Towards Property-Based Tests in Natural Language

Colin S. Gordon
Drexel University
Philadelphia, Pennsylvania, USA
csgordon@drexel.edu

ABSTRACT

We consider a new approach to generate tests from natural language. Rather than relying on machine learning or templated extraction from structured comments, we propose to apply classic ideas from linguistics to translate natural-language sentences into executable tests. This paper explores the application of *combinatory categorial grammars* (CCGs) to generating property-based tests. Our prototype is able to generate tests from English descriptions for each example in a textbook chapter on property-based testing.

ACM Reference Format:
Colin S. Gordon. 2022. Towards Property-Based Tests in Natural Language . In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510455.3512781

1 INTRODUCTION

The ability to specify functional and unit tests using natural language has applications to documenting test cases, tracing requirements to tests, reducing discrepancies between prose functionality descriptions and the properties encoded in test suites, and helping less technical clients build confidence that software is implemented consistently with their understanding of requirements. It is common to see test suites written using domain-specific languages (DSLs) modeled to look like fragments of English [46]; testing frameworks in the "Given-When-Then" behavior-driven development style [39] using specification languages like Gherkin [63] which allow developers to map developer-chosen fixed natural language to sections of test cases; RSpec-style [17] libraries that have natural language description be embedded in tests; or ad hoc practices in languages like F#, whose support for (quoted) identifiers containing strings is used in practice to name test cases using natural language descriptions. Prior research has explored variations on these ideas, which can be very effective (some are widely used). But many associate code and descriptions only by convention (F#, RSpec) or manual processes [55], or are limited by a host language (DSLs). Others rely on brittle heuristics to connect language and code, such as: Gherkin; recognizing hard-coded phrases in specific Javadoc clauses [15, 25, 59]; assuming a restrictive subject-verb-direct-object sentence structure that breaks on many natural sentences [35]; or using brittle keyword-based heuristics to extract tests for a single testing template [8]. Ahsan et al. [3] survey similar techniques. Sharma and Biswas [56] propose generating tests from a restricted logical representation of requirements that reads similarly to a fragment of English. While these approaches have notable successes (Motwani and Brun [51] generate tests from the official JavaScript specification, exploiting its unusually consistent wording to extract test elements via regular expressions), these are all limited by the lack of a true linguistic model, causing them to fail on novel sentence structures (natural languages are non-context-free [34]) and making some extensions non-compositional.

While not yet applied to test generation from text, machine-learning (especially deep learning) is increasingly used to associate code and text [4]. These approaches make good use of large corpora of public software source code to learn rich associations between text and code. However, a common critique of these techniques is that they are non-modular, making them brittle and difficult to repair. Adding an individual word to a neural network-based system [31, 40–42] that was absent in training data is not feasible, nor is correcting handling of individual existing words The only solution is to to retrain the network.

Classic work in linguistics on compositional models of grammar and sentence meaning offers another, principled way to relate code and natural language, in a way that generalizes to unseen inputs and naturally supports modular extension and bug fixes. We show that it is possible to use a classic linguistic semantic technique to generate property-based tests from natural language. Beyond its direct desirable properties, this approach opens the door to drawing on decades of knowledge from the linguistics community to more effectively generate tests from natural language.

2 CATEGORIAL GRAMMAR, SEMANTIC PARSING, AND PROPERTY-BASED TESTS

Linguists have long studied precise models of grammar and meaning, producing rich literature on computing natural language text's semantic meaning from a grammatical parse tree — the parse guides how the logical meanings of individual words are combined to compute the meaning of a whole sentence [12]. *Combinatory Categorial Grammars* (CCGs) [11, 58] are one established body of techniques to model both parsing and translation into a *logical form* representing sentence meaning, a task known as *semantic parsing*. CCGs are theoretically powerful (mildly-context-sensitive [36, 61]) and have been shown to capture general accounts of subtle linguistic phenomena in a number of natural languages, well enough to parse large corpora of English [30], German [28], Hindi [5], and Japanese [45]. CCGs can model meaning using any lambda calculus with a boolean-like type (technically, a Heyting Algebra) [38], so can be used to generate meanings in many logics or programming languages.

Property-based tests (PBTs) [19] are a form of guided random testing, focusing on properties that should hold for *classes* of inputs rather than individual inputs. Inputs are produced by combining

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

$$\dfrac{\dfrac{}{3 \vdash NP \Rightarrow 3}\,\text{LEX} \qquad \dfrac{\dfrac{}{\text{is} \vdash (S \setminus NP)/ADJ \Rightarrow (\lambda p.\, \lambda n.\, p\ n)}\,\text{LEX} \qquad \dfrac{}{\text{even} \vdash ADJ \Rightarrow (\lambda n.\, n\%2 = 0)}\,\text{LEX}}{\text{is even} \vdash (S \setminus NP) \Rightarrow (\lambda n.\, n\%2 = 0)}\,>}{3\ \text{is even} \vdash S \Rightarrow 3\%2 = 0}\,<$$

Figure 1: A derivation translating "3 is even" to the proposition $3\%2 = 0$.

and filtering primitive random input generators, and transforming results of other generators. PBTs generally take the form of a universal property `forall(g,f)`, where `g` is a generator and `f` is a function which either returns a boolean or performs assertions to check a property of any input produced by `g`. When run, the test draws many random inputs from `g`, calling `f` on each, and fails if `f` ever returns false or fails an assertion. Typical PBT implementations make heavy use of anonymous functions, and represent properties with a datatype which carries operations corresponding to conjunction, disjunction, negation, and implication of properties. These datatypes (representing logical claims) are then adequate to serve as a target semantics for CCG-based semantic parsing, meaning CCGs can be used to translate natural language to property-based tests.[1] The rest of this section outlines how this can work.

Categorial grammars annotate each word with a set of grammatical *categories* describing how it combines with other words and clauses. The supported categories include both primitive categories, and categories modeling words whose semantics take arguments. For English the base categories typically include sentences (S), noun phrases (NP), and adjectives (ADJ), among others. The (language-independent) non-primitive categories are built from left and right *slash types*. A left slash type $A \setminus B$ is the category of sentence fragments whose composition with a fragment of type B on its left result in a larger fragment of grammatical type A. A right slash type A/B expects the B to the right.[2] These intuitions, and other means of combining grammar sentence fragments, are captured by inference rules specifying how adjacent sentence fragments interact. For example:

$$\textsc{Right-Application}\ (>) \quad \dfrac{\Gamma \vdash X/Y \Rightarrow f \qquad \Delta \vdash Y \Rightarrow a}{\Gamma, \Delta \vdash X \Rightarrow f\ a} \qquad \textsc{Left-Application}\ (<) \quad \dfrac{\Gamma \vdash Y \Rightarrow a \qquad \Delta \vdash X \setminus Y \Rightarrow f}{\Gamma, \Delta \vdash X \Rightarrow f\ a}$$

Here Γ and Δ are non-empty sequences of words, X and Y (and the slash types) are grammatical types, and f and a are the semantics (classically, logical denotation) of the individual fragments. In the first rule, Γ is a sentence fragment that is nearly of grammatical type X, if it only had a Y to the right, so its logical form is a function f. Applying that function to the semantics of Δ (whose type is the needed Y) yields semantics for a grammatical phrase of type X. The second rule is symmetric. These rules plus assumptions about individual words is enough to build a kind of logical derivation that parses a simple sentence into its meaning as in Figure 1. There, "is" combines first with its right argument "even" (an adjective ADJ), yielding sensible semantics for verb phrase ($S \setminus NP$) "_ is even": $(\lambda n.\, n\%2 = 0)$. Then the result combines on the left with "3" (a noun phrase NP), completing the sentence. Note that it must be possible to formulate false claims (failing tests).

[1] In principle CCGs could be used for regular test assertions as well, but we focus on PBTs because they specify complete tests.

[2] In both cases the result category is on the left, the argument is on the right, and the top of the intervening slash "leans" in the direction of the argument.

Assumptions about individual words form a *lexicon*: a set of grammatical categories and semantics for each word. A word with multiple meanings may have multiple lexicon entries. CCGs isolate knowledge for specific natural languages to this lexicon, reusing the core rules across any natural language. *Wide-coverage CCG lexicons* capable of correctly parsing large text corpora exist for English [30], Hindi [5], German [28], and Japanese [45]. The natural modular structure of lexicons means that these existing lexicons can be directly extended with domain-specific terminology.

For a more complex example than Figure 1, consider a PBT for "every even integer is divisible by 2." This quantifies over all *even* integers from a generator, which is typically expressed in code by applying a filter operation to a generator. Our lexicon entry for the quantifier "every" captures this:

every $\vdash ((S/(S \setminus NP))/CN[Gen])/ADJ \Rightarrow$
`P=>gen=>claim=>fc.property(gen.filter(P),claim)`

This defines "every" as a word which, given an adjective (P), a common noun (*gen*) of a particular flavor (see Section 4), and a sentence fragment (*claim*) that is intuitively "missing its subject" (and looking for a suitable subject noun phrase to its left), yields a sentence. The semantics of that sentence (given using Javascript anonymous functions and a specific PBT library) is a property test that asserts that *claim* holds for all inputs produced by the generator *gen* corresponding to that common noun, for which the adjective P is accurate (implemented by filtering the generator).

CCGs include, *and our prototype uses*, 6 additional rules to extend coverage to linguistic constructs like long-distance dependencies [47, 48], unlike-coordination [13, 16], and cross-serial dependencies [57] (which are non-context-free); and extended slash types [11, 58] that capture how individual words restrict reorderings of other phrases (e.g., island constraints [24]). The established use of CCGs to analyze many subtle linguistic constructions [32, 58] in many languages [5, 28, 30, 45], and their demonstrated utility in parsing corpora like the *Wall Street Journal* [30] and *Alice in Wonderland* [64] strongly suggest that in contrast to popular heuristic approaches to generating tests from text (mentioned in the introduction), CCGs will impose no fundamental restrictions on the language used in test specifications — all restrictions on language used for tests in our approach would stem from the lexicon used, which as discussed can be modularly extended or improved over time.

3 PROTOTYPE IMPLEMENTATION

We have used NLTK [14] to implement a publicly-available prototype [26] that translates English to property-based tests in Javascript using the `fast-check` PBT library [1]. NLTK includes an implementation of semantic parsing using CCGs: given a set of base categories and lexicon entries, the library can construct a chart parser [20–22] for the specified lexicon. NLTK's CCG support assumes a simply typed lambda calculus for semantics; we generate semantics in that form with `fast-check` identifiers, and rewrite the

Table 1: STTP test specifications with variants handled by our prototype.

#	Original and Modified Test Specification, (Beta-Reduced) Logical Representation
1	For all floats, ranging from 1 (inclusive) to 5.0 (exclusive), the program should return false ⇝ Any float greater than or equal to 1 and less than 5 is not passing ↪ `foreach(filter(floats,\x.((lessthan(1,x) | equals(1,x)) & lessthan(x,5))),\n.-passing(n))`
2	For all floats, ranging from 5 (inclusive) to 10 (inclusive), the program should return true ⇝ Any float greater than or equal to 5 and less than or equal to 10 is passing ↪ `foreach(filter(floats,\x.(lessthanoreq(5,x) & lessthanoreq(x,10))),\n.passing(n))`
3	For all invalid grades (which we define as any number below 0.9 or greater than 10.1), the program must throw an exception ↪ For any float less than 1 or greater than 10 passing throws an exception ↪ `foreach(filter(floats,\x.(lessthan(x,1) | lessthan(10,x))),\x.checkthrows(isexc,\u.passing(x)))`
4	For all numbers divisible by 3, and not divisible by 5, the program returns "Fizz" ⇝ For any number divisible by 3 and not divisible by 5 fizzbuzz returns "Fizz" ↪ ✗`foreach(filter(integers,\x.(divisibleby(x,5) & -divisibleby(x,3))),\x.(fizzbuzz(x)=Fizz))`
5	For all numbers divisible by 5 (and not divisible by 3), the program returns "Buzz." ⇝ For any number divisible by 5 and not divisible by 3 fizzbuzz returns "Buzz" ↪ ✗`foreach(filter(integers,\x.(divisibleby(x,5) & -divisibleby(x,3))),\x.(fizzbuzz(x)=Buzz))`
6	For all numbers divisible by 3 and 5, the program returns "FizzBuzz".⇝ For any number divisible by 5 and 3 fizzbuzz returns "FizzBuzz" ↪ ✗`foreach(filter(integers,(\x.divisibleby(x,5) & divisibleby(x,3))),(\z.fizzbuzz(z)==FizzBuzz))`
7	The program throws an exception for all numbers that are zero or smaller.⇝ For any number less than or equal to zero fizzbuzz throws an exception ↪ `foreach(filter(integers,\x.(lessthan(x,0) | equals(0,x))),\x.checkthrows(isexc,\u.fizzbuzz(x)))`

syntax into JavaScript syntax after the fact. The prototype is primarily the lexicon itself. We constructed the lexicon by hand-writing entries for words present in our evaluation study based on a combination of examining the samples in our corpus, prior experience describing property-based tests (e.g., in teaching), and background knowledge about the treatment of certain English-language grammatical phenomena in CCGs and related grammar formalisms, such as quantification [32, 58], and coordination ("and" and "or") [16, 50].

This initially manual authoring process highlights some strengths relative to the increasingly popular use of deep neural networks for similar tasks [4]. As noted earlier, neural techniques are non-modular and do not permit fixes for individual words. By contrast, lexicons, while intricate, are inherently modular. Several times while working on the evaluation below, we wrote incorrect semantics or incorrect grammatical types for several words. Fixing them was a simple matter of correcting individual entries, which had no effect on examples that already worked but did not involve the word in question. The modular construction extends to the addition of new words. Using CCGs, adding basic negation support is a simple matter of adding one lexicon entry: not $\vdash ADJ/ADJ \Rightarrow$ `P=>x=>!P(x)`. More complete support for negation in natural language requires additional lexicon entries, but here again CCGs offer advantages over neural networks: detailed CCG treatments of negations in natural language already exist [58], while neural networks still struggle to handle language surrounding boolean operations [23, 60].

We expect that in practice, adding support for new properties and data types (e.g., packaging extensions for a specific library or program) will require CCG's strong support for modularity. In the future we plan to adapt techniques for *learning* CCG lexicons [10, 37] to extend an existing English lexicon [30], but those results will retain the ability to individually extend or correct individual words.

4 EXPERIENCE WITH THE PROTOTYPE

To evaluate basic feasibility of our approach, we collected complete sentences describing properties paired with corresponding PBTs, from books teaching PBT. To avoid biasing results towards what we thought would be easier to support, we sought cases where an existing text had a pairing of a PBT with a sentence describing what the test checked. We surveyed 6 books covering property-based testing [6, 18, 27, 43, 52, 62], locating 26 such pairs. We translated 7 descriptions from Aniche's *Software Testing: From Theory to Practice* (STTP) [6] into Javascript `fast-check` tests, with mild rephrasing to replace use of "the program" with the specific function being tested and to work around limitations of the NLTK frontend (lack of support for punctuation and numerals). We then studied the 19 other descriptions looking for signs of additional linguistic complexity beyond what is treated in existing texts on categorial grammars for English [16, 33, 49, 58].

Basic Feasibility. Table 1 shows the original text, rephrasings, and computed λ-calculus logical forms for the 7 STTP tests, which the prototype converts to Javascript.

Adequacy of Generated Tests. Each generated test correctly formalizes the corresponding text. 3 of the STTP exemplars (marked with ✗) fail when run on a correct implementation because they were under-specified. Test #7 requires an exception for negative numbers and 0. But these can also be divisible by 3 or 5, conflicting with tests #4–6. The textbook's solutions filter generators to produce only inputs are greater than or equal to 1 for those tests. By adding a lexicon entry for "positive" we can generate tests equivalent to the solutions (using "positive numbers"). This kind of detail is both the sort of detail sometimes omitted intentionally in informal prose, but also the sort of detail often omitted *unintentionally* when under-specifying. Directly connecting the English to the tests revealed the English was imprecise about expected behavior — exactly the sort of divergence between natural language specifications and tests we sought to identify.

Natural language often admits multiple parses for the same sentence, which occasionally yields different logical forms. The sentences in Table 1 yielded respectively 20, 4, 6, 45, 45, 3, and 1 distinct

parse trees, but all parses of each sentence produced equivalent logical forms (e.g., differing only in the names of bound variables).

Lexicon Size. Parsing these examples containing 29 words and numbers requires 46 lexicon entries. We believe this is reasonable. Only 3 rules are specific to the subjects under test: the entries for "passing" and "fizzbuzz" (there are two entries for "passing", usable as a function or an adjective). 7 rules are working around limitations of NLTK, which does not have generic handling of numerals (4) of string literals (3). A more robust implementation would have generic handling of numeric constants and string literals. Thus there are 36 reusable lexicon entries for 19 reusable words. We would expect lexicon size to be roughly linear in the number of words handled, and clearly 19 entries is a lower bound for 19 words. Duplicates handle words that may be used in multiple grammatical roles with similar semantics.

Linguistic Flexibility. Even these 7 tests involve a range of subtle and commonly-occurring linguistic phenomena, such as overloading "and" and "or" to coordinate multiple arguments ("divisible by 3 and 5") or multiple adjectives ("divisible by 3 and not divisible by 5"). Moreover, because the approach is built on CCGs and therefore directly model compositional meaning of natural language, we can have high confidence that additional sentences using the same words in known grammatical roles will also parse. As an example of this in tandem with modular lexicon extension, adding an entry for "is" allows generalizing already to "any float that is passing is greater than or equal to 5 or less than or equal to 10" — a plausible additional description which strengthens test #2 to imply that only numbers in that range are passing.

Linguistic Findings. Most of our grammatical types and semantics follow standard linguistic models. We did, however, find that using PBTs as linguistic semantics required one additional grammatical distinction in common nouns, between those used to constrain the domain of a property (and therefore correspond to constraining *generators*) and those which play a role in evaluating the property (and therefore correspond to *predicates*). For example, in considering "Every integer is an integer" as a PBT description, the English common noun "integer" corresponds once to a generator of integers (as the domain of the quantifier "every") and once to a predicate on values (which is true only for integers), which would manifest as two lexicon entries for "integer":

$$\text{integer} \vdash CN[Gen] \Rightarrow \texttt{fc.integer()}$$
$$\text{integer} \vdash CN[Chk] \Rightarrow \texttt{Number.isInteger}$$

Our prototype lexicon repeats this distinction with other common nouns to distinguish value checks from generators. This distinction refines standard grammars, similar to a distinction made for mathematical text [53, 54], where additional mathematically-required subcategorizations yield more precise grammars.

In examining the additional 19 PBTs from other texts [27, 43, 52] (available with our prototype [26]), every sentence structure is explained by standard grammatical theories, and can be parsed by a grammar-only CCG parser [65]. We have not implemented semantics for these tests because they require resolution of definite referents, a linguistic construct with subtle semantics [16, Ch. 7], [58, Ch. 7] for which we would prefer to import existing solutions [2]. These additional examples, however, reveal the limitations of focusing on educational material on PBT to find strict English-to-test correspondences. Two of the additional sources specify nearly the same tests for sorting a list [43, 52], one repeats the same FizzBuzz underspecification of STTP [43], and one explicitly states only trivial properties [27] (about adding integers), suggesting work is needed to develop suitable evaluation corpora.

Limitations. Our prototype's primary limitation is lexicon size; words absent from the lexicon cannot be used. Fortunately the lexicon is inherently modular, and for reasons outlined below (Section 5) we believe growing this lexicon is feasible. In addition, our extraction to JavaScript is currently text-based (e.g., textually replacing "`forall`" with "`fc.property`" because NLTK does not allow periods in logical forms). Future improvements should replace this with a proper traversal of the logical form's abstract syntax tree.

5 FUTURE WORK

The key piece of this approach is the lexicon, which clearly must be extended beyond our prototype. Fortunately, there is already a wide-coverage CCG lexicon for English, CCGBank [29, 30], which is able to parse a 4.5 million-word corpus [44] extracted from the *Wall Street Journal*. Though this certainly lacks many software-related terms and includes many terms irrelevant to testing, its more than 74,000 entries cover many different uses of common English words, including quantifiers, prepositions (*from*, *of*, *as*, ...), and distinctions between various verb classes (e.g., with or without direct objects). While some of these will require testing-specific semantics, their grammatical categories can be reused directly, cutting down the most laborious part of lexicon creation. Smaller cross-linguistic lexicons [2] offer a starting point to extend this beyond English. These can be supplemented by work to learn further lexicon extensions [10, 37].

Tests for any program will need to mention program-specific terms, which will need to be added by developers with no special linguistics background. Experiments on CCGBank showed [29] that when training on most of lexicon, the unseen words in a held-out test set were primarily nouns (35.1%) or transformations of nouns (e.g., adjectives, at 29.1%). These are also the simplest categories for non-linguists to provide semantics for (types, objects, and predicates), suggesting that it should be possible to make the lexicon extendable for an individual program by normal developers without special linguistic background. (Similar experiments for a wide-coverage lexicon of German [28] show over half of unknown words to be nouns, suggesting this feasibility extends beyond just English.)

A final key challenge — common to any approach to relate formal and natural language — is to find suitable evaluations for the efficacy of our technique. Focusing on educational material we were only able to identify 26 explicit statements of an intended property in English, among 4 textbooks with explicit coverage of property-based testing [6, 27, 43, 52] (2 others taught PBT without explicitly stating English for any test [18, 62]). Longer-term, the right evaluation approach is to ask developers currently using property-based tests to write natural language describing their test cases, or to seek access to existing closed-source pairings [9].

REFERENCES

[1] 2021. fast-check — Property based testing for JavaScript and TypeScript. https://dubzzz.github.io/fast-check.github.com/

[2] Lasha Abzianidze, Johannes Bjerva, Kilian Evang, Hessel Haagsma, Rik van Noord, Pierre Ludmann, Duc-Duy Nguyen, and Johan Bos. 2017. The Parallel Meaning Bank: Towards a Multilingual Corpus of Translations Annotated with Compositional Meaning Representations. In *EACL*.

[3] Imran Ahsan, Wasi Haider Butt, Mudassar Adeel Ahmed, and Muhammad Waseem Anwar. 2017. A comprehensive investigation of natural language processing techniques and tools to generate automated test cases. In *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*. 1–10.

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

[5] Bharat Ram Ambati, Tejaswini Deoskar, and Mark Steedman. 2018. Hindi CCGbank: A CCG treebank from the Hindi dependency treebank. *Language Resources and Evaluation* 52, 1 (2018), 67–100.

[6] Maurício Aniche. 2020. *Software Testing: From Theory to Practice.* https://sttp.site/ Offline because superceded by [7]. Archived copy at https://web.archive.org/web/20201229191633/https://sttp.site/.

[7] Maurício Aniche. 2022. *Effective Software Testing*. Manning Publications. https://www.effective-software-testing.com/

[8] Ahlam Ansari, Mirza Baig Shagufta, Ansari Sadaf Fatima, and Shaikh Tehreem. 2017. Constructing test cases using natural language processing. In *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*. IEEE, 95–99.

[9] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICS TW)*. IEEE, 1–4.

[10] Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics* 1 (2013), 49–62.

[11] Jason Baldridge and Geert-Jan M. Kruijff. 2003. Multi-modal Combinatory Categorial Grammar. In *EACL*.

[12] Chris Barker and Pauline Jacobson. 2007. *Direct compositionality.* Oxford University Press.

[13] Samuel Bayer. 1996. The coordination of unlike categories. *Language* (1996), 579–616.

[14] Steven Bird. 2006. NLTK: The Natural Language Toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*. 69–72.

[15] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA*.

[16] Bob Carpenter. 1997. *Type-logical semantics.* MIT press.

[17] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. 2010. The RSpec book: Behaviour driven development with Rspec. *Cucumber, and Friends, Pragmatic Bookshelf* 3 (2010), 25.

[18] Paul Chiusano and Runar Bjarnason. 2014. *Functional programming in Scala*. Simon and Schuster.

[19] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*.

[20] Stephen Clark and James R. Curran. 2003. Log-linear Models for Wide-coverage CCG Parsing. In *EMNLP*.

[21] Stephen Clark and James R. Curran. 2004. The importance of supertagging for wide-coverage CCG parsing. In *Proceedings of International Conference on Computational Linguistics 04*. 282–288.

[22] Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building Deep Dependency Structures with a Wide-coverage CCG Parser. In *ACL*.

[23] 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.

[24] Janet Dean Fodor. 1983. Phrase structure parsing and the island constraints. *Linguistics and Philosophy* (1983), 163–223.

[25] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA*.

[26] Colin S. Gordon. 2022. NLPropTest: Parsing English to Property-Based Tests with Categorial Grammars. (1 2022). https://doi.org/10.6084/m9.figshare.14774097.v1

[27] Daniel Hinojosa. 2013. *Testing in Scala.* O'Reilly Media, Inc.

[28] Julia Hockenmaier. 2006. Creating a CCGbank and a wide-coverage CCG lexicon for German. In *ACL*.

[29] Julia Hockenmaier and Mark Steedman. 2005. CCGbank: User's Manual. (2005).

[30] Julia Hockenmaier and Mark Steedman. 2007. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics* 33, 3 (2007), 355–396.

[31] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[32] Pauline Jacobson. 1999. Towards a variable-free semantics. *Linguistics and philosophy* 22, 2 (1999), 117–185.

[33] Pauline I Jacobson and Pauline Jacobson. 2014. *Compositional semantics: An introduction to the syntax/semantics interface.* Oxford University Press.

[34] Aravind K. Joshi, David J. Weir, and K. Vijay-Shanker. 1991. The convergence of mildly context-sensitive grammatical formalisms. In *Foundations issues in natural language processing*. The MIT Press, 31–81.

[35] Sunil Kamalakar, Stephen H Edwards, and Tung M Dao. 2013. Automatically Generating Tests from Natural Language Descriptions of Software Behavior.. In *ENASE*. 238–245.

[36] Marco Kuhlmann, Alexander Koller, and Giorgio Satta. 2015. Lexicalization and Generative Power in CCG. *Computational Linguistics* 41, 2 (2015), 187–219.

[37] M. Collins L. Zettlemoyer. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. *Proceedings of the Conference on Uncertainty in Artificial Intelligence* (2005).

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

[39] Richard Lawrence and Paul Rayner. 2019. *Behavior-Driven Development with Cucumber: Better Collaboration for Better Software.* Addison-Wesley Professional.

[40] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*.

[41] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*.

[42] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In *ESEC/FSE*.

[43] Mikael Lundin. 2015. Property-Based Testing. In *Testing with F#*. Packt Publishing Ltd. https://static.packt-cdn.com/downloads/1232OS_Chapter_11.pdf.

[44] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (June 1993), 313–330.

[45] Koji Mineshima, Ribeka Tanaka, Pascual Martínez-Gómez, Yusuke Miyao, and Daisuke Bekki. 2016. Building compositional semantics and higher-order inference system for a wide-coverage Japanese CCG parser. In *EMNLP*.

[46] Mockito Developers. 2021. Mockito Framework. https://site.mockito.org/

[47] Michael Moortgat. 1996. Generalized quantifiers and discontinuous type constructors. In *Discontinuous Constituency*. Mouton de Gruyter.

[48] Glyn Morrill. 1995. Discontinuity in categorial grammar. *Linguistics and Philosophy* 18, 2 (1995), 175–219.

[49] Glyn Morrill. 2011. *Categorial grammar: Logical syntax, semantics, and processing*. Oxford University Press.

[50] Glyn V Morrill. 2012. *Type logical grammar: Categorial logic of signs*. Springer Science & Business Media.

[51] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise oracles from structured natural language specifications. In *ICSE*.

[52] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. 2008. *Real World Haskell: Code You Can Believe In.* O'Reilly Media, Inc.

[53] Aarne Ranta. 1994. Syntactic categories in the language of mathematics. In *TYPES*.

[54] Aarne Ranta. 1995. Context-relative syntactic categories and the formalization of mathematical text. In *TYPES*.

[55] Valdivino Alexandre de Santiago Júnior and Nandamudi Lankalapalli Vijaykumar. 2012. Generating model-based test cases from natural language requirements for space application software. *Software Quality Journal* 20, 1 (2012), 77–143.

[56] Richa Sharma and Kanad K Biswas. 2014. Natural Language Generation Approach for Automated Generation of Test Cases from Logical Specification of Requirements. In *International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer, 125–139.

[57] Stuart M Shieber. 1985. Evidence against the context-freeness of natural language. In *Philosophy, language, and artificial intelligence*. Springer, 79–89.

[58] Mark Steedman. 2012. *Taking scope: The natural semantics of quantifiers.* Mit Press.

[59] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *ICST*.

[60] Aaron Traylor, Roman Feiman, and Ellie Pavlick. 2021. AND does not mean OR: Using Formal Languages to Study Language Models' Representations. In *ACL*.

[61] K. Vijay-Shanker and David J. Weir. 1994. The Equivalence Of Four Extensions Of Context-Free Grammars. *Mathematical Systems Theory* (1994), 27–511.

[62] Dean Wampler and Alex Payne. 2014. *Programming Scala: Scalability= Functional Programming+ Objects*. O'Reilly Media, Inc.

[63] Matt Wynne, Aslak Hellesoy, and Steve Tooke. 2017. *The cucumber book: behaviour-driven development for testers and developers.* Pragmatic Bookshelf.

[64] Richie Yeung and Dimitri Kartsaklis. 2021. A CCG-Based Version of the DisCoCat Framework. In *Workshop on Semantic Spaces at the Intersection of NLP, Physics, and Cognitive Science (SemSpace)*.

[65] Masashi Yoshikawa, Hiroshi Noji, and Yuji Matsumoto. 2017. A* CCG Parsing with a Supertag and Dependency Factored Model. In *ACL*.