Mining Programming Language Vocabularies from Source Code

Daniel P. Delorey[1], Charles D. Knutson[2], and Mark Davies[2]

[1] Google, Inc.
720 4th. Ave Ste 400
Kirkland, WA 98033
delorey@google.com

[2] SEQuOIA Lab
Department of Computer Science
Brigham Young University
Provo, UT 84602
knutson@cs.byu.edu

[3] Department of Linguistics
Brigham Young University
Provo, UT 84602
mark_davies@byu.edu

Abstract. We can learn much from the artifacts produced as the by-products of software development and stored in software repositories. Of all such potential data sources, one of the most important from the perspective of program comprehension is the source code itself. While other data sources give insight into what developers *intend a program to do*, the source code is the most accurate human-accessible description of what it *will do*. However, the ability of an individual developer to comprehend a particular source file depends directly on his or her familiarity with the specific features of the programming language being used in the file. This is not unlike the difficulties second-language learners may encounter when attempting to read a text written in a new language. We propose that by applying the techniques used by corpus linguists in the study of natural language texts to a corpus of programming language texts (i.e., source code repositories), we can gain new insights into the communication medium that is programming language. In this paper we lay the foundation for applying corpus linguistic methods to programming language by 1) defining the term "word" for programming language, 2) developing data collection tools and a data storage schema for the Java programming language, and 3) presenting an initial analysis of an example linguistic corpus based on version 1.5 of the Java Developers Kit.

1 Introduction

Vocabulary is a central focus of *linguistics*, the scientific study of language[2,8,17,18]. Vocabulary knowledge is a critical factor in reading comprehension and writing quality [13,3]. Research has also revealed thresholds of vocabulary size below which reading comprehension is impossible, as well as thresholds above which reading becomes pleasurable [9]. Vocabulary provides linguists one framework for discussing and assessing an individual's knowledge of and ability with a particular language.

However, linguists study natural language, and it is not yet clear that results based on the study of natural language can be generalized to programming languages. Naur [14,15] articulate the differences between natural language and programming language well. He argues that: 1) natural languages are used [and studied] by linguists mostly in spoken form, 2) programming languages are constrained to be precise and unambiguous so that they may be "understood" by compilers and interpreters, and 3) in programming language, there is no equivalent of literature available for study by researchers and language learners.

In recent years, however, shifts in both the study of linguistics and the use of programming language have made it possible to apply linguistic techniques to the study of programming language. We hypothesize that if it can be shown that the use of programming language mimics the use of natural language, it would be possible to, to paraphrase Brooks, "appropriate for [software engineering] the large body of knowledge as to how people acquire language" [5].

One important shift has been the emergence of *corpus linguistics* which emphasizes the study of *language use* based on data from *written language*. Traditionally, linguists have focused on the study of *language structure* relying primarily on data from *spoken language* [4]. The rise in the importance of written language is a necessary precondition of the application of linguistic techniques to the study of programming language which, of course, is presented almost exclusively in written form.

Another important shift has been the surge in Open Source Software (OSS) activity which has greatly increased the public availability of source code of widely varied origin and quality. While clearly not exactly what Naur envisioned, this wealth of programming language text can rightly be taken as the "body of literature" he references when he writes, "When a suitable body of literature has become available, other forms of linguistic study will become pertinent" [14].

A final important shift has been the advent of the software repository mining community which has created tools, techniques, and justifications for empirical studies based on the artifacts of software development. Such studies allow researchers to observe developer behavior retroactively without influencing it by imposing the burden of additional data collection.

In light of these changes, we believe the current environment is conducive to the expansion our study of programming language to include not only its design and theory but also its use by practitioners. Such study will allow us to answer basic questions such as:

– How many lexical items are used regularly by an average software developer or in an average project?
– What are the most commonly used lexical items in a particular programming language?
– How do patterns of language use vary across application domains and across programming languages?
– How has the use of programming languages evolved over time and does it in any way mimic our use of natural language?

Once we can answer fundamental questions about the ways in which programming language is used, we will be better able to understand how prior work by linguists can appropriately be extended to programming languages. Among the areas of prior work in linguistics we believe could be extended to programming language are: 1) *language learning* where linguists have developed methods for assigning grade levels to texts to assist students in choosing reading material, 2) *language teaching* where linguists have developed curricula for second-language learning, and 3) *language assessment* where linguists have developed tests to evaluate an individual's vocabulary.

In this paper we present a framework for extending the techniques of corpus linguistics to programming language. We propose corollaries for the relevant linguistic terms and present our designs for data collection tools and a storage schema. As a proof of concept, we also present preliminary results based on our analysis of version 1.5 of the Java Developers Kit (JDK 1.5).

2 Related Work

Empirical studies of vocabulary based on source code are not new. Reports of such studies appear in the literature. These can loosely be categorized as either 1) studies that focus on mappings from the identifiers used in programming language to words in some natural language (almost always English) or 2) studies that focus on the size and composition of the set of natural language words used by developers – the *programmer lexicon*, to borrow their term.

In the category of studies that map programming language vocabularies to natural language vocabularies, Takang [19], studied the effect of "full" identifier names – that is, complete English words rather than abbreviations – as well as the combined effect of identifiers and comments on program comprehension. Using a controlled experiment of 89 undergraduate Computer Science students and four versions of a small Modula-2 source file, he found that "full" identifiers had a greater positive influence on comprehension than did comments.

Work on mapping programming language identifiers to natural language words has more recently been extended by Lawrie [10,11] who uses a mapping to quantify the "quality" of identifiers. She defines quality as "the possibility of constructing the identifier out of dictionary words or known abbreviations." Based on her analyses, she concludes "better programming practices [produce] higher quality identifiers" and "proprietary code [has] more acronyms than open source code."

In the category of studies that attempt to catalog the programmer lexicon, Antoniol [1], mined the set of natural language words used by developers on three OSS projects. The goal of that research was to compare the vocabulary changes to structural changes during the same time period (For example, do the words developers use to talk about a program evolve at the same rate as the structures they use to represent it?). The three main findings of this research are: 1) the lexicon is more stable than the structure, 2) changes to the lexicon are rare during software evolution, and 3) as a system becomes more functionally stable, the structural and lexical stabilities converge toward 1 (unchanging).

In contrast to these previous studies of vocabulary based on source code, our research focuses exclusively on programming language. We do not deconstruct identifiers into natural language components. Instead, we apply techniques developed for the study of natural language to programming language directly. We believe that rather than superseding or competing with the research cited above, our work is complementary and can give a richer understanding of the use of programming language in practice.

3 Define "Word"...

Unlike prior studies which focused on natural language and could, therefore, use the linguists' definitions of terms directly, our focus on programming language requires us to find suitable corollaries to their terms in our own domain. We begin in this section by defining the semantically loaded term "word" for programming languages.

For our purposes, *words* are the atomic units from which vocabularies are composed[1]. While it may initially seem trivial to define "word," in practice the definition can be quite elusive. In natural language, for example, a simple definition of "word" could be a string of contiguous alphabetic characters (i.e., word form). This definition, however, ignores the fact that a word form may have multiple meanings and the fact that groups of words (known as *idiomatic phrases*) may have a meaning that is independent from the meanings of the individual words. Similar problems exist in programming languages where a particular string of characters may be reused to represent more than one function and/or object. For a more detailed discussion of this difficulty as it applies to natural language, see [7].

3.1 Levels of Abstraction

There are, broadly speaking, four potential levels of abstraction at which to define "word" for programming language vocabularies: 1) the lexicographic level, 2) the syntactic level, 3) the semantic level, and 4) the conceptual/functional level. The specifics of each of these potential definitions are provided below.

Defining "word" at the *lexicographic* level of abstraction means that identical strings of contiguous non-white-space characters are counted as the same word. This is the simplest form of word counting. Borrowing terminology from compiler design, a data gathering tool based on a lexicographic definition of "word" requires only a general purpose lexical analyzer (*lexer*) and need not recognize any of the syntactic rules of the underlying programming language. This approach is the most broadly applicable and the most forgiving of errors in source code.

[1] Linguists also define a semantic unit below *word* which is the *morpheme*. We will not consider the programming language equivalent of morphemes in this paper.

These gains, however, come at the cost of accuracy. While such a tool can be applied without modification to all written texts regardless of language, the word frequency counts it produces are highly suspect. It is almost certain, for example, that the "most frequent" word forms it identifies are also the most polysemous[2] or homonymous[3] words in the language.

If we instead define "word" at the *syntactic* level of abstraction, we group lexicographically equivalent words only if they are used as the same part of speech. For example, a class name and a function name that are lexicographically isomorphic are counted as separate words. Again borrowing our terms from compilers, this is a more complex form of word counting that requires not only a *lexer*, but a language-specific *parser* for each target programming language and a mechanism for determining which parser to use on a given source file. The parsers can be made resilient to syntactic errors, however, so that incomplete or partially incorrect source files can still be processed. Also, by using syntactic knowledge intelligently, we are able to differentiate the *definition* of a word from its *use*. This allows us to detect polysemy and make approximate adjustments in our analyses.

A definition of "word" at the *semantic* level of abstraction allows the most complex form of word counting that can be handled automatically for programming languages[4]. A semantic definition of "word" groups syntactically equivalent words only if they refer to the same *definition*. In addition to a lexer and a parser, a *symbol table* is required to accommodate a semantic definition of "word". A data gathering tool that relies on a semantic definition of "word" is the most language-specific and the least tolerant of errors. Further it requires that the definitions of all words used in a source file, even those defined in external libraries and packages, be available during processing. Data gathered using a semantic definition of "word" have the advantage of allowing accurate identification of polysemy and homonomy.

The fourth potential level of abstraction at which "word" may be defined is included here for analytical completeness although its implementation is infeasible in practice. When using a conceptual/functional definition of "word", the string of characters used to represent a word are no longer relevant. Instead, two words are the same if they have the same "meaning." Two functions in a programming language, for example, may have the same meaning if they produce identical results (including all outputs and side effects) for all inputs. Two variables may have the same meaning if they always contain identical data values and exist in memory at the same points in time. It should be clear to anyone familiar with the Halting Problem [6] that such similarity comparisons are impossible in all but the most trivial cases.

For the preliminary tools and results we present in this paper, we have chosen to define "word" at the syntactic level of abstraction although we have extended our data collection to allow some simplistic approximations of the semantic level. This allows us to process incomplete and "un-compilable" source files, an important consideration when gathering data from the version control repositories of active software development projects. The source code in such repositories tends to be in a perpetual state of flux with high levels of manual preparation required to get the project into a "compilable" state.

Having determined the level of abstraction at which to categorize words, we next define which tokens in a source file constitute a word.

3.2 Types of Words

Linguists recognize two types of words in natural language: *function words* and *content words*. In addition, many written natural languages include punctuation which is usually uninteresting from the perspective of vocabulary research.

[2] Polysemes are words that have multiple related meanings.
[3] Homonyms are words that have multiple unrelated meanings.
[4] The syntactic level is currently the most complex form of word counting that can be consistently handled automatically for natural languages.

Function words belong to the closed set of a language and carry little or no meaning on their own. They are necessary to express the relationship between the content words in a sentence and to identify for the reader the parts of speech represented by adjacent words. Conjunctions, articles, and prepositions are examples of function words. The set of function words is always smaller than the set of content words in natural language and it is common for fluent speakers of a language to know most, if not all, the function words.

Content words belong to the open set of a language and carry the majority of the meaning in a text. They represent the abstract ideas and the actions in a sentence. Nouns, verbs, adjective and adverbs are examples of content words. Since the set of content words is open, new content words are often created and it is rare for an individual speaker to employ most, or even a large percentage, of all the content words in a language.

In programming language, keywords, like function words, belong to the closed set of the language and are easily recognizable to a fluent user of the language. Keywords provide the structure of the source code.

Identifiers belong to the open set of a programming language and carry much of the meaning of a piece of source code. The percentage of identifiers in a specific piece of source code that are recognizable and familiar to a developer depends largely on the developer's past experience with the particular file, with other files from the same project or application domain, and with other files written by the original author of the source code.

In addition, many programming languages include *operators* whose form resembles natural language punctuation but whose purpose is akin to that of function words. For example, *mathematical operators* and *scoping blocks* in Java serve to express relationships between identifiers.

The reader may rightly question our decision to include operators as "words" in a programming language. Clearly there can be arguments made both for and against including them. Our philosophy regarding data is that we must make every effort not to discard potentially useful data during the data gathering phase of our research so that it will be available should we find reason to use it in the analysis phase of our research. Put simply, we can exclude the data from analysis once we have it, but we cannot include it in the analysis if we never collect it.

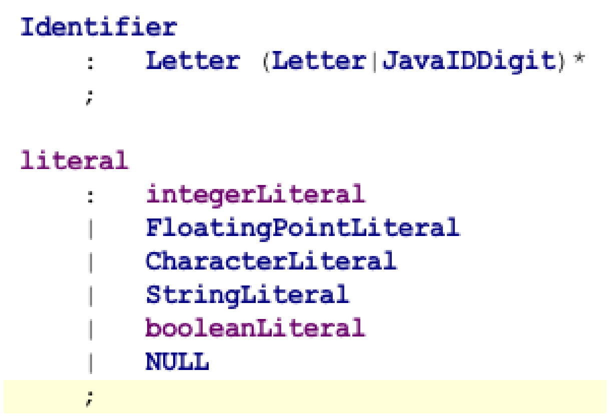
Together keywords, operators, and identifiers constitute the set of tokens which may be considered "words" in a *programming language vocabulary.* Having defined the term "word" for programming languages, we now use our definition to discuss the creation of a *linguistic corpus of programming language.*

4 Linguistic Corpora from Source Code

In order to perform corpus-linguistic analyses, we must first construct a *corpus.* A corpus is a collection of texts selected to be broadly representative of the language being studied and is machine-readable [12]. A discussion of the guiding principles for selecting a representative set of texts for a linguistic corpus is beyond the scope of this paper (we direct the interested reader to [20]). In this section we discuss a method for creating a linguistic corpus from source code repositories and present a preliminary analysis of an example corpus.

In order to make a corpus machine-readable, the individual tokens of each text in the corpus must be tagged with some level of meta-data. The amount of detail and the level of abstraction required in the meta-data depends on the analyses the researcher intend to perform on the corpus. In the case of natural language, the process of rendering texts machine-readable may require a high level of human involvement, especially if semantic meta-data is required. However, syntactic tagging (for example, part-of-speech tagging) and some limited semantic tagging (for example, lemmatization) can often be handled automatically.

Programming language, on the other hand, is intended to be machine readable by design. Of course, individual programming language texts may not be machine readable due to either syntactic errors (for example, forgotten closing braces) or semantic omissions (for example,



```
Identifier
    :   Letter (Letter|JavaIDDigit)*
    ;

literal
    :   integerLiteral
    |   FloatingPointLiteral
    |   CharacterLiteral
    |   StringLiteral
    |   booleanLiteral
    |   NULL
    ;
```

Fig. 1. Example ANTLR Rules

missing libraries). In general, though, the process of using a computer to parse and analyze source code is well understood and can be readily employed to tag individual tokens in a source file with both the syntactic and the semantic meta-data necessary in a linguistic corpus.

As a proof of concept of our methodology, we collected a small example corpus of the Java programming language based on the JDK 1.5. The reasons we chose Java include: 1) Java is widely popular and there is a large amount of Java source code publicly available for analysis, and 2) Java offers an appropriate level of syntactic complexity for our prototype development – neither too complex nor too simple.

4.1 Data Collection

We used ANTLR [16] to develop a Java-specific lexer and parser. ANTLR is a tool which generates language recognizers from *grammars*. Grammars are expressed as a set of *rules* composed of combinations of regular expressions, lexical tokens, and references to other rules. Two simple example ANTLR rules are shown in Figure 1.

In order to extract vocabulary data from Java source files, we produced two ANTLR grammars. The first, which consists of 139 *parser rules* and 21 *lexer rules*, takes source code as input and produces an abstract syntax tree (AST) as output. The second, which consists of 80 *tree parser rules*, takes an AST as input and use a data access object to output vocabulary data to a database.

We used the source code from the JDK 1.5 to validate these parsers. We briefly discuss these efforts in Section 4.3

4.2 Data Storage

The data extracted from source code is stored in a relational database. The schema for this data store is shown in Figure 2. This schema is specific to Java and tailored to our current definition of the term "word" as discussed in Sections 3 and 3.2. The schema contains six tables, each of which is described below.

The records in the `file_collection` table can be used to partition the data from individual runs of the data gathering tool. Each record consists of an automatically generated primary key identifier, an arbitrary name assigned by the user, and an automatically generated time stamp. This allows us to compare groups of files, such as in time-based analyses on multiple revisions of the same project.

Records in the `file` table each contain the data needed to describe an individual file within a file collection. The first two fields are the automatically generated primary key and the foreign

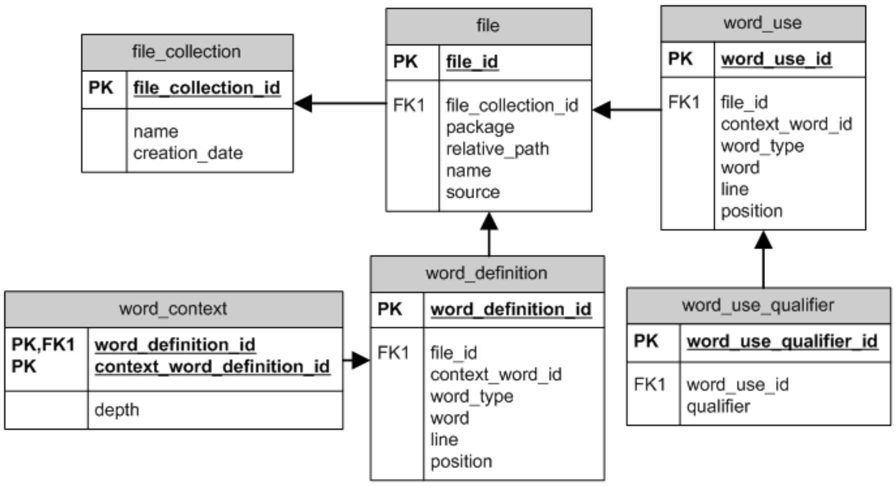


Fig. 2. ER Diagram of the Storage Schema

key to the `file_collection` table. The `package` field stores the Java package to which this file belongs. This value can be used to group and separate files by package within and across file collections. The `relative_path` and `name` fields locate a file within its file collection. They can also be used, for example, in comparisons of multiple revisions of a single file across file collections. The `source` field contains the original source code of the file which allows future analyses to be performed directly from the database instead of requiring researchers to maintain both the database and a collection of source files. This also simplifies the distribution of the data to other researchers.

The remaining four tables in the schema store data that deal directly with the words themselves. Each of the two main tables, `word_definition` and `word_use`, has an ancillary table, `word_context` and `word_use_qualifier` respectively.

The `word_definition` and `word_context` tables store data about words that are defined in a file. Each `word_definition` record has a foreign key to the `file` table indicating the file in which the word is defined. Each record also has a self-referential foreign key, `context_word_id`, to the context word that provides the scope in which the current word is defined. For example, a class method would have a foreign key to the definition of the class to which it belongs; a local variable would have a foreign key to the definition of the method in which it is defined. The top-level class in each file has a null value for the `context_word_id` field since it is not defined in the context of any other word.

Since context relationships between word definitions can be many layers deep (e.g., a variable may be defined inside a method which is defined inside an internal class which is defined inside another class, etc.) the `word_context` table is used to directly link a definition to all the words defined within its scope regardless of their depth. This allows us to avoid running computationally expensive recursive queries to rebuild these relationships.

The `word_use` and `word_use_qualifier` tables store data about the words that occur in a file. Each `word_use` record has one foreign key to the file where the word occurs and one foreign key to the scope in which the word occurs. Each word used in a definition is counted as having occurred inside the scope of that definition even if it occurs before the identifier being defined. For example, the keyword `class` is counted as part of the definition of a class even though it occurs before the class name.

For programming languages that allow identifiers to be reused, compilers require explicit disambiguation. The Java programming language provides a disambiguation mechanism known

Table 1. Possible values of the `word_type` field of the `word_definition` and `word_use` tables

Value	Description	In `word_definition`?	In `word_use`?
annotation	An annotation class	Yes	Yes
label	The label of a labeled statement	Yes	Yes
method	A standard method	Yes	Yes
annotation_method	An annotation method	Yes	No
class	A standard class	Yes	No
constructor	A constructor method	Yes	No
enhanced_for_element	The control element of a for-each loop	Yes	No
enum	An enumeration class	Yes	No
enum_constant	A constant element of an enumeration class	Yes	No
initilializer	A class initializer block	Yes	No
interface	An interface class	Yes	No
parameter	A standard parameter	Yes	No
static_initializer	A static class initializer block	Yes	No
type_parameter	The type parameter of a generic method or class	Yes	No
var_arg_parameter	A variable argument parameter	Yes	No
variable	A variable definition	Yes	No
binary_operator	Any operator with two operands	No	Yes
cast_operator	The cast operator	No	Yes
condition_operator	The ternary conditional operator	No	Yes
exception	An exception object	No	Yes
field	A data member of a non-primitive type	No	Yes
keyword	A Java keyword	No	Yes
label_operator	The label operator	No	Yes
package	A package name	No	Yes
primitive_type	A Java primitive type	No	Yes
scoping_block	A scoping block	No	Yes
type	A non-primitive or class type	No	Yes
unary_operator	Any operator with one operand	No	Yes

as a *qualifier*. These strings of period-separated tokens direct the Java compiler as it attempts to locate the correct definition of an identifier in its symbol table. For example, when we write the statement `System.out.println("Hello World!");`, we are calling the method `println` with the parameter "`Hello World!`" and we are directing the compiler that the `println` method we intend is the one that is a member of the `PrintStream` object name `out` which is a data field of the `System` object.

While this is a simple example, Java qualifiers can become arbitrarily complex and a complete symbol table is required for their accurate interpretation. As a rudimentary approximation, we separate any qualifiers we encounter from the words they are being used to disambiguate and store them in the `word_use_qualifier` table. We use a secondary table rather than storing the qualifier as a field in the `word_use` table because it is far more common for a word not to have a qualifier and the secondary table allows us to avoid having a large number of null fields in the `word_use` table.

As shown in the schema diagram, both the `word_definition` and the `word_use` tables have a `word_type` field. These are similar to the part-of-speech attribute of a natural language word and constitute the essential bit of information that separates the lexicographic level of abstraction from the syntactic level of abstraction. The possible values these fields may contain are listed in Table 1.

Word types that can be used only in the `word_definition` table are those which can be identified at the syntactic level from a word definition, but not from a word use. For example, when a class is defined, syntactically we know it is a class because the keyword `class` precedes the identifier. When a class is used, however, syntactically we know only that a non-primitive type is being used. We do not know, at the syntactic level, whether that non-primitive type is a class, an interface, or an enumeration.

Word types that can be used only in the `word_use` table include those which cannot be redefined in the Java programming language, such as operators and keywords. In addition, there are word types in the `word_use` table which provide the information about a word that is available at the syntactic level when it is used. For example, when the `package` keyword is used, we know the subsequent word must be a package name.

4.3 Preliminary Results

We validated our data collection and storage proposals by constructing an example corpus from the 10,947 Java source files that comprise version 1.5 of the Java Developers Kit (JDK 1.5). The JDK is the standard set of libraries provided by Sun Microsystems, the creators of the Java language. These libraries are widely used among Java developers and together employ all features of the Java grammar specification. Collectively these files contain 3,148,796 lines of source code. On a MacBook Pro laptop computer with a 2.33 GHz Intel Core 2 Duo processor and 3 GB of RAM, our tools can process these files in approximately two and a half minutes.

We first consider the words that are defined in our corpus. In total, there are 503,619 word definitions. However, only 111,819 of these are unique. The counts of total word definitions and unique word definitions grouped by word type are shown in Table 2.

From these counts in Table 2 we can see, for example, that unique names are much more common for classes, interfaces, and enumerations than they are for parameters, especially type parameters. Such a result confirms what a seasoned Java programmer already knows — there are conventions that govern the naming of parameters and particularly the naming of type parameters in Java. The fact that this knowledge is had among experienced programmers does not discount the importance of developing methods that can detect it automatically.

By being able to detect conventions in programming language use with automated tools we make them more accessible to novices and thus lower the barrier to entry into a new programming language and into individual projects. Also, by recognizing such conventions automatically, we can, for example, develop style checking tools that adjust themselves to local conventions and style changes overtime thereby making themselves more useful and acceptable to developers.

Table 2. Definition Counts by Word Type

Word Type	Total Definitions	Unique Definitions
annotation	7	7
annotation_method	3	1
class	13,143	11,480
constructor	14,204	8,653
enhanced_for_element	460	151
enum	30	26
enum_constant	233	197
interface	1,854	1,738
label	173	97
method	119,294	36,381
parameter	159,342	11,812
type_parameter	477	11
var_arg_parameter	65	15
variable	193,526	60,234

We next consider the words that occur in our corpus and, in particular, the *frequency* with which each word occurs. Frequency is a common metric used in corpus linguistics as a proxy for the importance of a word. It has been shown that word use follows Zipf's law which predicts that the frequency of any word in a corpus is inversely proportional to it frequency-ordered rank within the corpus [21]. That is, the first most common word is expected to occur approximately twice as often as the next most common word and approximately three times as often as the third most common word, etc. Thus, graphs of word frequency versus rank are expected to exhibit an exponential decay.

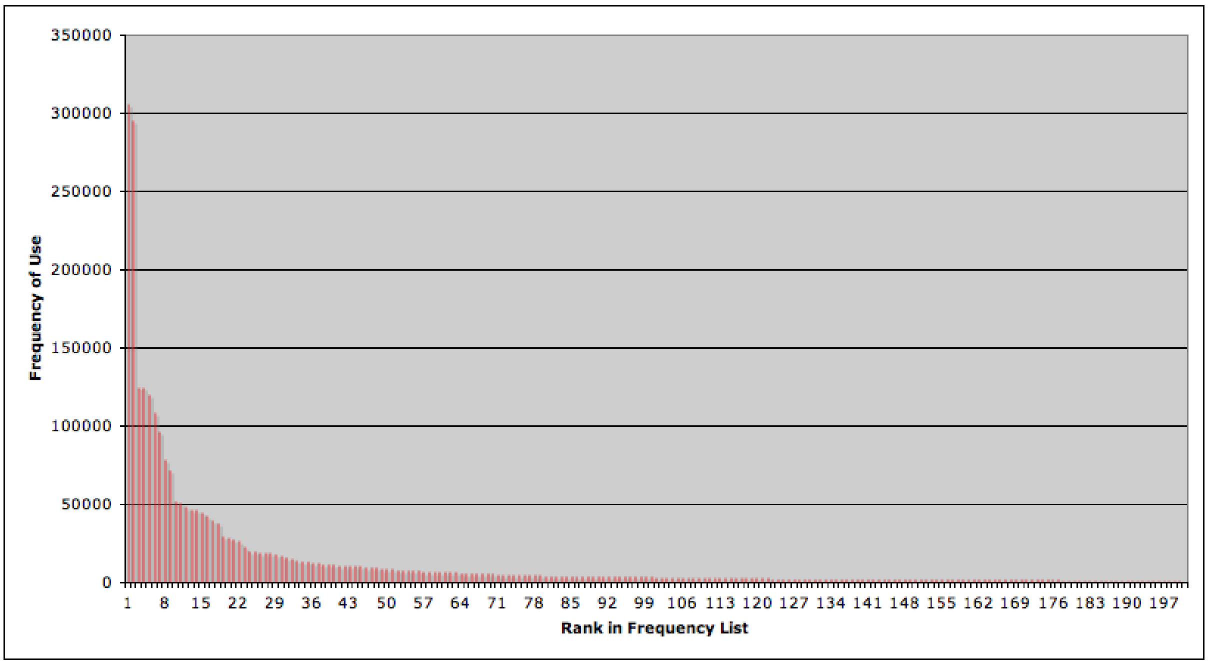


Fig. 3. Word Frequency vs Rank

There are 3,895,485 occurrences of 98,879 unique words in the JDK 1.5. The rank versus frequency graph for the 200 most frequent of these words is shown in Figure 3. This graph clearly follows Zipf's law adding support to our claim that the use of programming language follows the patterns of natural language.

As with natural language where the function words are the most frequent, the most common words in our corpus all belong to the closed set of the language. The most commonly occurring word is the `+` operator (305,685) followed by the scoping block (295,726) and the `=` operator (124,813). If we exclude operators and scoping blocks from our analysis, the most frequent words are `public` (124,399), `if` (119,787), and `int` (108,709).

The most common identifier (the programming language equivalent of a lexical word in natural language as discussed in Section 3.2), is `String`. It is the ninth most frequently occurring word overall with 71,504 occurrences. This pseudo-primitive type in Java is a special case of a non-primitive that has nearly achieved primitive status in the language and may well do so in either a future version of Java or a derivative language it spawns. The next three most frequent lexical words are `length` (19,312), `Object` (18,506), and `IOException` (11,322).

Of course, the analyses we have presented here are cursory and inconclusive. However, they each give a small glimpse of the insight that could be gained from a more complete linguistic analysis of a representative programming language corpus and they begin to build support for our claim that the use of programming language mimics that of natural language.

5 Conclusion

In this paper we have presented a methodology for applying the techniques which are traditionally used by corpus linguists in the study of natural language to the study of programming language. We have defined the term "word" in the context of programming languages, discussed our data collection and storage implementations, and validated our proposals with an preliminary study.

Our initial results indicate that the use of programming language does indeed follow the same patterns as that of natural language. In particular, a few words occur with a very high frequency while many words occur with very low frequency.

Using our example corpus and a simple analysis, we are able to identify important behavioral patterns which may be taken for granted by fluent users of a programming language but remain hidden to novice users.

The preliminary results in this paper are just a few examples of the insights we may gain about programming language by learning from linguists. There are many more linguistic results that we can reasonably expect will extend to programming language. For instance:

– *Language Acquisition* – Linguists have developed criteria, based on analyses of word frequency in natural language corpus, for assigning levels to texts. Using these assigned levels as a guide, language learners (whether they be students learning a first language or those learning a second language) can progress from simpler texts to more complex texts as their ability with the language increases, thus easing their acquisition of the language. Similar criteria for programming language texts could be used to select source files to help teach students in various stages of a Computer Science curriculum or to include in various sections of a programming language's documentation.
– *Language Assessment* – Standardized language assessment tests such as the TOFEL are common for natural languages. Similar tests predicated on word frequency counts for programming languages could be used as part of a course evaluation or interview process.
– *Language Reference* – Corpus linguists create frequency dictionaries of natural language to allow readers to quickly determine not only the meanings of a word, but also the relative frequency with which each meaning occurs. Similar reference materials for a programming language or for an individual project could help new developers better target their searches as they become familiar with a new code base.

The necessary first step in extending these or other linguistic results to programming language is a demonstration that there are clear parallels between the use of programming language and the use of natural language. To accomplish this, we must create corpora of programming languages as described here and analyze those corpora using the techniques of corpus linguistics.

References

1. Giuliano Antoniol, Yann-Gael Gueheneuc, Ettore Merlo, and Paolo Tonella. Mining the lexicon used by programmers during sofware evolution. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 14–23, 2007.
2. Pierre Arnaud and Henri Bjoint. *Vocabulary and applied linguistics.* Macmillan, Houndmills, Basingstoke, Hampshire, 1992.
3. Gusti Astika. Analytical assessments of foreign students' writing. *RELC Journal*, 24(1):61–70, June 1993.
4. Douglas Biber, Susan Conrad, and Randi Reppen. *Corpus linguistics: Investigating language structure and use.* Cambridge University Press, Cambridge, UK, 1998.
5. Fredrick Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison Wesley, Boston, MA, 1995.
6. M. Davis. *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions.* Dover Publications, Mineola, N.Y., 2004.
7. Dee Gardner. Validating the construct of word in applied corpus-based vocabulary research: A critical survey. *Applied Linguistics*, 28(2):241–265, June 2007.
8. Evelyn Hatch and Cheryl Brown. *Vocabulary, semantics, and language education.* Cambridge language teaching library. Cambridge University Press, Cambridge; New York, 1995.
9. Batia Laufer. The development of passive and active vocabulary in a second language: Same or different? *Applied Linguistics*, 19(2):255–271, June 1998.
10. Dawn Lawrie, Henry Feild, and David Binkley. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.
11. Dawn Lawrie, Henry Feild, and David Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388, 2007.
12. Tony McEnery and Andrew Wilson. *Corpus linguistics.* Edinburgh University Press, Edinburgh, 1996.
13. K Mezynski. Issues concerning the acquisition of knowledge: Effects of vocabulary training on reading comprehension. *Review of Educational Research*, 1983.

14. Peter Naur. Programming languages, natural languages, and mathematics. *Communications of the ACM*, 18(12):676–683, 1975.
15. Peter Naur. *Programming Languages are not Languages: Why 'Programming Language' is a Misleading Designation.* Addison-Wesley, Reading, MA, 1991.
16. T. Parr. *The definitive ANTLR reference: Building domain-specific languages.* Pragmatic Bookshelf, Raleigh, N.C., 2007.
17. David Qian. *Depth of vocabulary knowledge : Assessing its role in adults' reading comprehension in English as a second language.* Department of Curriculum, Teaching and Learning, Ontario Institute for Studies in Education of the University of Toronto, Toronto, 1998.
18. Norbert Schmitt. *Vocabulary in language teaching.* Cambridge language education. Cambridge University Press, Cambridge; New York, 2000.
19. A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 4(143):167, 1996.
20. Martin Wynne. *Developing Linguistic Corpora: A Guide to Good Practice.* Oxford: Oxbow Books, Oxford, UK, 2005.
21. G. Zipf. *Selective Studies and the Principle of Relative Frequency in Language.* MIT Press, Cambridge, MA, 1932.