SPLASH: G: On the Lexical Distinguishability of Source Code

Martin Velez

Ph.D. Student at University of California, Davis, **ACM Student Member:** 7823282
Email: marvelez@ucdavis.edu, ***Address:*** 2249 Kemper Hall, One Shields Ave, Davis, CA 95616
Ph.D. Advisor: Prof. Zhendong Su, su@ucdavis.edu

Abstract

Natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. Some words in a sentence, however, cannot be lost without changing the meaning of the sentence. We call these words "wheat" and the rest "chaff". The word "not" in the sentence "I do not like rain" is wheat and "do" is chaff. For human understanding of the purpose and behavior of *source code*, we hypothesize that the same holds. To quantify the extent to which we can separate code into "wheat" and "chaff", we study a large (100M LOC), diverse corpus of real-world projects in Java. Since methods represent natural, likely distinct units of code, we use the ∼9M Java methods in the corpus to approximate a universe of "sentences." We extract their wheat by computing the function's *minimal distinguishing subset (*MINSET*)*. Our results confirm that functions contain much chaff. On average, MINSETS have 1.56 words (none exceeds 6) and comprise 4% of their methods. Beyond its intrinsic scientific interest, our work offers the first quantitative evidence for recent promising work on keyword-based programming and insight into how to develop a powerful, alternative programming model.

1. Introduction

A basic but strong assumption underlies many research and engineering efforts like code search, code completion, keyword programming, and natural programming: From a "small" subset of words, a system can find or generate a larger, executable piece of code.

Code Search breaks the search problem into three subproblems 1) how to store and index code [1, 10], 2) what queries (and results) to support [14, 15], and 3) how to filter and rank the results [1, 9, 11]. To find a piece of code, the user only has one concern: What do I type? *Is there anything the user can type?*

Almost a decade ago, Little *et al.* devised a keyword programming technique to translate keyword queries into valid Java expressions [7]. Several tools and tools and techniques grouped under the general term of Sloppy Programming followed [8, 12]. These tools interpret keyword queries directly by first translating them into source code. SmartSynth [5] is a much more recent incarnation. It generates automation scripts for smartphones from natural language queries. First, it uses natural language processing techniques to parse the queries. Then it applies program synthesis techniques to the parsing result to construct the scripts.

Our *vision* is to *generalize* current keyword programming systems into a new programming model where users "program" using a minimalistic, universal programming language. The programmer should write down thoughts and not worry about syntax details.

Our idea to advance this vision is inspired by the observation that natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. In other words, the sentence or one similar can often be reconstructed given a few key words. We call these words "wheat" and the rest "chaff".

> **Wheat and Chaff Hypothesis:** Units of code 1) consists of "wheat" and "chaff", and 2) the "wheat" is small.

"Wheat" is what a system needs to find or generate a larger, executable piece of code. The definition of "wheat" and "chaff" may vary by application.

To test this hypothesis, we formalize and empirically study a phenomenon of source code we call *lexical distinguishability*. We propose MINSET as a definition for "wheat" potentially useful in code search and code synthesis applications. We hope that if we can *distill* source code into "wheat", then, perhaps, we can gain insights into how to *expand* "wheat" into source code and, thus, take a step toward realizing the new programming model we envision.

Our empirical results support the Wheat and Chaff hypothesis. We find small subset of words that uniquely *map* to larger, executable pieces of code; thus, we provide evidence supporting the assumption underlying much work. The main limitation of our approach is that the "wheat" is artificial; it may not be what a human would use in applications like code search or keyword programming. We show that by focusing on certain lexical features, we can induce more "natural" "wheat".

2. Problem Formulation

2.1 Bag-of-Words Model

In this study, we view functions as the units of code. Functions are natural, likely distinct, pieces of code and functionality. We represent a unit of code, function, as a set of lexical features or bag-of-words. We disregard syntactic structure, order, and multiplicity. First, we parse each function to get its set of lexemes. Then, we map each lexeme to a *word*. A *word* is a lexeme, or some abstract or refined form of it.

2.2 Lexicons

A *lexicon* is a set of words.

There are two natural views we can take of code: the raw sequence of lexemes the programmer sees when writing and reading code, and the abstract sequence of tokens the compiler sees in parsing code. We want to explore those two views, and capture each one as a lexicon, a set of words. LEX is the set of all lexemes found in our corpus (5,611,561 words). LTT is the set of lexer token types defined by the compiler (101 words) [13]. Each word in LTT is an abstraction of a lexeme, like `3` into `INTLIT`.

The freedom to define the lexicon allows us to sharpen, blur, or even disregard certain lexical features. New lexicons can be formed by abstraction over lexemes. New lexicons can also be defined by filtering specific lexemes. For example, we can define a lexicon consisting of all lexemes except separators, like "`(`" and "`)`".

Functions may contain, to adapt a word from linguistics, *homonyms*: identical lexemes with distinct effects on behavior. For example, in Java, the lexeme "`get`" could be a method call of "`java.util.Map.get()`" or "`java.util.List.get()`". We can also blur lexical differences by abstracting distinct lexemes we suspect have the same effect on behavior, *i.e. synonyms*, to the same word.

2.3 Illustration of Two Bags

Figure 1 shows two Java methods found in two real-world projects, Apache Log4j and JMRI (A Java Model Railroad Interface), represented as bags of words over LEX. To help visualize the similarity between these two methods, we have shaded the words in common; there have 21 words in common. This should not be surprising. Both methods implement the same functionality.

2.4 Lexically Distinguishable Code

Finding what distinguishes a function lexically is thus reduced to finding a unique subset of lexical features or words. This unique subset distinguishes each function from all other functions (when each functions is represented as a bag-of-words). We call any such subset a *distinguishing subset*, and define it precisely in Definition 2.1. A function may not have a distinguishing subset. We call those that do, *distinguishable* (Definition 2.2).

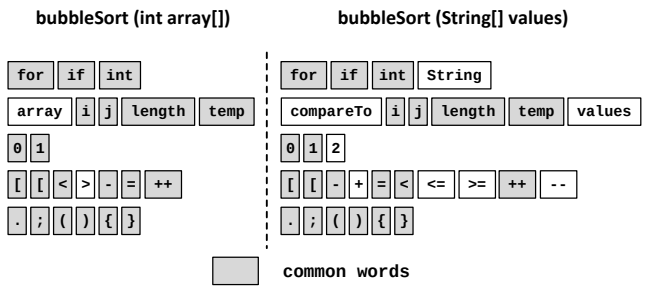

Figure 1: Two bags of words. (left) Sorts an array of integers using Bubble Sort. (right) Sorts an array of Strings using Bubble Sort.

Definition 2.1. Given a finite set S, and a finite collection of finite sets $\mathcal{C}$, S^* is a *distinguishing subset* of S if and only if

(P1) $S^* \subseteq S$ — S^* is a subset of S

(P2) $\forall C \in \mathcal{C},\ S^* \not\subseteq C$ — S^* is *only* a subset of S

Definition 2.2. A unit of code is *lexically distinguishable* if it has a distinguishing subset.

A unit of code may have more than one distinguishing subset. To determine if it is distinguishable, we simply need to find one. We focus on finding a *minimum distinguishing subset* (MINSET). We call this The MINSET Problem (Definition 2.3). It is the *core* computational problem that we study.

Definition 2.3 (The MINSET Problem)**.** Given a finite set S, and a finite collection of finite sets $\mathcal{C}$, find a *minimum* distinguishing subset (*minset*) S^* of S.

Theorem 2.1. MINSET *is NP-hard.*

Proof. We reduce HITTING-SET to MINSET. □

A MINSET identifies a piece of code. It consists of lexically distinguishing features. Some features may crucially differentiate its behavior from similar functions. Some may not. In the keyword-query sense, a MINSET is the smallest query that will uniquely identify and recall a piece of code. It may not be what humans would actually attempt to use. That is a separate challenge. In this study, we focus on finding and studying minsets.

2.5 The MINSET Algorithm

Since the MINSET problem is NP-hard, we developed MINSET, a greedy (approximation) algorithm which we omit due to lack of space. At a high level, in each iteration, the algorithm selects the rarest element of a set S with respect to $\mathcal{C}$, a collection of sets. At termination, it finds the locally minimal distinguishing subset of S, if one exists. Figure 2 shows a sample run of the algorithm. The worst case time complexity of MINSET$(S,\mathcal{C})$ is $O(|S|^2|\mathcal{C}|)$. We have also developed a multiset version of the MINSET algorithm.

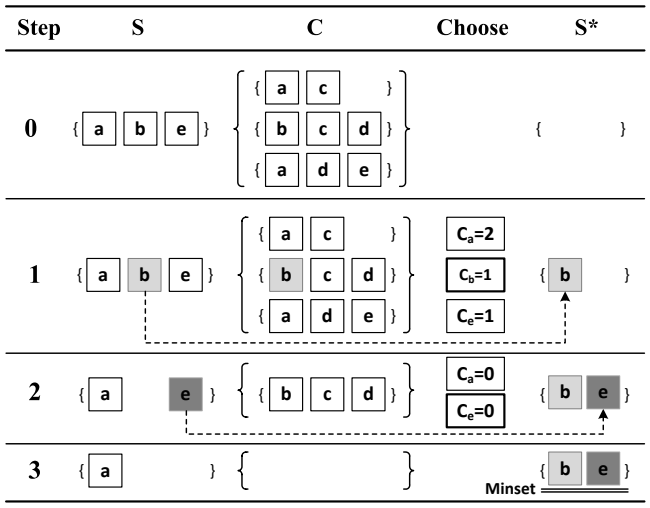

Figure 2: MINSET($\{a,b,e\},\{\{a,c\},\{b,c,d\},\{a,d,e\}\}$).

Table 1: Lexicons.

Name	Description	Size (words)
LEX	All (raw) lexemes	5,611,561
LTT	All lexer token types	101
MIN1	Fully qualified standard library method names and basic operators	55,543
MIN2	MIN1 plus control keywords	55,556
MIN3	MIN2 plus fully qualified public type names	91,816
MIN4	MIN3 plus additional keyword and token types	91,829

3. Empirical Setup

We selected a very popular, modern programming language, Java, and collected a large (100M lines of code), diverse corpus of real-world projects. Ignoring scaffolding and very simple methods, which we define as those containing fewer than 50 tokens[1], there are 1,870,905 distinct methods in our corpus. We selected a simple random sample of 10,000 methods[2]. Our software and data is available online[3].

4. Results and Analysis

We provide quantitative and qualitative results for the following questions:

1. How many units of code are lexically distinguishable?
2. How much of code is needed to distinguish it?
3. To what extent do minsets also capture code behavior and behavior differences?
4. What is a natural, minimal lexicon?

[1] This filtering of Java methods is consistent with other research [2, 6].

[2] Given the population size, this sample size gives us a confidence level of 95%, and a margin of error of ±1% in our measures.

[3] `https://bitbucket.org/martinvelez/code_essence_dev/downloads`.

Measures[2] We define the *yield* of a lexicon to be the percentage of distinguishable methods in our corpus. The second question can be addressed in terms of absolute *minset size*, or in terms of *minset ratio*, minset size to threshed method size. While minset sizes and minset ratios will almost undoubtedly vary across functions, we hypothesize that the *mean* minset size and the *mean* minset ratio are small.

Lexicons We studied 6 different lexicons, listed in Table 1. LEX and LTT are the lexicons we discussed in Section 2.2. MIN1- MIN4 are lexicons we explore in the search for a natural, minimal lexicon.

Summary Code is lexically distinguishable. Perhaps just as importantly, only 1.56 words, on average, or just 4%, are needed to distinguish a unit of code from all others in the corpus over LEX. The problem with minsets over LEX is that they do no capture behavior and behavior differences well. Over MIN4, on the other hand, minsets are still small but reveal much more about the behavior of the code because we intentionally blurred lexical differences which we suspect do not distinguish behavior. We elaborate on this point in Section 4.2 and Section 4.3.

4.1 Lexical Distinguishability over LEX and LTT

Figure 3 show the results of computing minsets over LEX. Using LEX, a tiny proportion of code is needed to distinguish it. The minset of a method, on average, contains 4.57% of the unique lexemes in a method which means that methods in Java contain a significant amount of chaff, 95.43% on average. More surprisingly, the number of lexemes in a minset is also just plain small. The mean minset size is 1.55. The largest minset consists of only 6 lexemes.

Yield Figure 4 shows that the yield decreases as the lexicon becomes coarser, measured roughly by the number of words in the lexicon. Our coarsest lexicon, LTT, blurs lexical differences too much. Over LTT, only 87 out of 10,000, 0.87%, methods have a minset. This is in great part due to the fact that LTT induces many duplicates. Recall that all of these methods are unique at the source code level. In contrast, LEX appears to preserve sufficient lexical differences so that 9,087 out of 10,000 methods have a minset.

4.2 What is a Natural, Minimal Lexicon?

The problem with minsets over LEX is that they do not capture behavior and behavior differences well. An example of a minset is {`"Joda"`}.

Goal We set the goal of finding a lexicon that is *minimal*, as small as possible, and *natural*, consisting of words that a human would know and use in applications like code search and code synthesis. The words in this lexicon should be *meaningful*, in the sense that they reveal information about the behavior of the code to us, humans.

Strategy We search by exploring the lexicon spectrum toward more abstract views of code. We additively construct a bag of words that approximates what a programmer might

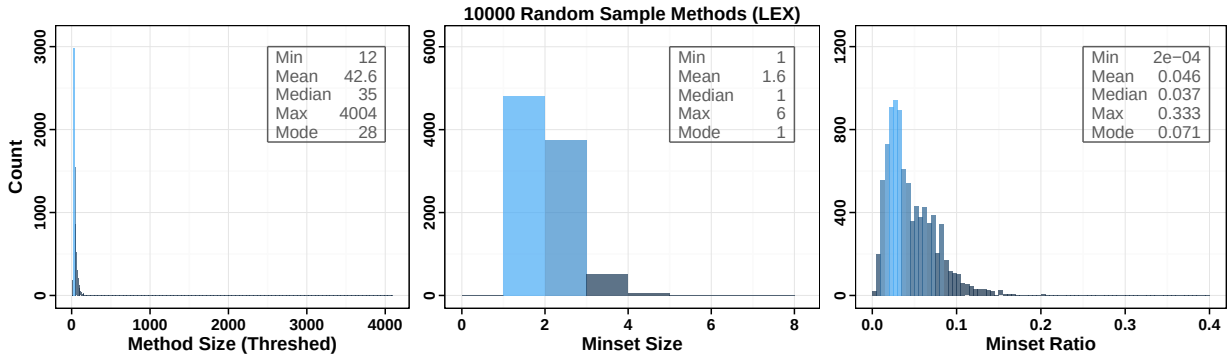

Figure 3: The histogram of minset sizes tells us that minsets are small. Comparing minset sizes with method sizes shows that minsets are also relatively small. The minset ratio histogram confirms this.

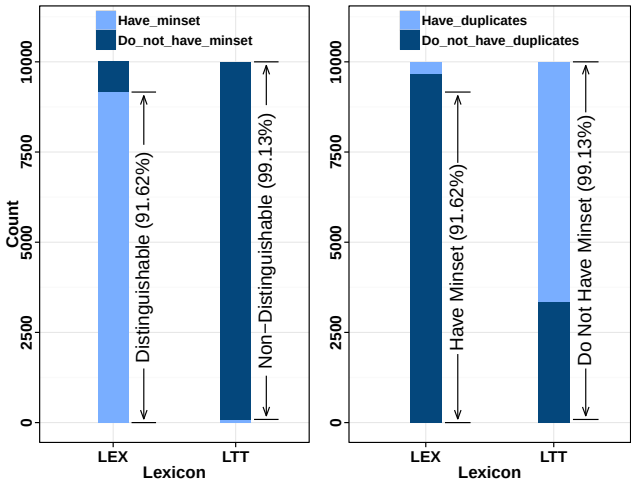

Figure 4: Random Sample of 10,000 Methods: (left) *Proportion of Methods with Minsets* (right) *Proportion of Methods with Duplicates*

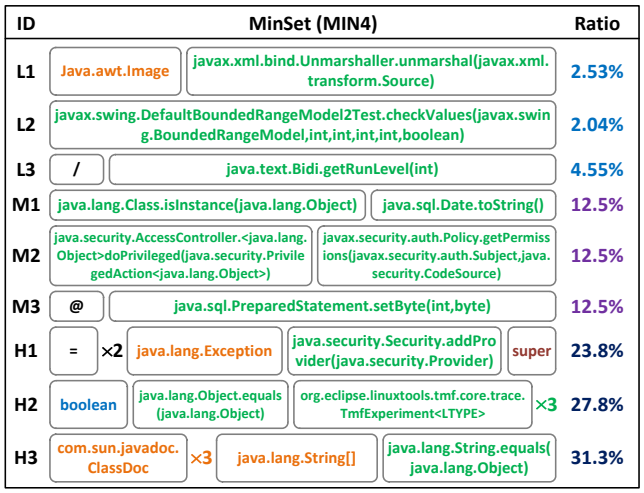

Figure 5: This shows the minsets of nine methods (MIN4). L1-L3 are minsets that have low minset ratios. M1-M3 have medium minset ratios. H1-H3 have high minset ratios.

naturally use in applications like code search and code synthesis. We considered four candidates, lexicons. We list and briefly define them in Table 1. We settled on MIN4.

MIN4 MIN4 includes everything in MIN3 plus `false`, `true`, and `null`, object reference keywords, like `this` and `new`, and the token types of constant values, such as the token type `Character-Literal` for 'Z' or, for `5`, `Integer-Literal`. In total, we added 13 new words. Our intuition is that the *use* of hard-coded strings and numbers is connected to behavior. Certainly, knowing that hard-coded values are used can be informative. Also, in an application, a programmer may need to indicate that some constant string or number will be used. For example, if the programmer wishes to find a method that calculates the area of a circle, then it would be natural to indicate that target method likely contains a `Float-Literal` like `3.14`. After including these words, the mean and maximum minset size remain small, 3.06 and 10, respectively. The yield increased from 41.44% to 44.79%.

Adding this small number of semantically-rich words to the lexicon seems to be another reasonable exchange for a noticeable gain in yield: under this lexicon, the minsets are easier to interpret (see Section 4.3 for our analysis of the interpretability of minsets built from these words) while remaining small enough for humans to work with, *e.g.* a human could potentially write a minset from scratch while programming using key words [7].

4.3 Minsets Over MIN4

Figure 5 shows nine minsets over MIN4. *To what extent do minsets over* MIN4 *capture behavior and behavior differences amongst methods?* We find a qualitative answer to this question via case studies: Minsets over MIN4 give insight into the behavior of a method. Here we discuss one.

Low: **L1** The method named `javax.xml.bind.Unmarshaller.unmarshal` from `(java.xml.transform.Source)` deserializes

XML documents and returns a Java content tree object; `java.awt.Image` is an abstract classes that represents graphical images. From this minset, we infer that this method handles images and XML files. Since it reads the XML file, we also infer that it uses XML data in some manner. Perhaps the file contains a list of images, or the data in the file is used to create or alter an image. After inspecting the source code, we find that it is a method in the `LargeInlineBinaryTestCases` class of the Eclipse Link project, which manages XML files and other data stores. Our understanding was not far off: the method does read a binary XML file that contains images.

5. Related Work

Although we are the first to study the phenomenon of lexical distinguishability of source code, and propose the Wheat and Chaff Hypothesis[4], a few strands of related work exist.

Code Uniqueness At a basic level, our study is about uniqueness. *What lexical features distinguish or uniquely identify a unit of code?* Gabel and Su also studied uniqueness [3]. They found that software generally lacks uniqueness which they measure as the proportion of unique, fixed-length token sequences in a software project. We studied uniqueness differently. We capture uniqueness as the absolute or proportional size of minsets. The elements in a MINSET may not be unique or even rare but together uniquely identify a piece of code. We keep in mind that syntactic differences do not always imply functional differences as Jiang and Su demonstrated [4]. Thus, in some cases the uniqueness may be accidental. Two minsets may, in fact, represent the same behavior at some higher, more abstract semantic level.

6. Conclusion and Future Work

We have shown that source code is lexically distinguishable, and provided qunatitative and qualitative evidence to support the Wheat and Chaff Hypothesis. We believe this reinforces the assumption underlying work like code search and keyword programming. This work also provides confidence and insight into realizing the new, alternative programming model we envision where the programmer writes down "wheat" and the system fills in the "chaff".

In future work, we will study alternative units of code like blocks, abstract syntax tress, *n*-grams. We will also study different programming languages especially in different paradigms like Lisp and Prolog.

References

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 681–682, 2006.

[2] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 513–516, 2007.

[3] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 147–156, 2010.

[4] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[5] V. Le, S. Gulwani, and Z. Su. SmartSynth: synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, pages 193–206, 2013.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[7] G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, pages 84–93, 2007.

[8] G. Little, R. C. Miller, V. H. Chou, M. Bernstein, T. Lau, and A. Cypher. Sloppy programming. In A. Cypher, M. Dontcheva, T. Lau, and J. Nichols, editors, *No Code Required*, pages 289–307. Morgan Kaufmann, 2010.

[9] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PDLI*, pages 48–61, 2005.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *ICSE*, pages 111–120, 2011.

[11] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *ICSE*, pages 848–858, 2012.

[12] R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, D. Karger, and m. schraefel. Inky: a sloppy command line for the web with rich visual feedback. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, pages 131–140, 2008.

[13] Oracle openJDK. `http://openjdk.java.net/`, 2012.

[14] S. P. Reiss. Semantics-based code search. In *ICSE*, pages 243–253, 2009.

[15] S. P. Reiss. Specifying what to search for. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 41–44, 2009.

[16] R. Rivest. Chaffing and winnowing: Confidentiality without encryption, March 1998. web page.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[4] Others have used the "wheat and chaff" analogy in the computing world but in different domains [16, 17].