cases, unless there are multiple introductory sequences with opposing orders). Even better would be to additionally compare to L1-Python and L1-Java learners who knew nothing of the other language (e.g., L1-Python with no Java exposure and vice versa).

Language acquisition work also highlights an important understated limitation to the study, which was the exclusive use of participants with high English proficiency. The paper notes that only 2 out of 196 participants in Study 1 (just over 1%), and 7 of 166 participants in Study 2 (just over 4%) were non-native English speakers, meaning almost 99% and 96% respectively were native English speakers. The study was conducted at a US university, all of which require strong scores on English proficiency exams for international students whose home country's official language is not English. Numbers are not reported for Studies 3 and 4. So the study results are, at best, true only for high-proficiency English speakers; students whose primary language was not English may have found different syntaxes more intuitive than this group, based both on linguistics results and more recently on experiences with Hedy, with the notable example of supporting right-to-left syntax for users whose dominant language is right-to-left [175]. This is especially relevant for low-proficiency English users [72].

Again, we must conclude that familiarity with the science of how people learn natural languages is highly relevant to studies of learning programming languages.

3.3 Other Linguistic Human Factors

Many other human factors aspects of program development would benefit from linguistics knowledge. On the language use side, we focused on subtleties of the communication via an individual utterance in some abstract notion of a usage context. However, the linguistics literature goes much deeper, and considers other aspects of communication as well. *Discourse analysis* is the study of how longer language exchanges — notably *interactions* — are structured. In Section 5 we discuss this in more detail in the context of analyzing developer communications, but it also seems relevant to the design of interactions with tools. On the language learning side, there is a rich literature on *individual differences* in learning, connected to this term in the broader field of learning science and its particular intersection with linguistics, going well beyond some of the high-level mention in this section of factors like which specific languages a learner already knew, and what order they were learned in. One example with a clear analogue to programming languages is that extramural use of a language (i.e., use outside the classroom and assignments) has a significant impact on gaining proficiency [174]; this connects to student use of a particular programming language in hobby projects, hackathons, and jobs. Given that most of our programming languages, tools, and knowledge repositories emphasize the textual form, knowledge of how humans interact with natural language seems worthwhile.

4 Evolution of Language Design and Use

Let us now shift our perspective slightly, from the focus on individual humans thinking about programs to the languages themselves, but still considering that humans as a group play a role in the design of programming languages. There is a question of to what degree programming languages

[12] Hermans [81] shares an amusing anecdote of a student pausing mid-sentence when reading aloud an error message involving a comma, rather than naming the comma.

[13] We avoid the word *alphabet* because not all languages use an alphabet, e.g., Mandarin and Japanese.

are designed vs. evolve. Unquestionably, programming languages are designed and engineered in ways that human languages (with the exception of *conlangs* [64][14]) are not. Some authors [77, 171] have critiqued the design process of programming languages for not basing every evolution on controlled trials showing benefits.[15] However, even setting aside the narrow epistemology of expecting every language change to bring widespread obvious benefits to all users while language designers acknowledge adding features useful in narrow but high-value cases (Gibbons [62] points to other flaws in this thinking), the choice of language feature inclusion has always been as much a social process as a technical one. Anyone who has been involved with a language standardization effort, or even a proposal for a language enhancement, will readily admit that usually the social case must be made before technical feasibility is even considered.

A linguistics area of relevance is the study of *language contact and change* [85], which studies how different languages change when used simultaneously by the same people. An analagous situation arises in the creation of software: a plurality, if not a majority, of developers work with multiple programming languages, and naturally habits from one language tend to carry over to others (see earlier discussion on language acquisition), sometimes leading to growing demand for new language features and thus to language extensions, either officially or via libraries or metaprogramming. Examples of this can been seen in the gradual spread of asynchronous programming constructs like async-await spreading across established languages (e.g., from research languages [122] to niche production languages like F# [176] to mainstream languages like C# [15], and JavaScript [114]), or type classes (popularized by Haskell [74] before spreading to Scala [132] and Rust). At a minimum developers working across different languages are known to result in code written in one language (e.g., Python) in a style more typical of another (e.g., Java) — *non-idiomatic* code. Such code may be more difficult to understand (or less efficient to execute) than idiomatic code, leading to existing efforts to automatically normalize non-idiomatic code [188–190].

Two major ways programming languages evolve are by borrowing features from other languages and by developing shorthands for expressing common idioms that were originally more verbose. It turns out, these are also the two major ways that human languages evolve:

- Languages in regular *contact* [85] (in the sense of having a large number of multilinguals working in that particular combination of languages) tend to absorb features either by intentional mimicry (loanwords and borrowing phrases across languages [143]) or accidental cross-pollination (when non-native speakers encounter an idea they wish to communicate but lack the lexical or grammatical knowledge to express it in the target language, they often fall back to mixing in lexical entries or grammatical structures from a more dominant known language [153]), which sometimes stick.
- Idioms and additional meanings develop via a variety of methods, but most prominently via gradual merging of words and dropping of portions [103]. Linguists believe this is a natural mechanism to optimize communicative efficiency.

We can see examples of human language transfer in action even in our own scientific literature. For example, there are examples throughout the literature of a grammatical construction involving "to ⟨*verb*⟩" in places that native English speakers deem it awkward, such as "to synthesize a type is straightforward" rather than "synthesizing a type is straightforward." These arise from work by authors whose primary language lacks a direct equivalent of the English gerund (the -ing ending). For example, the sentence above in German would be either "einen Typ zu synthetisieren ist unkompliziert" (the most parallel translation) where "zu synthesisieren" translates word-by-word as "to synthesize" or "das Synthetisieren eines Typs is uncompliziert" (literally, "the synthesis of a type is straightforward). As this new form proliferates, it is likely to eventually be accepted as typical by native English speakers, used by them outside academic contexts, and so on. Or consider that a common construction in linguistics text is to say that some construction or analysis or interpretation "obtains." The word "obtains" in typical English requires a direct object that is the thing being obtained, where the subject is the actor doing the obtaining; linguists however use "X obtains" to mean that someone or something managed to "obtain X." For example, "the typing derivation obtains" or "the analysis result obtains."

We can see examples when developers attempt to carry idioms from one programming language to another. Complaints about such idioms seeming natural but either being disallowed or performing poorly lead to language changes. Consider Java finally acquiescing to add closures in 2014.

Many programming language changes are driven by similar shorthanding mechanisms, such as the now-popular addition of for-each loops to replace manually writing code to retrieve and then loop through an iterator.

Other changes are a combination of both. Consider the relatively recent addition of pattern matching switch statements to Java. This is a feature that was common in other related languages (C# and Scala as closest competitors in the field, besides the decades of earlier examples) but were added largely because it permitted more concise expression of patterns that were already prevalent in Java code. The documentation [133] explicitly motivates the use of every

[14] *Constructed languages*, which are mostly languages designed for hobby purposes, though occasionally for books or cinema.

[15] Though we also note they do not demand the same evidence for library evolution, despite many libraries acting as embedded domain-specific languages.

form of pattern matching as a more concise alternative to common idioms.

These aspects of natural language evolution, and clear parallels in how programming languages do evolve in practice, further suggest considering the common-sense position that programming languages, like natural languages, are subject to multiple constraints beyond learnability.

When it comes to studying change in usage over time, the broad family of *corpus linguistics* techniques has relevance, given the abundance of studies in how usage of language features changes over time, via the intersection of *historical* or *diachronic* linguistics (the study of language change over time) with corpus linguistics [87].

5 Corpus Methods, for Programs and Text About Them

Let us now shift our attention down another layer of detail, to focus more squarely on linguistic textual data (whether text *about* programs or the text of programs themselves) and *how to process it*, through the lense of methodological tools from linguistics with relevance to PL and SE. *Corpus linguistics* offers many parallels with and lessons for PL and SE research. One of its major domains of application has been to the study of extended conversational texts, a topic of perrenial interest in software engineering.

Corpus Linguistics and Empirical Studies. Empirical studies have played a significant role in software engineering [10, 16, 18, 35], program analysis (of all varieties) [95, 129], and even the design of programming languages (the classic value restriction for ML polymorphism was based on a simple corpus study [187], and more recently adjustments to Julia's type system have been validated based on an extensive study of public Julia code [14]). Likewise, empirical analyses of the relative frequency of various linguistic phenomena has been a source of great knowledge for linguists, and is roughly as old as software engineering, with the practice emerging in earnest starting in the 1950s [120], and solidly established by 1967 [105].[16] The simplest form of corpus linguistics studies relative frequencies of different occurrences (e.g., of words (known as lexical variation), or grammatical constructions [4]), which may (as in PL and SE research) be either manually annotated or programmatically identified. Another step in that direction concerns the change in properties of corpora over time, which has a long history in linguistics [87], and which programming languages and software engineering researchers have reinvented alone to study phenomena such as the adoption process of Java generics [136]. While our motivations for such studies might differ from the motivations for studying the (once-contentious [25]) emergence of singular "you" to displace "thou" in English, many of the same principles apply to data gathering and analyses for these problems.

Much empirical work in SE and PL independently motivates separate techniques from corpus linguistics, based on general statistical knowledge (e.g., a notion of seeking a representative sample) and the particulars of the domain and question at hand. While such results are valid when well-executed [9], it seems highly unlikely that empirical software engineering has already independently invented *all* of the relevant insights used in another discipline that studies relative frequencies of phenomena in largely-textual data.

Corpus linguists have spent decades focusing on the finer points of obtaining reasonably representative samples and dealing in principled ways with known skew in datasets, including sampling across various subpopulations, and even articulating how to determine when a group *is* a subpopulation [47]. Indeed, there are examples of ways to examine corpora that we have yet to see reflected in many empirical SE studies or nearly any PL studies.[17] Most large-scale empirical investigations in our fields are focused on frequencies: how often a phenomenon of interest occurs, in any context. The corpus linguistics literature suggests additional views that would be of interest. Corpus linguistics has developed systematic approaches to studying various kinds of co-occurrence of features of interest [170, 177]. For example, one key technique beyond frequency is analysis of *collocations* [57, 113]: alternatively either the frequency with which two items of interest are adjacent or the examination of which pairs (or trios, etc.) of items co-occur together frequently. A natural SE application of this idea might be to analyze which APIs are used together frequently (while not necessarily being code clones), perhaps to identify candidates for refactoring out common logic [28]. Linguists have applied corpus linguistics techniques to study a range of other problems with analogues in PL and SE discussed later.

Discourse Analysis. So far we discussed the connection between ideas from corpus linguistics and analysis of software code. Unsurprisingly, such ideas can also be applied to natural language written *about* software or code, where the lessons of linguistic corpus analysis become even more relevant given the focus on natural language. In this case software engineering researchers have already pursued some forms of this, such as analyzing the linguistic structure of and categorizing noun phrases in bug reports [101], with an eye towards consequences. But software engineering research has also explored corpora of bug reports [191], chat logs [32–34], mailing lists [73], specifications [46, 149], and more.

At least some ideas from linguistic annotation of corpora have made their way into software engineering work [6], but there is more to be had.

[16] A determined proponent of linguistics could make a case that corpus linguistics is in fact a couple millenia older than software engineering, with early grammars of Sanskrit discussing relative frequency in written literature of the time [96].

[17] We are intentionally excluding here small-scale empirical investigations, such as when a pluggable type system is run on a small number of programs.

Linguists have also studied many of the same topics whose corpora are studied now in software engineering, though in broader contexts under the broad area of *discourse analysis*: the study of how meaning is conveyed across larger pieces of text (think software requirements) and interactions/dialogue (think developer chats and mailing lists). This includes study of both various forms of communication (e.g., chat messages [1], email [36, 39], contracts [68]) and cross-cutting aspects of communication such as how emoji [45, 118, 155] and humor [88] are deployed, and how work relationships between participants influence these interactions [75, 88, 140]. This work covers a wide range of phenomena in written artifacts which would seem to have relevance to software engineering analyses, and incorporates a variety of techniques and theories drawn from other areas of linguistics (notably sociolinguistics). Much of the relevant work is under the heading of *computer-mediated communication* or *computer-mediated discourse analysis* [83, 84], including work studying the choice of which communication method to use [126, 127]. And these are not merely case studies, but proposals for ways to systematically understand unique aspects of these forms of communication. For example, there is extensive work on *disrupted adjacency* [60, 82], where multiple threads of conversation are interleaved in chat or email contexts, which has been tackled in SE [31, 48] (note that the state-of-the-art in doing this for developer chats [31] is directly derived from techniques drawing on discourse studies in linguistics [49]).

Note that we are not suggesting that any of the SE work above is redundant; most discourse analysis work in linguistics is focused on understanding reasons for behaviors and factors in choices of behavior, and how information is communicated. Much (though not all) lacks proposals for algorithms to extract or reconstruct relevant views of information, and we are unaware of any that applies that knowledge directly to software engineering contexts. But it seems likely, given the overlapping interests of the communities, that there is opportunity for discourse studies to suggest approaches or explanations, or contextualize results, in ways complementary to existing SE work.

6 Analysis of Program Text

Finally, let us shift our focus down a few more notches. We have discussed high-level human factors, mid-level questions of how humans influence the evolution of languages, and research techniques. Let us now discuss specific technical tools from linguistics with direct relevance to programming language design and implementation. When designing and implementing programming languages, we spend a great deal of time dealing with the definition, modeling, and analysis of the syntax and semantics of programming languages. Again, linguists have decades of experience doing the same for natural languages, and have come from such a different perspective that they have concrete solutions to challenges PL and SE researchers are actively working on.

Syntax and Grammar. Practically every undergraduate computer science program, and most software engineering programs, require their students to take a course on formal language theory, focusing on finite state machines and automata, context-free grammars and pushdown automata, and Turing machines. Often lost in the shuffle is that these ideas arose originally out of linguistic investigations of the complexity of natural language syntax; the gradation of complexity between regular, context-free, context-sensitive, and recursive languages is called the Chomsky Hierarchy because Chomsky (a linguist) was the one who articulated the gradation [37]. These courses tend to gloss over the context-sensitive languages because their grammatical characterization is slightly awkward, and computationally the notion of a linear-bounded automaton looks like an artificially limited Turing machine — why bother bounding the tape? This conflation however has, we believe, limited the imagination of language designers, even as colleagues in computational linguistics explore grammatical formalisms with additional power and tractable parsing [93, 106, 107].

Consider that nearly every example of a language modeled on a Turing machine in popular automata theory texts — $a^n b^n c^n$, $a^n b^m c^n d^m$, $\{ww \mid w \in \Sigma^*\}$, and more — are in fact context-sensitive(!), and use only linearly-bounded amounts of tape. This conflation seems, anecdotally, to lead to a confusion that anything beyond context-free languages is hopelessly intractable, and that all programming language syntax is inherently context-free, when in fact the distinction between syntax and semantics is somewhat subjective. Consider the problem of checking that a program identifier is in scope. Typical language implementations parse a program using a context-free grammar that ignores scoping, then perform a series of analysis passes over that parse tree to check properties like scoping and typing. We have seen preprints of published machine learning papers (later corrected in press) which initially indicated surprise that context-free grammars were insufficient to avoid generating programs with ill-scoped identifier uses, even though identifier scoping is inherently a context-sensitive property, for the same reason XML is a context-sensitive language: the set of identifiers (or XML tag names) is not finite, so well-scoped programs / valid XML documents cannot be captured by a finite set of context-free production rules.[18] In fact, the form that scope checking takes — a recursive walk over a tree where identifiers are typically expected to be declared in higher/earlier nodes of a parse tree — can be modeled as a non-deterministic pushdown tree automaton; the yield languages of such automata

[18] Note that even matching a single pair of opening and closing XML tags requires matching the tag name in order — just validating that <mytag></mytag> has matching tags without hard-coding `mytag` in the grammar is non-context-free, a cousin of $\{ww \mid w \in \Sigma^*\}$.

(the in-order serialization of leaf nodes of accepted trees) are exactly the context-sensitive languages [156]. (This is why identifier scoping, and even type-checking for type systems without type-level computations, are in fact context-sensitive language problems, though they are typically implemented in terms of trees rather than strings.)

In short, context-sensitive syntax is plausible, and in fact already used in any statically-typed language, just formalized implicitly as the intersection of a context-free grammar and a pushdown tree grammar, an idea that has now arisen for other purposes in PL [3]. There is strong evidence that many human languages are not context-free [21, 86, 166] and lie in a restricted class of *mildly context-sensitive* grammars [93]. For example, Higginbotham [86] gives a fragment of English using gendered pronouns to demonstrate cross-serial dependencies of grammatical agreement, much like the cross-serial dependencies in Dutch [21] which inspire the textbook non-context-free language $a^n b^m c^n d^m$. These mildly context-sensitive gramamars can also be parsed efficiently [93, 94, 106] — not as efficiently in the worst case as context-free grammars of course, but efficiently enough that such grammars are plausible ($O(n^6)$ upper bounds rather than $O(n^3)$). In principle one could establish complexity results on various type-checking problems by connecting them to the rich literature on tree automata and grammars, though to the best of our knowledge this has not been explored.

A useful trend in linguistic grammars, as opposed to many of the context-free grammars used in programming languages and software engineering, is the emphasis on *lexicalized* grammar formalizations. These formalizations shift the emphasis from describing the abstract phrase types to describing simply how individual words combine with adjacent constructions. The result is that it becomes easier to specify grammars *modularly*: adding a new grammatical construction is "simply" a matter of giving a certain grammatical "type" to a new word. In a world of language workbenches [184], wouldn't it be nice to mix and match syntactic fragments modularly for experiments? In fact, there are already papers where this has been necessary, such as abstracting over the addition of new evaluation contexts for parameterized studies of effect checking [117]. One could also argue that work on metatheory a-la-carte [40, 41, 92] does something similar when defining grammatical structures indirectly, for example by defining ASTs using an algebra over partial Mendler encodings [99], though in that case a final fixed tree structure is still computed (as a least fixed point), so a full set of language (syntax) extensions must be fixed at some point in time.

Lexicalized grammars remove this restriction. Consider, rather than the classic CFG for arithmetic expressions:

$$e ::= n \mid e + e \mid e - e$$

a lexicalized version, using Lambek's syntax [109] and a couple *general* combinatory rules:

$$+ \vdash (e \setminus e)/e \qquad - \vdash (e \setminus e)/e$$

$$\frac{}{n \vdash e} \qquad \frac{s \vdash A \qquad s' \vdash A \setminus B}{s, s' \vdash B} \Leftarrow \qquad \frac{s \vdash B/A \qquad s' \vdash A}{s, s' \vdash B} \Rightarrow$$

This is a presentation of the same grammar as a Lambek calculus [109] with an additional unary rule to promote numerals to expressions. Plus and minus operators are each given a grammatical category, in this case both *slash* categories indicating they are syntactically directed functions. Each takes an e to its right, followed by an e to its left, resulting in an e (in both cases, the slashes act as grammatical function type constructors, and the argument is "under" the slash).

The first two lines describing the plus and minus operators are the *lexicon*, where each entry gives the assumed grammatical role for various lexical items (here the operators). We explain the grammatical roles momentarily.[19] The rules above have conclusions of the form $s \vdash c$, which we can read as saying "the sequence of words s can be parsed as belonging to grammatical category c. They are inference rules of the form common in logic and type theory: to conclude the claim below the line, one must prove (using the same set of rules) all claims above the line. s, A, and B are variables ranging over (non-empty, but possibly singleton) word sequences (s) or grammatical categories (A and B).

The first rule above can be read as saying that any number n is an expression (e). The second rule (labelled $\Leftarrow$) reads as saying that for any two sequences s and s', if s has grammatical category A, and s' has the grammatical category $A \setminus B$ indicating that it would be a B if an argument of category A were placed to its left, then as long as s is actually placed to the left of s', the two can combine into a fragment in grammatical category B. The final rule ($\Rightarrow$) is symmetric, with the "function" category looking to its right for an argument, and the fragment s' on the right is the argument in this case.

This presentation is similar to writing a CFG in Chomsky Normal Form, where all rules combining non-terminals are binary. The rules are in fact logical rules of inference, and a derivation is isomorphic to a (binary) parse tree (with leaves at the top). Thus the categorial grammar equivalent of a parse tree witnessing that 3*3 is an expression would be:

$$\dfrac{\dfrac{}{3 \vdash e} \qquad \dfrac{\dfrac{}{* \vdash (e \setminus e)/e} \quad \dfrac{}{3 \vdash e}}{*3 \vdash e \setminus e} \Rightarrow}{3 * 3 \vdash e} \Leftarrow$$

Presenting the rules this way — *lexicalized*, with each rule anchored to a piece of syntax — allows concise syntactic additions, specifying new constructs simply by giving new *lexicon entries* like those given for + and −. Specifying multiplication and division in this way is not particularly enlightening,

[19] It is possible to extend what we describe here to deal with deriving lexicon entries for derived word forms from their base forms.

but one could add an additional base category b (booleans), give every comparison operator the category $(e \setminus b)/e$ (looking first to the right for an expression, then left for another expression, resulting in a boolean expression), boolean operators $(b \setminus b)/b$ (similar, where the arguments are boolean expressions), and continue building, without revisiting the combinatory rules. The presence of the slash categories removes the need to name many intermediate categories, in exchange for some familiarity with functional programming or substructural logic. Parsing algorithms exist for this approach which do not require additional compilation or grammar transformation, making it suitable for use in language workbenches or for language support for syntactic extensions.

Nothing in this approach forbids giving multiple lexical entries for the same construct: try for example may have distinct entries for whether or not a finally block follows. Note that we ignore operator precedence here, but this (and other syntactic extensions) can be modeled in this approach as well. Lest this raise questions about how semantics might be given to programs with syntax given in this way, we discuss this next.

Semantics. "Formal semantics" in linguistics is the study of formal models of meaning, including meanings that may vary by context. Most approaches assign mathematical meanings to parse trees of sentences (or larger texts) compositionally, composing the meanings of subtrees to produce the meaning of a tree node much like denotational semantics assigns meanings to abstract syntax trees.[20] While many semantic representations are used in linguistics (and especially computational linguistics), it is generally agreed that any semantic representation used should have a denotation in terms of some formal logic, or a lambda calculus with a Heyting Algebra (an algebraic abstraction of logical operators) at its core [110]. Ultimately this should not be terribly surprising: the progenitor of most modern approaches to linguistic semantics was Richard Montague [123, 124], a logician by training (a PhD student of Tarski, in fact). Thus modern approaches to linguistic semantics and programming language semantics have common roots.

Natural language, like all programming languages, has expressions which refer to other expressions, including pronouns ("*it* is up-to-date") and some uses of definite articles ("*the* system orchestrator"). As in programming languages, some of these references are resolved statically, while others are resolved dynamically. Formal grammars with semantic components often include variants of a classic proposal [91] to add grammatical categories $A|B$, that is, an A with a B (the referent) missing somewhere internally, essentially tracking the set of free variables of an expression, plus rules that combine with the A while lifting the B (e.g., a C/A left of $A|B$ turns into $C|B$). There are also *dynamically* resolved references (akin to heap accesses), if semantics are enriched to pass a data structure modeling the current state of discourse (see Section 5), those correspond to dynamic lookups in a data structure [71, 119], as a special case of the use of (delimited) continuations in natural language semantics [12]. Thus one could use these grammar formalisms to specify the semantics of language construct extensions directly alongside syntactic extensions, in a fully modular fashion (recall that continuations can be used to express computational effects [54]).

As a concise example, we revisit the arithmetic expressions from earlier, this time giving denotational semantics as part of the grammar combination rules:

$$\frac{}{n \vdash e \Rightarrow \mathsf{parseNum}(n)} \qquad \frac{s \vdash A \Rightarrow a \qquad s' \vdash A \setminus B \Rightarrow f}{s, s' \vdash B \Rightarrow f\ a}$$

$$\frac{s \vdash B/A \Rightarrow f \qquad s' \vdash A \Rightarrow a}{s, s' \vdash B \Rightarrow f\ a}$$

This grammar is the same as the earlier example, extended with a term of a lambda calculus representing how meanings are composed, to the right of $\Rightarrow$ in each rule. The semantics of a (string representation of a) number are given by reifying that number as a numeric datatype. The semantics for the "predicate-argument" rules of the grammar are in fact function application: the text fragment whose grammatical category has a slash with an argument is in fact a function taking an argument, and its semantics (f) are applied to the semantics of the argument (a). Here, there is a strict relationship between the grammatical categories and their denotations: es denote numbers (we ignore representational choices here), slash types denote functions from the denotation of their grammatical domain to the denotation of their grammatical codomain. Lexical entries are similarly augmented with semantics. For example, the denotation assigned to $*$ would be $\lambda r.\, \lambda l.\, l * r$ (where the $*$ in the semantics is the denotational multiplication operation). Thus we can extend our earlier "parse tree" to both parse and assign semantics:

$$\dfrac{\dfrac{}{3 \vdash e \Rightarrow 3} \qquad \dfrac{\dfrac{}{* \vdash (e \setminus e)/e \Rightarrow \lambda r.\, \lambda l.\, l * r} \quad \dfrac{}{3 \vdash e \Rightarrow 3}}{*3 \vdash e \setminus e \Rightarrow \lambda l.\, l * 3}}{3 * 3 \vdash e \Rightarrow 9}$$

Note that the $*$ on the left is syntax for multiplication, while the $*$ on the right is the actual multiplication operation of the lambda calculus for the semantics.

As with the syntax example, this does not demonstrate the full generality of this technique. Studies of formal linguistic semantics have ventured far into the domains of semantics that exploit monads [8] and continuations [12] in conjunction with syntax (of natural language) given in this form. These ideas compose neatly with parses of context-sensitive languages as above, as the *result* of the parsing remains a

[20] This is slightly over-simplifying, since denotations are sometimes ascribed to derivations involving trees, rather than trees themselves, though the same caveat carries over analogously to our discussion here.

$$\dfrac{\dfrac{}{3 \vdash e \Rightarrow (\text{Const } 3)} \quad \dfrac{\dfrac{}{* \vdash (e \setminus e)/e \Rightarrow \lambda r.\, \lambda l.\, \text{Times } l\ r} \quad \dfrac{}{4 \vdash e \Rightarrow (\text{Const } 4)}}{* 4 \vdash e \setminus e \Rightarrow \lambda l.\, \text{Times } l\ (\text{Const } 4)}}{3 * 4 \vdash e \Rightarrow \text{Times } (\text{Const } 3)\ (\text{Const } 4)}$$

Figure 1. Producing a classic parse tree from a categorial grammar.

tree structure (recall that common restrictions on trees yield context-sensitive languages [156]). This can also be extended beyond the typical realms of semantics for natural languages, including generating formal specifications [67, 162, 163] or even tests [66] in fragments of English with limited vocabulary but no a priori restrictions on grammatical structure. This seems a promising toolbox for modularly defining language syntax and semantics.

This kind of need for modularity arises not only in metatheory [40, 41], but also in practical implementations. Hermans' Hedy language [80] is one of the few languages which actually internationalizes its syntax (and script [175]). Van der Storm and Hermans [183] describe how this works via a rich formalism for defining modifications or derivatives of context-free grammars which can be compiled into parsers yielding the same structures, even when parsing a localization of Hedy which may not only translate constants and keywords, but even change ordering of portions of syntactic constructs. They also describe grammar-level support for language levels (where certain constructs are withheld for students at varying points in their learning) akin to (though seemingly more flexible than) the language levels used in teaching Racket [52, 56]. Lexicalized grammars can do much of the same. We could switch our arithmetic example to prenex operator syntax by swapping the operator entries above for $+ \vdash (e/e)/e \Rightarrow \lambda l.\, \lambda r.\, l + r$. This entry differs from the earlier entry for + by looking for both arguments to its right. The first (outer, rightmost) e argument will be the left argument, and the inner e argument will be to the right *of that partial parse result*:

$$\dfrac{\dfrac{\dfrac{}{+ \vdash (e/e)/e \Rightarrow \lambda l.\, \lambda r.\, l + r} \quad \dfrac{}{3 \vdash e \Rightarrow 3}}{+ 3 \vdash e/e \Rightarrow \lambda r.\, 3 + r} \quad \dfrac{}{4 \vdash e \Rightarrow 4}}{+ 3\ 4 \vdash e \Rightarrow 7}$$

Something similar to (the syntactic aspects of) language levels can be done by partitioning lexical entries and including only entries from the desired range when parsing (similar to how van der Storm and Hermans handle productions from different levels). Localization (swapping in entries for different natural language keywords and syntactic rearrangements) can be handled by also controlling which lexical entries are included or excluded, e.g. swapping in a lexical entry for wahr rather than true when localizing to German.

Moreover, semantics are not *required* to be logical formulae or other structures with established mathematical denotations, but the results can in fact simply be other structures, such as more traditional parse trees. As a toy example, we could consider changing parsing order and symbols while still generating an arithmetic expression from an AST:

type Expr = Const Nat | Add Expr Expr | Sub Expr Expr | ...

We can replace the earlier lexical entries for + and ∗ by

$$+ \vdash (e \setminus e)/e \Rightarrow \lambda r.\, \lambda l.\, \text{Add l r}$$

$$* \vdash (e \setminus e)/e \Rightarrow \lambda r.\, \lambda l.\, \text{Times l r}$$

(and similarly for other operators) to generate ASTs rather than denotations, as in Figure 1. (This applies to the prenex version just as easily.)

Experienced language implementors might ask at this point what error messages would look like for such an approach. This is an excellent open question, whose solutions would have applications in both PL and SE research [66, 67] and linguistics.

7 Conclusions

We conceive of this essay as an invitation to PL and SE researchers to sample from the deep, broad reservoir of linguistics ideas. In this essay we have attempted to draw out a wide range of connections between problems addressed by software engineering and programming languages research in regard to programs (and text about programs), and questions addressed by linguistic research in regard to natural language in varied contexts. We have demonstrated a range of thematic parallels between these two sides suggesting some potential value, examples of linguistics results refining the context for results from PL and SE research, and touched on cases of likely productive transfer from linguistics into PL and SE research. Our examples, large and small, have drawn out the connections across many facets of PL and SE research, from human factors like interpretation of linguistic instructions and learning of programming languages, to sociological processes of language evolution, general methodological concerns, and specific technical devices. Our hope is that this encourages researchers in these areas to explore the wealth of ideas that linguists have been exploring for decades in parallel with our communities' work, for additional perspective if not direct gain. Linguistics as a field has insights close to virtually every corner and every style of research in programming languages and software engineering.

Acknowledgments

This work was supported in part by NSF Award CCF-2220991.

References

[1] Atef Odeh AbuSa'aleek. 2015. Internet linguistics: A linguistic analysis of electronic discourse as a new variety of language. *International journal of English linguistics* 5, 1 (2015), 135.

[2] Paul Adamczyk. 2011. On the language metaphor. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. 121–128.

[3] Michael D Adams and Matthew Might. 2017. Restricting grammars with tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–25.

[4] David Adger and Graeme Trousdale. 2007. Variation in English syntax: theoretical implications. *English Language & Linguistics* 11, 2 (2007), 261–278.

[5] Nobuhiko Akamatsu. 2003. The effects of first language orthographic features on second language reading in text. *Language learning* 53, 2 (2003), 207–231.

[6] Waad Alhoshan, Riza Batista-Navarro, and Liping Zhao. 2018. Towards a corpus of requirements documents enriched with semantic frame annotations. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 428–431.

[7] Nicholas Asher, Swarnadeep Bhar, Akshay Chaturvedi, Julie Hunter, and Soumya Paul. 2023. Limits for learning with language models. In *Proceedings of the 12th Joint Conference on Lexical and Computational Semantics (*SEM 2023)*, Alexis Palmer and Jose Camacho-collados (Eds.). Association for Computational Linguistics, Toronto, Canada, 236–248. https://doi.org/10.18653/v1/2023.starsem-1.22

[8] Ash Asudeh and Gianluca Giorgolo. 2020. *Enriched meanings: Natural language semantics with category theory*. Oxford Studies in Semantics and Pragmatics, Vol. 13. Oxford University Press.

[9] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94.

[10] Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian. 2016. How Android app developers manage power consumption? An empirical study by mining power management commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 37–48.

[11] Camilla Bardel and Ylva Falk. 2007. The role of the second language in third language acquisition: The case of Germanic syntax. *Second language research* 23, 4 (2007), 459–484.

[12] Chris Barker and Chung-chieh Shan. 2014. *Continuations and natural language*. Oxford Studies in Theoretical Linguistics, Vol. 53. Oxford University Press.

[13] John G Barnitz. 1982. Orthographies, bilingualism and learning to read English as a second language. *The Reading Teacher* 35, 5 (1982), 560–567.

[14] Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2024. Decidable Subtyping of Existential Types for Julia. *Proceedings of the ACM on Programming Languages* 8 (2024). Issue PLDI.

[15] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause'n'play: Formalizing asynchronous c. In *European Conference on Object-Oriented Programming*. Springer, 233–257.

[16] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa meets python: A boa dataset of data science software in python language. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 577–581.

[17] Andrew P Black, Kim B Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. 85–98.

[18] Barry Boehm, Hans Dieter Rombach, and Marvin V Zelkowitz. 2005. *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer Science & Business Media.

[19] Jeffrey Bonar and Elliot Soloway. 1983. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 10–13.

[20] Jürgen Börstler, Kwabena E Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, et al. 2023. Developers talking about code quality. *Empirical Software Engineering* 28, 6 (2023), 128.

[21] Joan Bresnan, Ronald M Kaplan, Stanley Peters, and Annie Zaenen. 1982. Cross-Serial Dependencies in Dutch. *Linguistic Inquiry* 13, 4 (1982), 613–635.

[22] Anne Brygoo, Totou Durand, Pascale Manoury, Christian Queinnec, and Michele Soria. 2002. Experiment around a training engine. *TelE-Learning: The Challenge for the Third Millennium* (2002), 45–52.

[23] A Brygoo, T Durand, P Manoury, C Queinnec, and M Soria. 2002. Un cédérom pour Scheme, Chacun son entraîneur, un entraîneur pour tous. In *Actes du colloque TICE*.

[24] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. 2006. Re-evaluating the role of BLEU in machine translation research. In *11th conference of the european chapter of the association for computational linguistics*. 249–256.

[25] Stan Carey. 2013. Singular they, you, and a 'senseless way of speaking'. https://stancarey.wordpress.com/2013/01/29/singular-they-you-and-a-senseless-way-of-speaking/

[26] Bob Carpenter. 1991. The generative power of categorial grammars and head-driven phrase structure grammars with lexical rules. *Computational linguistics* 17, 3 (1991), 301–314.

[27] Bob Carpenter. 1999. The Turing-completeness of multimodal categorial grammars. *JFAK: Essays dedicated to Johan van Benthem on the occasion of his 50th birthday. Institute for Logic, Language, and Computation, University of Amsterdam. Available on CD-ROM at http://turing. wins. uva. nl* (1999).

[28] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2016. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering* 43, 10 (2016), 954–974.

[29] Gary Charness, Edi Karni, and Dan Levin. 2010. On the conjunction fallacy in probability judgment: New experimental evidence regarding Linda. *Games and Economic Behavior* 68, 2 (2010), 551–556.

[30] Sarah E Chasins, Elena L Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (2021), 98–106.

[31] Preetha Chatterjee, Kostadin Damevski, Nicholas A Kraft, and Lori Pollock. 2020. Software-related slack chats with disentangled conversations. In *Proceedings of the 17th international conference on mining software repositories*. 588–592.

[32] Preetha Chatterjee, Kostadin Damevski, Nicholas A Kraft, and Lori Pollock. 2021. Automatically identifying the quality of developer chats for post hoc use. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–28.

[33] Preetha Chatterjee, Kostadin Damevski, and Lori Pollock. 2021. Automatic extraction of opinion-based Q&A from online developer chats. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1260–1272.

[34] Preetha Chatterjee, Kostadin Damevski, Lori Pollock, Vinay Augustine, and Nicholas A Kraft. 2019. Exploratory study of slack q&a chats as a mining source for software engineering tools. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 490–501.

[35] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. 2016.

An empirical study on the practice of maintaining object-relational mapping code in java systems. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 165–176.

[36] Thomas Cho. 2010. Linguistic features of electronic mail in the workplace: A comparison with memoranda. *Language@ internet* 7, 3 (2010).

[37] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory* 2, 3 (1956), 113–124.

[38] Jennifer Culbertson. 2023. Artificial language learning. In *The Oxford Handbook of Experimental Syntax*. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780198797722.013.9

[39] Rachele De Felice, Jeannique Darby, Anthony Fisher, and David Peplow. 2013. A classification scheme for annotating speech acts in a business email corpus. *Icame Journal* 37 (2013), 71–105.

[40] Benjamin Delaware, Bruno C d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 207–218.

[41] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno CdS Oliveira. 2013. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 319–330.

[42] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 75–80.

[43] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 208–212.

[44] Samanta Dey, Venkatesh Vinayakarao, Monika Gupta, and Sampath Dechu. 2022. Evaluating commit message generation: to BLEU or not to BLEU?. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 31–35.

[45] Eli Dresner and Susan C Herring. 2010. Functions of the nonverbal in CMC: Emoticons and illocutionary force. *Communication theory* 20, 3 (2010), 249–268.

[46] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*. 411–420.

[47] Jesse Egbert, Douglas Biber, and Bethany Gray. 2022. *Designing and evaluating language corpora: A practical framework for corpus representativeness*. Cambridge University Press.

[48] Osama Ehsan, Safwat Hassan, Mariam El Mezouar, and Ying Zou. 2020. An empirical study of developer discussions in the gitter platform. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–39.

[49] Micha Elsner and Eugene Charniak. 2010. Disentangling chat. *Computational Linguistics* 36, 3 (2010), 389–409.

[50] Allyson Ettinger. 2020. What BERT is not: Lessons from a new suite of psycholinguistic diagnostics for language models. *Transactions of the Association for Computational Linguistics* 8 (2020), 34–48.

[51] Dror G Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2020. How developers choose names. *IEEE Transactions on Software Engineering* 48, 1 (2020), 37–52.

[52] Mattias Felleisen. 1998. The DrScheme project: an overview. *ACM Sigplan Notices* 33, 6 (1998), 17–23.

[53] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press.

[54] Andrzej Filinski. 1994. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 446–457.

[55] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *Journal of functional programming* 12, 2 (2002), 159–182.

[56] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP'97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, September 3–5, 1997 Proceedings 9*. Springer, 369–388.

[57] John R. Firth. 1957. Modes of meaning. In *Papers in Linguistics, 1934–1951*. Oxford University Press.

[58] Scott D Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 1–41.

[59] Suzanne Flynn, Claire Foley, and Inna Vinnitskaya. 2004. The cumulative-enhancement model for language acquisition: Comparing adults' and children's patterns of development in first, second and third language acquisition of relative clauses. *International journal of multilingualism* 1, 1 (2004), 3–16.

[60] Angela Cora Garcia and Jennifer Baker Jacobs. 1999. The eyes of the beholder: Understanding the turn-taking system in quasi-synchronous computer-mediated communication. *Research on language and social interaction* 32, 4 (1999), 337–367.

[61] Susan M Gass and Larry Selinker. 1992. *Language transfer in language learning: Revised edition*. Vol. 5. John Benjamins Publishing.

[62] Jeremy Gibbons. 2017. On "Methodological Irregularities in Programming Language Research". (2017). https://www.cs.ox.ac.uk/jeremy.gibbons/publications/methodological.pdf

[63] Gerd Gigerenzer. 1996. On Narrow Norms and Vague Heuristics: A Reply to Kahneman and Tversky (1996). *Psychological Review* 103, 3 (1996), 592–596.

[64] Grant Goodall. 2023. Constructed languages. *Annual Review of Linguistics* 9, 1 (2023), 419–437.

[65] Adam Goodkind and Klinton Bicknell. 2018. Predictive power of word surprisal for reading times is a linear function of language model quality. In *Proceedings of the 8th workshop on cognitive modeling and computational linguistics (CMCL 2018)*. 10–18.

[66] Colin S Gordon. 2022. Towards property-based tests in natural language. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 111–115.

[67] Colin S Gordon and Sergey Matskevich. 2023. Trustworthy Formal Natural Language Specifications. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 50–70.

[68] Stanisław Goźdź-Roszkowski. 2021. Corpus linguistics in legal discourse. *International Journal for the Semiotics of Law-Revue internationale de Sémiotique juridique* 34, 5 (2021), 1515–1540.

[69] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2022. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming* 7, 2 (2022).

[70] Herbert P Grice. 1975. Logic and conversation. In *Speech acts*. Brill, 41–58.

[71] Julian Grove and Jean-Philippe Bernardy. 2022. Algebraic effects for extensible dynamic semantics. *Journal of Logic, Language and Information* (2022), 1–27.

[72] Philip J Guo. 2018. Non-native english speakers learning computer programming: Barriers , desires, and design opportunities. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–14.

[73] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie Van Deursen. 2013. Communication in open source software development mailing lists. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 277–286.

[74] Kevin Hammond and Stephen Blott. 1990. Implementing Haskell type classes. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop 21–23 August 1989, Fraserburgh, Scotland*. Springer, 265–286.

[75] Michael Handford. 2017. Corpus linguistics. In *The Routledge handbook of language in the workplace*. Routledge, 51–64.

[76] Stefan Hanenberg. 2010. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 22–35.

[77] Stefan Hanenberg. 2010. Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 933–946.

[78] Randy Harris. 1994. Review of The Language Instinct. *The Globe & Mail* (18 June 1994). https://arts.uwaterloo.ca/~raha/reviews/Harris-Pinker.pdf

[79] Randy Allen Harris. 2021. *The linguistics wars: Chomsky, Lakoff, and the battle over deep structure*. Oxford University Press.

[80] Felienne Hermans. 2020. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*. 259–270.

[81] Felienne Hermans. 2023. Creating a learnable and inclusive programming language. Keynote presentation at Onward! 2023.

[82] Susan Herring. 1999. Interactional Coherence in CMC. *Journal of Computer-Mediated Communication* 4, 4 (1999).

[83] Susan C Herring. 1996. Computer-mediated communication. *Computer-Mediated Communication* (1996), 1–332.

[84] Susan C Herring. 2004. Computer-mediated discourse analysis: An approach to researching online behavior. *Designing for virtual communities in the service of learning* 338 (2004), 376.

[85] Raymond Hickey. 2020. *The handbook of language contact*. John Wiley & Sons.

[86] James Higginbotham. 1984. English Is Not a Context-Free Language. *Linguistic Inquiry* 15, 2 (1984), 225–234.

[87] Martin Hilpert and Stefan Th Gries. 2016. Quantitative approaches to diachronic corpus linguistics. *The Cambridge handbook of English historical linguistics* (2016), 36–53.

[88] Janet Holmes and Maria Stubbe. 2015. *Power and politeness in the workplace: A sociolinguistic analysis of talk at work*. Routledge.

[89] Yan Huang. 2017. *The Oxford handbook of pragmatics*. Oxford University Press.

[90] James R Hurford. 1974. Exclusive or inclusive disjunction. *Foundations of language* 11, 3 (1974), 409–411.

[91] Pauline Jacobson. 1999. Towards a variable-free semantics. *Linguistics and philosophy* (1999), 117–184.

[92] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible Metatheory Mechanization via Family Polymorphism. *Proc. of the ACM on Programming Languages (PACMPL)* 7 (2023).

[93] Arvind K. Joshi, K. Vijay Shanker, and David Weir. 1991. The convergence of mildly context-sensitive grammatical formalisms. *Foundations issues in natural language processing* (1991), 31–81.

[94] Laura Kallmeyer. 2010. *Parsing beyond context-free grammars*. Springer Science & Business Media.

[95] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. 2013. Type refinement for static analysis of JavaScript. In *Proceedings of the 9th symposium on Dynamic languages*. 17–26.

[96] Sumitra Mangesh Katre et al. 1989. *Aṣṭādhyāyī of Pāṇini*. Motilal Banarsidass Publ.

[97] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM computing surveys (CSUR)* 37, 2 (2005), 83–137.

[98] Eric Kellerman. 1977. Towards a characterisation of the strategy of transfer in second language learning. *Interlanguage studies bulletin* (1977), 58–145.

[99] Steven Keuchel and Tom Schrijvers. 2013. Generic datatypes à la carte. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*. 13–24.

[100] Amy J Ko and Brad A Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*. 301–310.

[101] Amy J Ko, Brad A Myers, and Duen Horng Chau. 2006. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 127–134.

[102] Jordan Kodner, Spencer Caplan, and Charles Yang. 2022. Another model not for the learning of language. *Proceedings of the National Academy of Sciences* 119, 29 (2022), e2204664119.

[103] Charles W Kreidler. 1979. Creating new words by shortening. *Journal of English Linguistics* 13, 1 (1979), 24–36.

[104] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond.

[105] Henry Kučera and Winthrop Nelson Francis. 1967. *Computational analysis of present-day American English*. Brown university press.

[106] Marco Kuhlmann, Alexander Koller, and Giorgio Satta. 2015. Lexicalization and generative power in CCG. *Computational Linguistics* 41, 2 (2015), 215–247.

[107] Marco Kuhlmann, Giorgio Satta, and Peter Jonsson. 2018. On the complexity of CCG parsing. *Computational Linguistics* 44, 3 (2018), 447–482.

[108] Barry L Kurtz. 1980. Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*. 110–117.

[109] Joachim Lambek. 1958. The mathematics of sentence structure. *The American Mathematical Monthly* 65, 3 (1958), 154–170.

[110] Joachim Lambek. 1988. Categorial and categorical grammars. In *Categorial grammars and natural language structures*. Springer, 297–317.

[111] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. 2007. Scents in programs: Does information foraging theory apply to program maintenance?. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 15–22.

[112] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. 2010. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2010), 197–215.

[113] Jacqueline Léon. 2005. Meaning by collocation. *History of linguistics* (2005), 404–415.

[114] Matthew C Loring, Mark Marron, and Daan Leijen. 2017. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. 51–62.

[115] Kuang-Chen Lu, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. 2023. What Happens When Students Switch (Functional) Languages (Experience Report). *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 796–812.

[116] John H Mabry. 1995. Review of Pinker's the language instinct. *The Analysis of verbal behavior* 12 (1995), 87.

[117] Daniel Marino and Todd Millstein. 2009. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in*

language design and implementation. 39–50.

[118] Karoline Marko. 2022. "Depends on Who I'm Writing To"—The Influence of Addressees and Personality Traits on the Use of Emoji and Emoticons, and Related Implications for Forensic Authorship Analysis. *Frontiers in Communication* 7 (2022), 38.

[119] Jirka Maršík and Maxime Amblard. 2016. Introducing a calculus of effects and handlers for natural language semantics. In *Formal Grammar: 20th and 21st International Conferences, FG 2015, Barcelona, Spain, August 2015, Revised Selected Papers. FG 2016, Bozen, Italy, August 2016, Proceedings 21*. Springer, 257–272.

[120] Tony McEnery and Andrew Hardie. 2013. The history of corpus linguistics. In *The Oxford Handbook of the History of Linguistics*, Keith Allan (Ed.).

[121] Wayne S Messer and Richard A Griggs. 1993. Another look at Linda. *Bulletin of the Psychonomic Society* 31, 3 (1993), 193–196.

[122] Mark S Miller, E Dean Tribble, and Jonathan Shapiro. 2005. Concurrency among strangers: Programming in E as plan coordination. In *Trustworthy Global Computing: International Symposium, TGC 2005 , Edinburgh, UK, April 7-9, 2005. Revised Selected Papers*. Springer, 195–229.

[123] Richard Montague. 1970. English as a Formal Language. In *Linguaggi nella societa e nella tecnica*, Bruno Visentini (Ed.). Edizioni di Communita, 188–221.

[124] Richard Montague. 1973. The proper treatment of quantification in ordinary English. In *Approaches to natural language: Proceedings of the 1970 Stanford workshop on grammar and semantics*. Springer, 221–242.

[125] Hans-Georg Müller and Christoph Schroeder. 2024. On the influence of the first language on orthographic competences in German as a second language: A comparative analysis. *Applied Linguistics Review* 15, 2 (2024), 449–473.

[126] Denise E Murray. 1985. Composition as conversation: The computer terminal as medium of communication. *Writing in nonacademic settings* (1985), 203–227.

[127] Denise E Murray. 1988. The context of oral and written language: A framework for mode and medium switching1. *Language in society* 17, 3 (1988), 351–373.

[128] Frederick J Newmeyer. 2022. *American linguistics in transition: from post-Bloomfieldian structuralism to generative grammar*. Oxford University Press.

[129] Jeremy W Nimmer and Michael D Ernst. 2002. Invariant inference for static checking: An empirical evaluation. *ACM SIGSOFT Software Engineering Notes* 27, 6 (2002), 11–20.

[130] James Noble and Robert Biddle. 2023. programmingLanguage as Language. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 191–204.

[131] James Noble, Michael Homer, Kim B Bruce, and Andrew P Black. 2013. Designing grace: Can an introductory programming language support the teaching of software engineering?. In *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 219–228.

[132] Bruno C d S Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. ACM, 341–360.

[133] Inc. Oracle. 2021. Java Pattern Matching. https://docs.oracle.com/en/java/javase/17/language/pattern-matching.html

[134] Lalchand Pandia and Allyson Ettinger. 2021. Sorting through the noise: Testing robustness of information processing in pre-trained language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 1583–1596. https://aclanthology.org/2021.emnlp-main.119

[135] 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.

[136] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Software Engineering* 18, 6 (2013), 1047–1089.

[137] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.

[138] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2020. A look into programmers' heads. *IEEE Transactions on Software Engineering* 46, 4 (2020), 442–462.

[139] P Stanley Peters Jr and Robert W Ritchie. 1973. On the generative power of transformational grammars. *Information sciences* 6 (1973), 49–83.

[140] Kelly Peterson, Matt Hohensee, and Fei Xia. 2011. Email formality in the workplace: A case study on the Enron corpus. In *Proceedings of the Workshop on Language in Social Media (LSM 2011)*. 86–95.

[141] Amy E Pierce. 1992. *Language Acquisition and Syntactic Theory: A Comparative Analysis of French and English Child Grammars*. Springer.

[142] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

[143] Shana Poplack and David Sankoff. 1984. Borrowing: the synchrony of integration. (1984).

[144] Christopher Potts. 2015. Presupposition and implicature. *The handbook of contemporary semantic theory* (2015), 168–202.

[145] Chantel S Prat, Tara M Madhyastha, Malayka J Mottarella, and Chu-Hsuan Kuo. 2020. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports* 10, 1 (2020), 3817.

[146] Eloi Puig-Mayenco, Jason Rothman, and Susagna Tubau. 2022. Language dominance in the previously acquired languages modulates rate of third language (L3) development over time: A longitudinal study. *International Journal of Bilingual Education and Bilingualism* 25, 5 (2022), 1641–1664.

[147] Zhuang Qiu, Xufeng Duan, and Zhenguang Cai. 2023. Does ChatGPT Resemble Humans in Processing Implicatures?. In *Proceedings of the 4th Natural Logic Meets Machine Learning Workshop*. 25–34.

[148] Christian Queinnec and Pierre Weis. 1996. Programmation applicative, état des lieux et perspectives. *Technique et science informatiques* 15 (1996). Issue 7.

[149] Abderahman Rashwan, Olga Ormandjieva, and Rene Witte. 2013. Ontology-based classification of non-functional requirements in software specifications: A new corpus and SVM-based classifier. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE, 381–386.

[150] Stephanie A Robertson and Martin P Lee. 1995. The application of second natural language acquisition pedagogy to the teaching of programming languages—a research agenda. *ACM SIGCSE Bulletin* 27, 4 (1995), 9–12.

[151] Jason Rothman. 2015. Linguistic and cognitive motivations for the Typological Primacy Model (TPM) of third language (L3) transfer: Timing of acquisition and proficiency considered. *Bilingualism: language and cognition* 18, 2 (2015), 179–190.

[152] Geoffrey Sampson. 2007. There is no language instinct. *Ilha do Desterro: A Journal of English Language, Literatures in English and Cultural Studies* 52 (2007), 35–63.

[153] Gillian Sankoff. 2004. Linguistic outcomes of language contact. *The handbook of language variation and change* (2004), 638–668.

[154] Vicki L Sauter. 1986. Predicting computer programming skill. *Computers & Education* 10, 2 (1986), 299–302.

[155] Tatjana Scheffler, Lasse Brandt, Marie de la Fuente, and Ivan Nenchev. 2022. The processing of emoji-word substitutions: A self-paced-reading study. *Computers in Human Behavior* 127 (2022), 107076.

[156] Karl M Schimpf and Jean H Gallier. 1985. Tree pushdown automata. *J. Comput. System Sci.* 30, 1 (1985), 25–40.

[157] Jean Scholtz and Susan Wiedenbeck. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction* 2, 1 (1990), 51–72.

[158] Jean Scholtz and Susan Wiedenbeck. 1991. Learning a new programming language: a model of the planning process. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Vol. 2. IEEE, 3–12.

[159] Jean Scholtz and Susan Wiedenbeck. 1992. An Analysis of Novice Programmers Leaming a Second Language. In *Empirical Studies of Programmers: Fifth Workshop (PPIG 1992)*. 187–205.

[160] Bonnie D Schwartz and Rex A Sprouse. 1996. L2 cognitive states and the full transfer/full access model. *Second language research* 12, 1 (1996), 40–72.

[161] Sandro Sciutti. 2020. The acquisition of clitic pronouns in complex infinitival clauses by German-speaking learners of Italian as an L3: The role of proficiency in target and background language (s). In *Third language acquisition: Age, proficiency and multilingualism*. Language Sciences Press.

[162] Hiroyuki Seki, Tadao Kasami, Eiji Nabika, and Takashi Matsumura. 1992. A method for translating natural language program specifications into algebraic specifications. *Systems and computers in Japan* 23, 11 (1992), 1–16.

[163] Hiroyuki Seki, Eiji Nabika, Takashi Matsumura, Yujii Sugiyama, Mamoru Fujii, Koji Torii, and Tadao Kasami. 1988. A processing system for programming specifications in a natural language. In *[1988] Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track*, Vol. 2. IEEE, 754–763.

[164] Peter Sells, Stuart Merrill Shieber, and Thomas Wasow. 1991. *Foundational issues in natural language processing*. MIT Press.

[165] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[166] Stuart M Shieber. 1985. Evidence against the context-freeness of natural language. In *The Formal complexity of natural language*. Springer, 320–334.

[167] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering*. 378–389.

[168] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. 2020. Studying programming in the neuroage: just a crazy idea? *Commun. ACM* 63, 6 (2020), 30–34.

[169] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[170] Anatol Stefanowitsch and Stefan Th Gries. 2003. Collostructions: Investigating the interaction of words and constructions. *International journal of corpus linguistics* 8, 2 (2003), 209–243.

[171] Andreas Stefik and Stefan Hanenberg. 2017. Methodological irregularities in programming-language research. *Computer* 50, 8 (2017), 60–63.

[172] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40.

[173] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*. Ieee, 83–92.

[174] Pia Sundqvist. 2024. Extramural English as an individual difference variable in L2 research: Methodology matters. *Annual Review of Applied Linguistics* (2024), 1–13.

[175] Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. 13–25.

[176] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 175–189.

[177] Stefan Th. Gries. 2021. Analyzing dispersion. In *A practical handbook of corpus linguistics*. Springer, 99–118.

[178] Michael Tomasello. 1995. Language Is Not An Instinct. *Cognitive Development* 10 (1995), 131–156.

[179] Aaron Traylor, Roman Feiman, and Ellie Pavlick. 2021. AND does not mean OR: Using Formal Languages to Study Language Models' Representations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*.

[180] Ethel Tshukudu and Quintin Cutts. 2020. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM conference on international computing education research*. 227–237.

[181] Amos Tversky and Daniel Kahneman. 1983. Extensional versus intuitive reasoning: The conjunction fallacy in probability judgment. *Psychological review* 90, 4 (1983), 293.

[182] Anne Vainikka and Martha Young-Scholten. 1996. Gradual development of L2 phrase structure. *Second language research* 12, 1 (1996), 7–39.

[183] Tijs van der Storm and Felienne Hermans. 2022. Gradual Grammars: Syntax in Levels and Locales. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 134–147. https://doi.org/10.1145/3567512.3567524

[184] Guido H Wachsmuth, Gabriël DP Konat, and Eelco Visser. 2014. Language design with the spoofax language workbench. *IEEE software* 31, 5 (2014), 35–43.

[185] Alex Warstadt and Samuel R Bowman. 2022. What artificial neural networks can tell us about human language acquisition. In *Algebraic structures in natural language*. CRC Press, 17–60.

[186] Marit Westergaard, Natalia Mitrofanova, Roksolana Mykhaylyk, and Yulia Rodina. 2017. Crosslinguistic influence in the acquisition of a third language: The Linguistic Proximity Model. *International Journal of Bilingualism* 21, 6 (2017), 666–682.

[187] Andrew K Wright. 1995. Simple imperative polymorphism. *Lisp and symbolic computation* 8, 4 (1995), 343–355.

[188] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 696–708.

[189] Zejun Zhang, Zhenchang Xing, Xiwei Xu, and Liming Zhu. 2023. Ridiom: Automatically refactoring non-idiomatic Python code with pythonic idioms. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 102–106.

[190] Zejun Zhang, Zhenchang Xing, Dehai Zhao, Qinghua Lu, Xiwei Xu, and Liming Zhu. 2024. Hard to Read and Understand Pythonic Idioms? DeIdiom and Explain Them in Non-Idiomatic Equivalent Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1011–1011.

[191] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.

Received 2024-04-25; accepted 2024-08-08