修 士 論 文

題　目　　ソースコード中の識別子の語長や単語数の傾向が
品質に与える影響の分析

主任指導教員　　辻野　嘉宏　教授

指導教員　　水野　修　准教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　11622010

氏　　名　　川本　公章

平成25年2月8日提出

様式 2 号

学位論文の要旨（和文）

平成 25 年 2 月 8 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 23 年入学
学生番号　　11622010
氏　　名　　川本　公章　㊞

（主任指導教員　　辻野　嘉宏　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

ソースコード中の識別子の語長や単語数の傾向が品質に与える影響の分析

2.　論文内容の要旨（400 字程度）

関数名や変数名などの識別子は，ソースコードを理解するための重要な情報源となっている．識別子の命名方法はソフトウェアの理解に多大な影響を及ぼし，その結果として，ソースコードの品質を左右していると考えられる．本研究では，識別子命名の要素として識別子の語長と単語数に着目し識別子命名の傾向の分析とソフトウェアの不具合予測の実験を行った．識別子の語長は，ソースコードから識別子のみを抽出し，識別子の文字数を計算する．識別子の単語数は，識別子の単語の組み合わせ方の特徴を利用して識別子の文字列を単語ごとに分割し単語数を計算する．オープンソースソフトウェアの 7,896 プロジェクトから大量の識別子を抽出し分析することで，ソフトウェア開発者の識別子命名の傾向を知ることができた．不具合予測実験の結果，識別子の語長と単語数を用いて不具合有無の予測がある程度可能であることがわかった．

On the impact of length of identifiers to software quality.

2013 11622010 *Kimiaki KAWAMOTO*

Abstract

Identifiers such as variable names and function names in source code are essential information to understand code. Identifiers are developers' main source of information when source code is not sufficiently commented. The naming for identifiers affects on code understandability, thus, we expect that they affect on software quality. In this study, we focus on length of identifier and number of words. Analyzing number of words requires splitting identifiers into their constituent words. The way of splitting identifiers is to apply programming language naming conventions. Identifiers are extracted from source codes in 7,896 projects of open source software. We investigate the trends of length of identifier and number of words. We examine the relationship between the length of identifiers and existence of software faults in a software module. The results of experiment using the random forest technique show that there is a positive relationship between the length of identifier and existence of software faults.

To extract the word information from identifiers, we implemented a splitter of identifiers.

In experiments,

目　次

1. 緒言

プログラムで使われている関数名や変数名などの識別子は，識別子を構成する単語の意味を読み取ることで，ソースコード中のコメント文と同じようにソースコードを理解するために重要な情報源となっている [1]．また，識別子の語長や単語の組み合わせ，単語の省略の方法などといった，識別子の命名がソースコードの品質や可読性に影響を与えている [2][3]．識別子は，ソースコードを構成する要素の中でも重要なものであり，識別子を解析することでソフトウェアの開発に対する有益な情報が得られるのではと考えられる．

そこで，本研究では，識別子命名の要素のである識別子の語長と識別子を構成する単語の数に対して分析と実験を行う．まず，オープンソースソフトウェアのソースコードを大量に取得し，そこに含まれる識別子の語長と単語数を計算し，語長毎の識別子の数と単語数毎の識別子の数を集計する．プログラミング言語毎に識別子の傾向を調べることで，ソフトウェア開発者の識別子命名の傾向について分析を行う．

識別子命名の要素はソースコードの品質に影響するため，ソフトウェアの不具合へ影響すると考えられる．したがって，識別子の語長と単語数という識別子命名の要素を使ったソフトウェアの不具合予測が可能ではと考えられる．不具合を含むモジュールと含まないモジュールそれぞれに対して，モジュールに含まれる識別子の出現回数を語長毎と単語数毎に調べ，機械学習を用いて識別子の出現回数の分布から不具合有無を判別するモデルを構築する．新たなモジュールの不具合有無を，識別子の出現回数の分布にモデルを適用する事で判別し，その結果から，識別子の語長や単語数と不具合出現の関係について分析する．

ソフトウェア開発において不具合を含んでいそうなモジュール（fault-prone モジュール）を予測することは，効率的なソフトウェア開発を行うことを可能とし，開発コストの削減に役立てることが可能である．. 本研究の識別子の語長の分布を用いた不具合判別は，ソースコードを書いているときに，そのソースコードに含まれる不具合の予測を可能とするため，より効率的なソフトウェア開発が可能になると考えられる．

先行研究として，識別子の語長を用いたソフトウェアの不具合予測について分析し，識別子の語長による不具合の予測がある程度可能であることがわかっている

[4][5][6]．本研究では，次に示す研究設問を設定し，識別子の語長と単語数について分析を行う．

RQ1 ソフトウェア開発者は何文字の識別子をよく用いるか

RQ2 ソフトウェア開発者は識別子に幾つ単語を含ませるか

RQ3 ソースコード中で最も出現回数の多い識別子の語長と単語数は何か

RQ4 よく識別子に使われる単語は何か

RQ5 識別子の語長を用いてソースコードの品質を予測できるか

RQ6 識別子を構成する単語の数を用いてソースコードの品質を予測できるか

RQ7 識別子の語長や単語数を用いた予測と他の不具合予測ではどちらが良いか

本論文の以降の構成を示す．第 2 節では，識別子の重要性について詳しく説明する．第 3 節では，ソースコード中の識別子の抽出や分割，識別子を用いた不具合予測などの提案手法を説明する．第 4 節では，ソースコード中の識別子の分析と不具合予測の実験の方法と結果を述べる．第 5 節では，実験の結果に対する考察を述べる．第 6 節では，ソースコード中の識別子について研究している関連研究を述べる．第 7 節では，本研究のまとめと今後の課題について述べる．

2. 識別子の重要性

ソースコードのおよそ 70%は識別子で構成されており [1]，ソースコードを解析する時に識別子に注目する事は重要だと考えられる．識別子は予約語やリテラルでない関数名や変数名などの文字列の事であり，英単語を組み合わせた文字列になっていることが多い．

関数の処理を表す単語を組み合わせて関数名を作ることで，関数に対する注釈（コメント文）を読まなくても関数の処理内容を読み取ることができ，関数呼び出しで記述されている時も関数宣言を見なくても処理内容を知ることが可能となる．変数においても，格納する値の内容や型情報を表す単語を組み合わせて変数名を作ることで，ソースコード内で変数が使われている時に処理の内容を解りやすくできる．このように，識別子はソースコードの処理内容を理解するための情報を持たせることが可能であり，コメント文と同じようにソースコードを読むときの重要な情報源となり得る．

ここで，複数の単語を組み合わせた識別子を人間が読む場合は，適切な単語の区切りを読み取り単語ごとに分割することができるが，機械的に単語を認識するには問題が生じる．識別子は空白文字を含む事ができないため，複数の単語を組み合わせた場合も一つの文字列となってしまい単語の区切りが曖昧になってしまう．この問題に対して識別子の分割・展開を行う手法がいくつか研究されている [7][8][9][10][11][12]．

ソースコードの品質や可読性は，モジュールの不具合出現への影響があると考えられ，識別子がソースコードの品質や可読性に影響を与えるということは，識別子がモジュールの不具合出現に影響を与えることがあるのではないかと考えられる．

識別子を構成する文字列の長さ（語長），単語の組み合わせ方，文字の並び，大文字と小文字，単語の省略の方法などといった要素は，プログラマー毎の識別子の命名における違いとなり，それらがソースコードの品質や可読性に影響を与えている [2][3]．したがって，識別子の良い命名規則を作ることは，ソースコードの品質を高めることに繋がると考えられる．

ソースコードの品質や可読性に影響を与える識別子命名の要素として以下に示すようなものがある．

- 識別子の語長

- 単語の数
- 単語の省略形や頭文字
- ハンガリー記法
- 大文字と小文字
- 数字や数の単語

識別子の語長と識別子を構成する単語の数は，識別子の可読性や情報量を左右し，単語の数が少なければ識別子の持つ情報は減るが，語長が短くなり読みやすくなる．単語の数が多ければ識別子の持つ情報は増えるが，語長が長くなり読みやすさは下がる．

単語の省略形や頭文字によって，よく使われている単語を省略して記述することで識別子の語長を短くすることが可能となるが，コードを読む者がプログラミングやソフトウェア開発に精通していない場合は可読性が下がることもある．省略の方法もプログラマによって異なることがあり，ソースコードの可読性を大きく左右する要素となる．

ハンガリー表記は，識別子に型情報やスコープ範囲を表す特別な接頭文字を付ける命名法である．これにより，識別子の情報を簡単に得ることが可能となるが，表記の統一を行わなければソースコードの可読性を下げることもある．

大文字と小文字は，識別子の見た目を左右する要素であり，ソースコードの可読性に影響を与える．単語の先頭を大文字にして複数の単語を組み合わせるキャメルケースや，複数の単語のを組み合わせた “XML” や “DB” といった単語は，大文字と小文字の使い分けに意味を持たせている．

数字や数の単語は，同じような識別子を新たに作るときに，命名の負担を減らすために数字を付加して新しい識別子とすることがあるが，これは間違った識別子の指定の原因にもつながり，ソースコードの品質を下げることになる．また，数字を単語の省略形として単語の組み合わせに使うこともある．例として，“csv2xml” という識別子は “csv to xml” という意味を持ち，ここでは “to” を “2” と省略し単語を組み合わせている．

本研究では，識別子命名の要素の中で識別子の語長と識別子を構成する単語の数について注目している．識別子の語長と単語数は，言語に関係なく値を得ることが

可能であり，ソースコードから手軽に測定することができる要素である．また，識別子の省略や数字の利用などの要素は，識別子の語長や単語数にも影響しており，語長と単語数の分析は他の命名要素が持つソースコードへの影響も取り入れた分析を可能にすると考えられる．

3. 提案手法

本研究では，ソースコード中の識別子を対象とした分析手法として，ソースコードから識別子を取り出す識別子の抽出手法，識別子が複数の単語から構成させている場合に単語毎に分ける識別子の分割手法，識別子の抽出・分割から得られた情報からソースコードの不具合の予測を行う識別子の語長や単語数を用いた不具合予測の手法について提案を行う．

3.1 識別子の抽出

識別子はソースコード中に含まれる予約語やリテラルでない関数名や変数名などの文字列の事で，開発者が自由に決めることのできる文字列となっている．

ソースコードは以下に示す要素で構成されていると考えることができ，ソースコードから識別子以外の要素を取り除くことで，識別子だけを抽出することが可能となる．

- 空白文字
  空白（スペース）だけでなく，タブ文字や改行コードなどの文字であり，解析の際には文字列の区切りとなる．
- 演算子
  加減算，剰余算，ビット演算や代入などを表す記号で，識別子に用いることができない文字となっていることが多く，空白文字と同じ用に文字列の区切りとなる．
- リテラル
  ソースコード中にある定数で，数値や記号で囲まれた文字列である．文字列を囲む記号はプログラミング言語毎に異なる場合があり，複数行にまたがる場合など注意が必要となる．
- コメント
  特定の記号で囲まれた文字列や記号の後に書かれた文字列はコメントアウトとなり，コンパイル時に無視されるため，識別子を含むことはない．コメントアウトに用いる記号はプログラミング言語ごとに異なるため注意が必要である．

- キーワード

  ソースコード中に直接書かれる文字列の中で制御構文に用いる “if” や “for” など，特別な意味を持つ文字列をキーワードまたは予約語という．キーワードはプログラミング言語毎に異なった文字列となる事があり，解析には注意が必要である．
- 識別子

  ソースコード中に直接書かれる文字列の中でキーワード以外の文字列である．

識別子の抽出は，ソースコードを字句解析する必要があるが，構文解析まで行う必要は無く，ソースコードに対して比較的容易に実行できる処理といえる．

3.2 識別子の分割

識別子は複数の単語を組み合わせた文字列になっていることが多く，組み合わせ方針は大きく 2 つある．

- 記号・数字を利用した単語の連結

  単語の連結にはアンダーバー記号 “_” がよく用いられる．記号を用いた単語の連結では “print_buffer_size” のように，空白文字を記号で置き換えて連結という手法が用いられる．例えば，“print_buffer_size” は “print buffer size” という 3 つの単語の組み合わせである．数字を用いた単語の連結では，数字の前後に単語を連結する “csv2xml” のような場合や，識別子の最後に数字をつける場合がある．いずれの場合も数字の前後で分割を行い，数字も一つの単語として残す．したがって，“csv2xml” は “csv 2 xml” に分割でき，3 つの単語の組み合わせとする．
- 単語をそのまま連結

  識別子の文字列の中に記号や数字を含まないもので，“printBufferSize” のような文字列となり，大文字と小文字を利用した分割や単語を認識した分割などが考えられている．

記号を含む単語の分割では，識別子の文字列から記号を削除しその場所を単語の分割点とする．

数字を含む単語の分割では，数字の前後を単語の分割点とする．しかし，“mp3”，“win32” といった数字を含む単語もあり，単純な分割を行うことで元の単語の意味を損なう場合があり注意が必要である．

単語をそのまま連結した文字列は次の 4 つの場合がよく使われている．

1. 単語の最初の文字だけ大文字にして連結

   “getString” や “setPoint” のように単語を連結した識別子で，大文字の直前で単語を分割することができ，“get string”，“set point” にそれぞれ分割できる．
2. すべてが大文字の単語と連結

   “getMAXString” や “GPSState” のように単語を連結した識別子で，大文字と小文字が連続する直前で単語を分割することができ，“get MAX String”，“GPS State” にそれぞれ分割できる．
3. すべてが小文字で連結

   “databasefield” や “notype” のように小文字だけで構成された文字列で，単語を認識して分割する必要があり，今回の例では “database field”，“no type” にそれぞれ分割できることが望ましい．
4. すべてが大文字で連結

   “USERLIB” や “COUNTRYCODE” のように大文字だけで構成された文字列で，単語を認識して分割する必要があり，今回の例では “USER LIB”，“COUNTRY CODE” にそれぞれ分割できることが望ましい．

1 と 2 の連結方法はキャメルケース（CamelCase）と呼ばれる識別子の命名方法で幅広く取り入れられており，人が見ても計算機による解析でも容易に単語の分割を行うことができる．

3 と 4 は文字列の中に分割の目印となる要素が少なく，文字列の中に存在する単語の組み合わせを予測して分割を行う必要がある．正確な分割を機械的に行うことは難しいが，いくつかの手法が提案されている [8][10][11][12].

実際の識別子の命名では，記号を用いた単語の連結やキャメルケースによる連結がほとんどの場合で実施されており，3 と 4 のように分割が難しい識別子を無視してもかなり高い精度で識別子の分割ができると考えられている [11].

3.3 識別子の語長や単語数を用いた不具合予測

識別子の語長と識別子を構成する単語の数は，開発者の識別子命名方法に影響される要素であり，第 2 節で説明したように，識別子命名の要素がソースコードの品質に影響し更にソフトウェアの不具合へ影響すると考えられ，識別子の語長と単語数という識別子命名の要素を使ったソフトウェアの不具合予測を提案する．

不具合予測はソースコードのファイル単位で行い，ファイルが含む全ての識別子の語長と単語数を取得し，語長と単語数のそれぞれの出現回数を計算し正規化してその出現分布を用いて機械学習を行い，語長と単語数による不具合予測のモデルを構築する．ソースコードのファイル単位での不具合有無の情報は事前に他の手段で解析できているものとして，教師ありの機械学習を行う．

識別子の語長と単語数の出現回数は，ソースコードが含む識別子の数に比例し，その値は不具合予測で頻繁に用いられるコード行数と強い正の相関を持つ．したがって，出現回数をそのまま利用して不具合予測モデルを構築するとコード行数の影響を受けたモデルになることが考えられる．そこで，語長や単語数毎の出現回数をソースコードが含む識別子すべての出現回数で割り，語長や単語数を正規化することでコード行数の影響を排除した不具合予測モデルが構築できると考えられる．

4. 適用実験

大量のソースコードから識別子を抽出し，識別子の語長と識別子が含む単語数の傾向の分析を行う．更に，識別子の語長と単語数による不具合予測を行い，識別子命名の要素がソースコードに与える影響を分析する．

4.1 実験対象

識別子の語長と識別子が含む単語数の傾向の分析は，インターネット上でソフトウェア開発プロジェクトを共有できる GitHub (注 1) というウェブサービスから世界中で開発されているオープンソースソフトウェアのソースコードを取得し分析の対象とする．

識別子の語長と単語数による不具合予測の実験は，PROMISE と呼ばれるソフトウェア工学の実験のために公開されたデータセットからオープンソースソフトウェアの不具合有無の情報などを取得し実験の対象とする [13].

4.1.1 GitHub リポジトリ

GitHub は，Git と呼ばれるバージョン管理システムのリポジトリを中心にソフトウェア開発プロジェクトの共有サービスを展開している．個人的なソフトウェア開発から有名なオープンソースソフトウェア開発が GitHub を通して世界中に公開されており，利用者の多い開発コミュニティとなっている．

Git は分散型のバージョン管理システムであるため，コマンド一つでサーバ上にあるソフトウェアの開発履歴が全て詰まったリポジトリをローカルに保存することができ，プロジェクトの過去のソースコードにもサーバへのアクセスを行わずに高速に参照する事ができる．

本実験では，GitHub に公開されているプロジェクトの中で，プロジェクトへの注目度の多いプロジェクトから順にリポジトリを収集していき，7,896 プロジェクトを対象に分析を行う．

識別子の抽出には，プログラミング言語の種類が影響するため，GitHub で公開さ

(注 1)：https://github.com/

れている使用言語の割合(注 2)でコード行数が多い言語 5 つを選び分析の対象としている．

分析対象のプログラミング言語を表 4.1 に示す．C 言語と C++言語は，プログラミングスタイルが似ており，更に，構文がほとんど同じで，識別子を同じ処理で抽出できるため，一つにまとめて分析を行なっている．

GitHub での割合は，GitHub 内で各言語が占めるファイルのコード行数の割合を示す．拡張子は，言語毎に分析の対象とするファイルの拡張子を示す．ソースコードがどのプログラミング言語で書かれているのかは，ファイルの拡張子で判断する．プロジェクト数は，分析対象の 7896 のプロジェクトの中で，各言語のソースコードを含むプロジェクトの数を示す．プロジェクトは，複数の言語のソースコードを有することがあり，一つのプロジェクトが複数の言語のプロジェクト数にカウントされている場合がある．また，分析対象のプロジェクトの中で，分析を行うプログラミング言語のソースコードを含まないものも存在した．

表 4.2 にプロジェクトが管理しているファイル数の統計情報を言語毎に示す．

どの言語も 80 万を超えるソースコードを分析対象としており，分析から一般的なデータが取れると考えられる．プロジェクトが有するソースコードの数は，少ないものは 1 つしかないものや，多いものでは 2 万を超えるものもあり，小規模なプロジェクトから大規模なプロジェクトまで分析対象に入っていることがわかる．Java 言語を使用したプロジェクトは比較的規模の大きいものが多く，Javascript 言語を使用したプロジェクトは比較的規模の小さいものが多い傾向がある．

表 4.3 にプロジェクトが管理しているソースコードのコード行数の統計情報を言語毎に示す．コード行数は空行も含めた値となっている．

どの言語もコード行数が一億を超える量となっており，膨大な数の識別子を抽出できると考えられる．C と C++言語と Python 言語のソースコードは比較的規模の大きいものが多く，Ruby 言語のソースコードは比較的規模の小さいものが多い傾向が見られる．

(注 2)：https://github.com/languages

表 4.1　分析対象のプログラミング言語

プログラミング言語	GitHub での割合	拡張子	プロジェクト数
C と C++	10%	.c，.cc，.cpp，.h	1,757
Java	8%	.java	717
Javascript	21%	.js	3,831
Python	8%	.py	1,481
Ruby	13%	.rb	1,791

表 4.2　プロジェクトが管理しているファイル数の統計

プログラミング言語	最小値	中央値	平均値	最大値	合計
C と C++	1	58.0	702.1	20,988	1,232,105
Java	1	151.5	1,343.0	22,839	961,615
Javascript	1	52.0	381.6	20,341	1,461,236
Python	1	54.0	544.1	17,518	805,316
Ruby	1	102.0	643.5	26,116	1,151,914

表 4.3　プロジェクトが管理しているソースコードのコード行数の統計

プログラミング言語	最小値	中央値	平均値	最大値	合計
C と C++	1	174.0	651.6	144,533	802,828,243
Java	1	101.0	254.1	61,762	244,385,015
Javascript	1	102.0	527.4	665,189	770,693,685
Python	1	157.0	415.1	1,705,628	334,314,435
Ruby	1	56.0	137.3	93,154	158,163,336

4.1.2 PROMISE データセット

本実験では，PROMISE で Jureczko らが公開しているデータセットを用いる [14]. データセットには，ソースコードのファイル毎に，コード行数，不具合の数，ソフトウェア工学で用いられるメトリクスの数々が用意されており，手軽に不具合予測の実験を行えるデータセットになっている．

データセットには，いくつものプロジェクトが含まれているが，その中から以下に示す 7 つのプロジェクトを実験対象とする．以下のプロジェクトは全て Java 言語を用いて開発されている．

- Apache **Ant**：ビルド自動化ツール
- **Jedit**：テキストエディタ
- Apache **Lucene**：検索エンジン
- Apache **POI**：Microsoft Office 形式のファイルの読み込み書き込みライブラリ
- Apache **Velocity**：テンプレートエンジン
- Apache **Xalan**：XML 文書の XSLT 変換と XPath 検索
- Apache **Xerces**：XML 文章のパーサ

各プロジェクトは，3 から 5 つのバージョンを持っており，データセットでは，バージョン毎にコード行数や不具合の数などの情報がまとめられている．本実験では，ソースコードが必要となるため，データセットに含まれるファイル名を用いてソースコードとのリンクを行い，識別子の情報とデータセットを結びつけている．

表 4.4 に，各プロジェクトで使用するバージョンとそのリリース日，ソースコードのファイル数を示す．図 4.1 には，ソースコードのファイル数のグラフを示す．

プロジェクトによって，使用するバージョンにメジャーバージョンアップを含むものや，バージョンが連続して居ない場合がある．ファイル数は，バージョンアップを重ねる毎に増加する傾向があるが，Xerces だけは，バージョンアップでファイル数が減少するという特殊な例となっている．

表 4.5 と図 4.2 に，各プロジェクトが管理しているコード行数の統計情報を示す．

コード数の合計はファイル数と同じ傾向となるが，コード行数の中央値や平均値に注目すると，Xalan ではバージョンアップでファイル毎のコード量も増加するが，Poi や Xerces などではバージョンアップによりファイル毎のコード量が減少している．

表 4.4　プロジェクトのファイル数

プロジェクト	バージョン	リリース	ファイル数
Ant	1.4	2001-09-03	177
	1.5	2002-07-10	292
	1.6	2003-12-18	350
	1.7	2006-12-16	741
Jedit	3.2	2001-08-29	260
	4.0	2002-04-12	293
	4.1	2003-05-24	300
	4.2	2004-12-01	355
	4.3	2009-12-23	487
Lucene	2.0	2006-05-26	186
	2.2	2007-06-17	234
	2.4	2008-10-08	330
Poi	1.5	2002-05-06	235
	2.0	2004-01-26	309
	2.5	2004-02-29	379
	3.0	2007-05-18	438
Velocity	1.4	2003-10-09	195
	1.5	2007-01-28	214
	1.6	2008-12-01	229
Xalan	2.4	2002-09-03	676
	2.5	2003-04-16	762
	2.6	2004-02-29	875
Xerces	1.2	2000-08-28	439
	1.3	2001-01-31	452
	1.4	2001-05-22	328

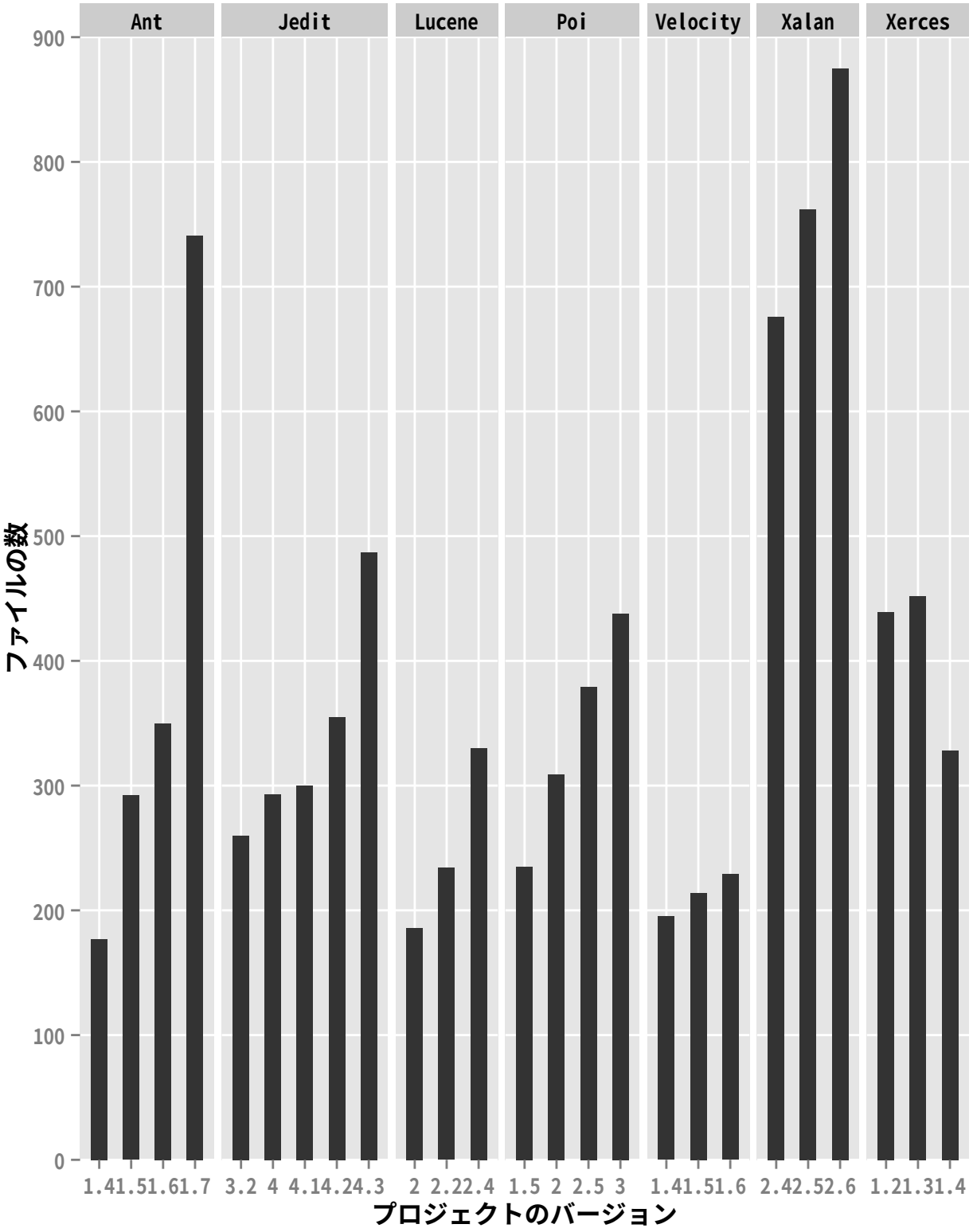

図 4.1　プロジェクトのファイル数

表 4.5　プロジェクトのコード行数の統計

プロジェクト	バージョン	最小値	中央値	平均値	最大値	合計
Ant	1.4	0	152.0	305.7	2,040	54,114
	1.5	0	144.5	297.8	4,244	86,966
	1.6	0	156.5	323.3	4,238	113,165
	1.7	0	144.0	281.2	4,541	208,354
jedit	3.2	1	173.0	484.3	23,350	125,913
	4.0	1	170.0	482.7	23,683	141,442
	4.1	1	176.0	499.5	23,590	149,848
	4.2	1	203.0	471.7	12,200	167,444
	4.3	1	175.0	411.1	12,535	200,184
lucene	2.0	1	132.0	264.2	3,709	49,149
	2.2	1	124.0	262.5	3,785	61,435
	2.4	1	123.5	306.7	8,474	101,211
poi	1.5	1	137.0	235.8	2,395	55,408
	2.0	0	141.0	300.9	9,849	92,973
	2.5	0	152.0	312.9	9,857	118,572
	3.0	0	125.0	294.8	9,886	129,143
velocity	1.4	0	92.0	264.9	10,760	51,658
	1.5	0	86.0	248.3	12,396	53,141
	1.6	0	77.0	249.0	13,175	57,012
xalan	2.4	0	139.5	323.2	3,479	218,509
	2.5	0	145.5	392.1	4,275	298,757
	2.6	0	164.0	469.9	4,331	411,125
xerces	1.2	0	42.0	362.7	8,696	159,226
	1.3	0	43.0	369.6	10,701	167,067
	1.4	0	28.0	117.0	3,050	38,359

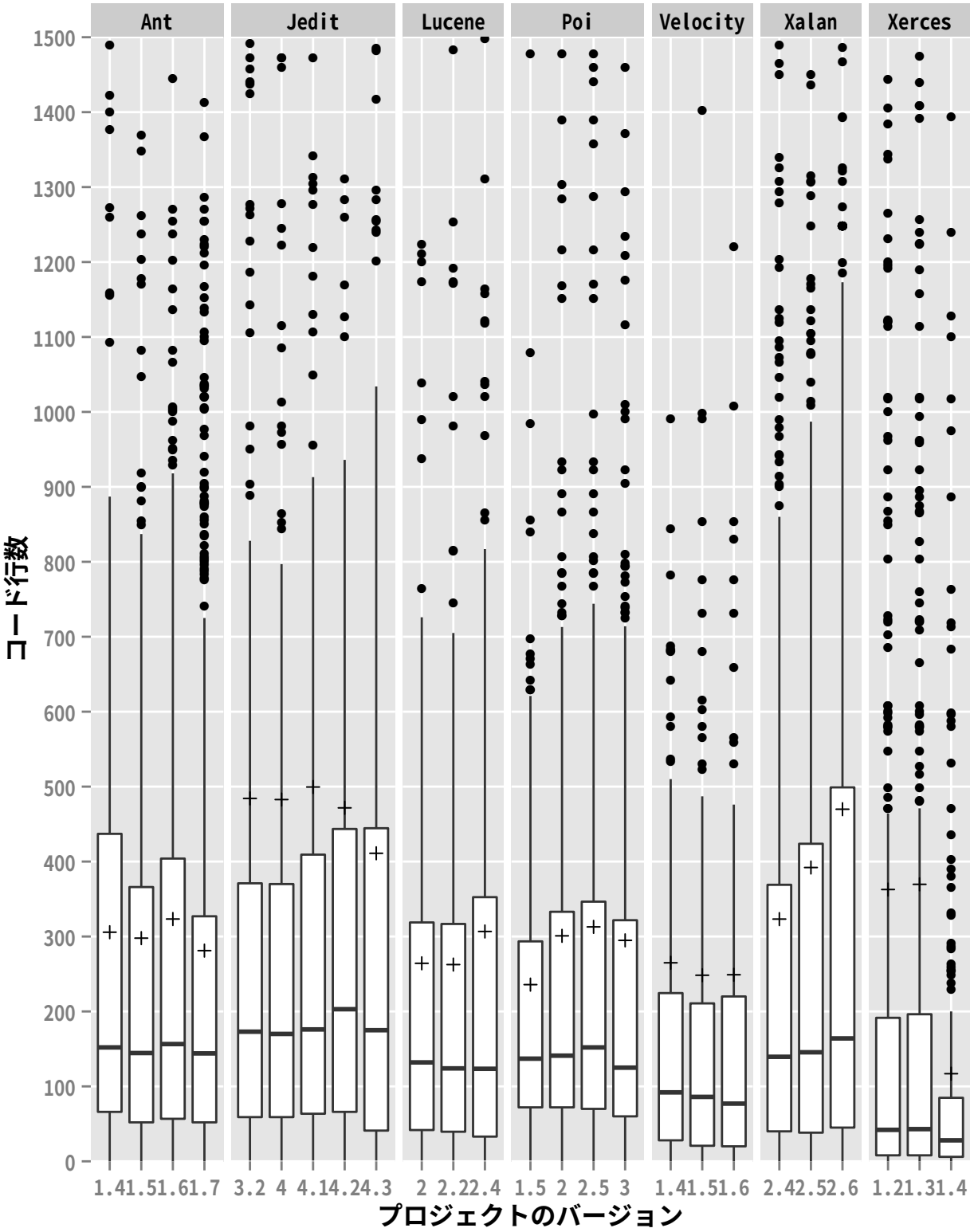

図 4.2　プロジェクトのコード行数の統計

4.2 GitHub リポジトリの分析

分析対象の Git リポジトリ 7896 個に対して，識別子の抽出と識別子の分割を行い，識別子の語長と単語数の分析を行う．

4.2.1 分析方法

分析は次に示す手順で行う．

1. Git リポジトリから分析の対象となるソースコードを取得
2. ソースコードから識別子を抽出
3. 識別子の分割
4. 識別子の語長と単語数の集計

手順 1 では，まず，Git リポジトリに対してコマンド `$ git log --pretty=format:"%h" --name-only` を実行し，コミット毎の変更されたファイル名を取得する．次に，表 4.1 に示す各言語の拡張子を利用してソースコードを取得するファイル名を調べる．最後に，Git リポジトリに対してコマンド `$ git show [commit hash]:[file path]` を実行し各ファイルの内容を取得する．

手順 2 では，第 3.1 節で解説した識別子の抽出手順を，字句解析器を生成できるソフトウェア lex で実装し，取得したすべてのソースコードに対して実行する．

手順 3 では，第 3.2 節で解説した識別子の分割のうち，アンダーバー記号を用いた単語の分割，数字を用いた単語の分割，キャメルケースを用いた単語の分割の 3 つの手順を実装し，すべての識別子に対して実行する．

手順 4 では，抽出した識別子の文字数や分割した識別子の単語の数を調べ，ソースコードのファイル毎にまとめておき，集計を行う．

4.2.2 分析結果

GitHub から取得したリポジトリに対して，プログラミング言語別に識別子の語長と単語数についてグラフや表にまとめた．

分析結果では，識別子の種類と出現回数の 2 つの数を扱う．

- 識別子の種類

同じ文字列の識別子を重複してカウントしない．“get, set, add, get, add” の 5 つの識別子がソースコード中にあるとき，語長 3 の識別子が 3 種類ある，単語数 1 の識別子が 3 種類あるとカウントする．同じ語長や単語数でどれだけ識別子に種類があるかを示す．

- 識別子の出現回数

  同じ文字列の識別子を重複してカウントする．“getNames, toString, setValue, toString, setValue” の 5 つの識別子がソースコード中にあるとき，語長 8 の識別子が 5 つ出現し，単語数 2 の識別子が 5 つ出現したとカウントする．同じ語長や単語数の識別子がどれだけプログラミング中に用いられるかを示す．

図 4.3 と図 4.4 に識別子の語長と語長毎の識別子の種類数を示す．図 4.4 では，すべての語長の種類数の合計で各語長の種類数を割り，各語長の種類数の割合を示している．

どのプログラミング言語も，語長が短いと文字の組み合わせに上限があるため種類数は少なく，語長が長くなると種類数が増える傾向がある．しかし，語長 10〜12 の辺りで種類数は最大になり，それより長い語長では種類数が減少していく．

図 4.5 と図 4.6 に識別子の語長と語長毎の識別子の出現回数を示す．図 4.6 では，すべての語長の出現回数の合計で各語長の出現回数を割り，各語長の出現回数の割合を示している．

ソースコードには，短い語長の識別子が多く書かれている事がわかる．プログラミング言語による傾向の違いもあるが，どの言語も語長 10 より長い識別子は語長が長くなるほど出現回数が減少していくことがわかる．

図 4.7 と図 4.8 に識別子の単語数と単語数毎の識別子の種類数を示す．図 4.8 では，すべての単語数の種類数の合計で各単語数の種類数を割り，各単語数の種類数の割合を示している．

どのプログラミング言語も，単語数 2 の識別子が最も多い種類があることがわかる．Javascript，Python，Ruby のスクリプト言語は，C や Java 言語に比べ単語数 1 の識別子が多く使われる傾向がある．単語数が 3 以上になると単語数が多くなるほど識別子は減少していく傾向がある．

図 4.9 と図 4.10 に識別子の単語数と単語数毎の識別子の出現回数を示す．図 4.10

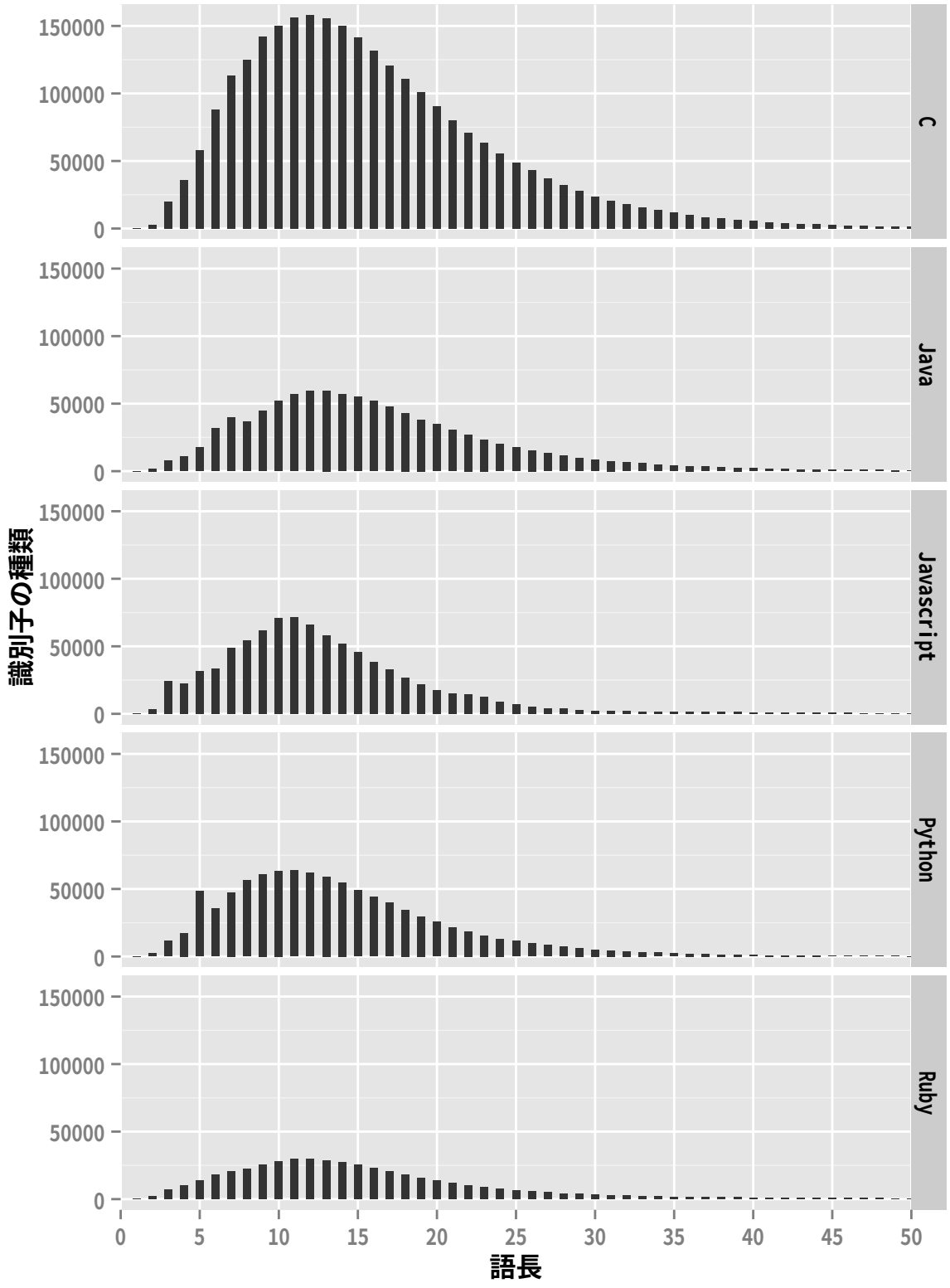

図 4.3　語長毎の識別子の種類の数

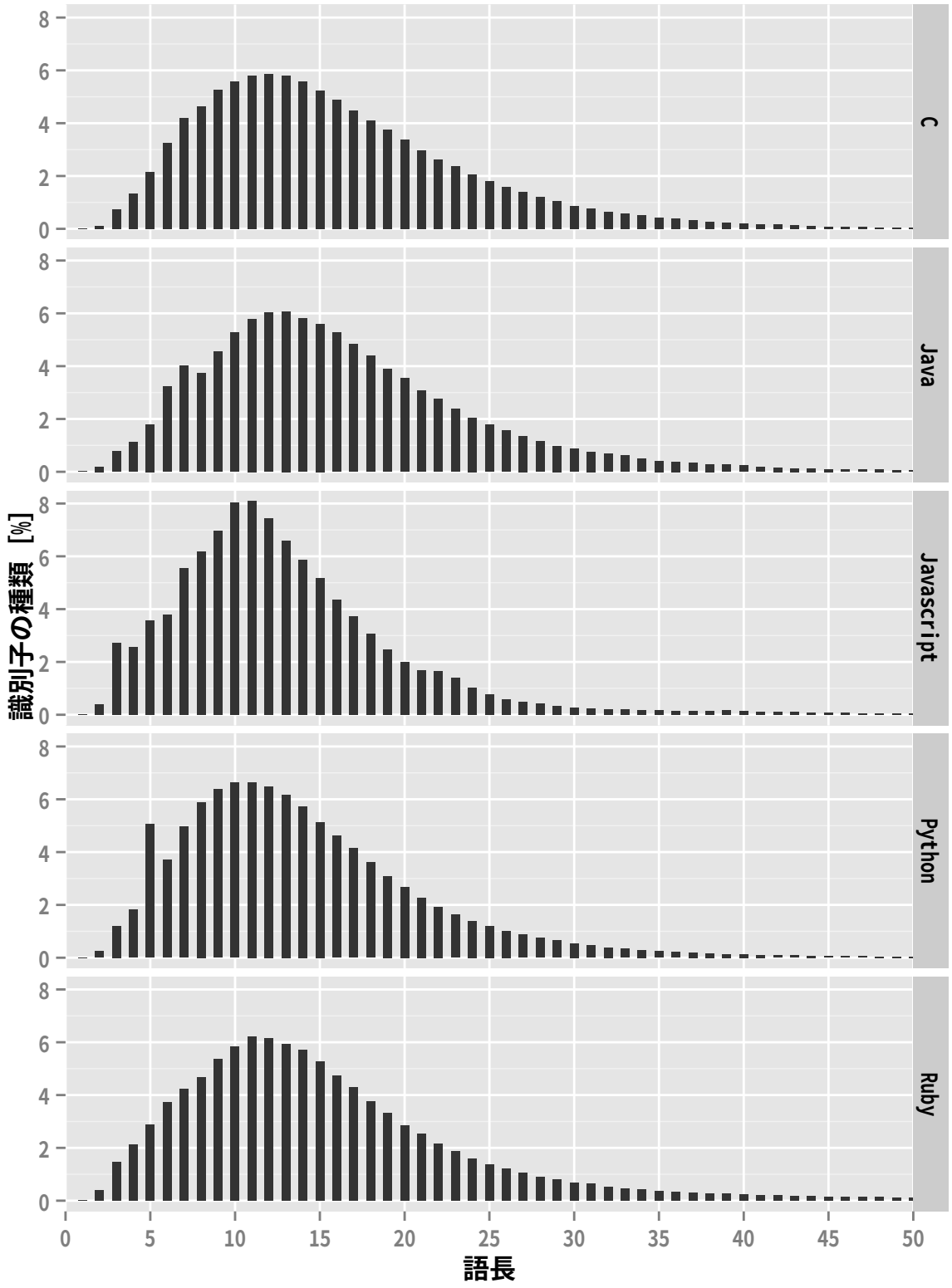

図 4.4　語長毎の識別子の種類の割合

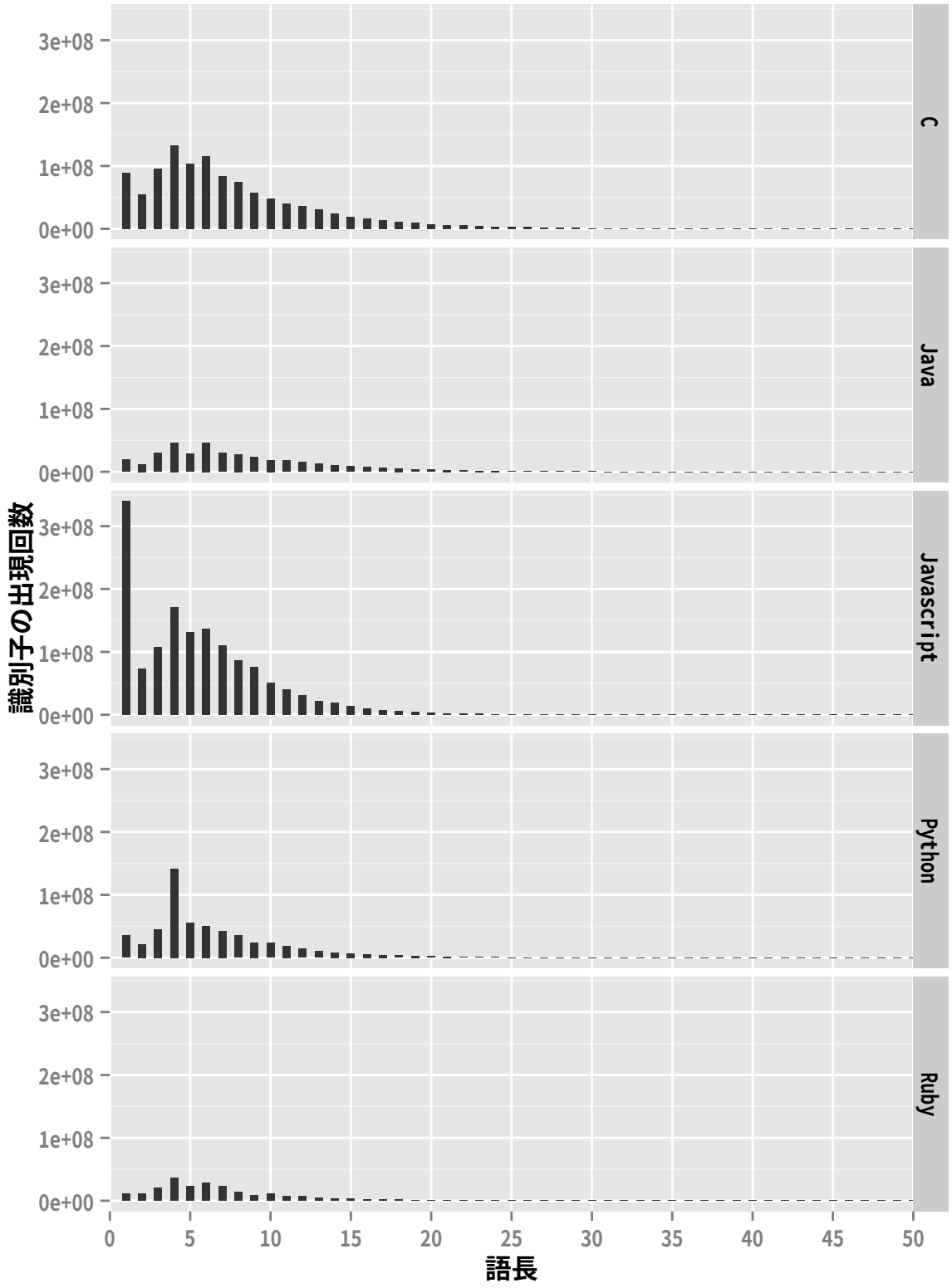

図 4.5　語長毎の識別子の出現回数

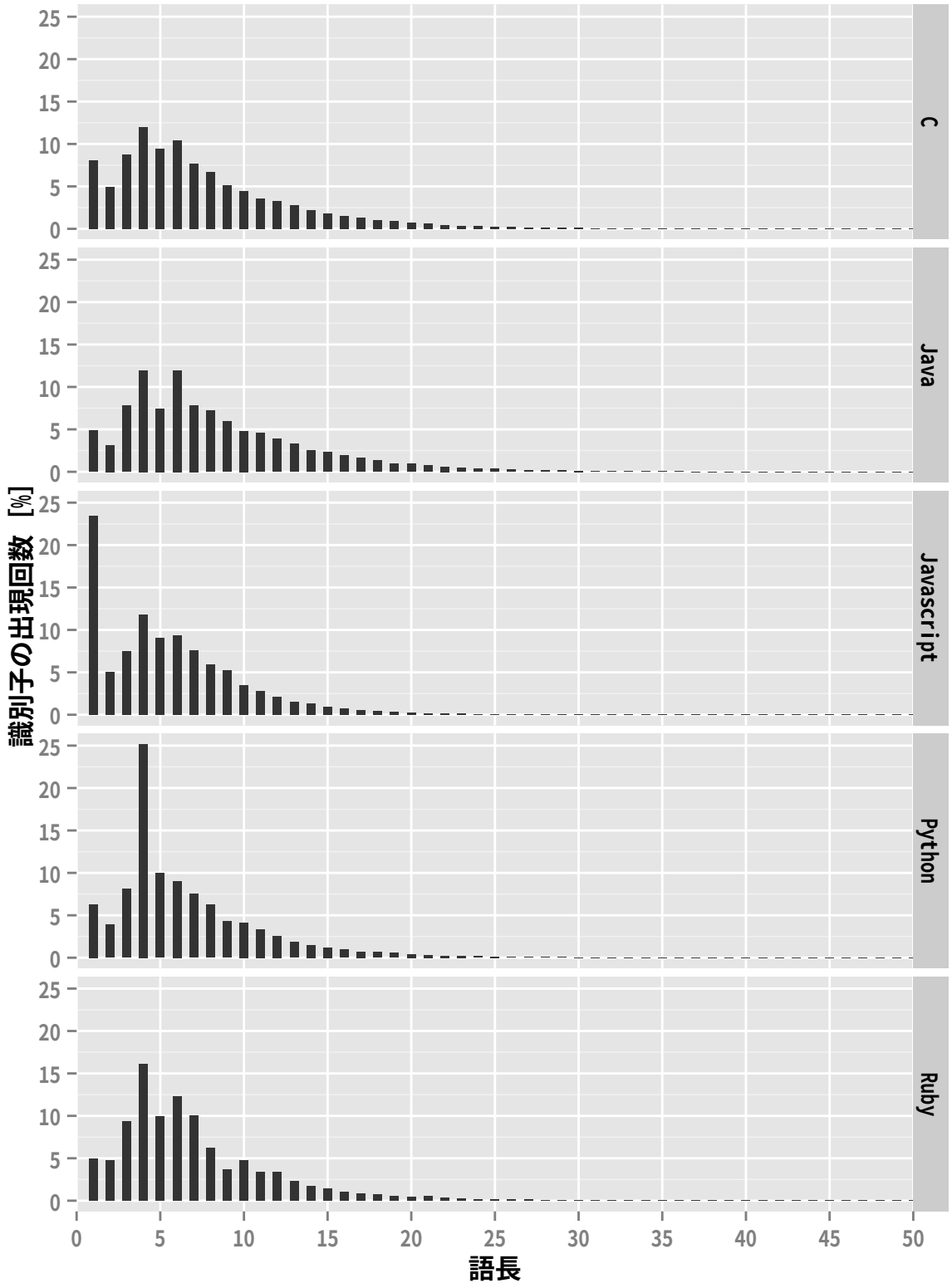

図 4.6　語長毎の識別子の出現回数の割合

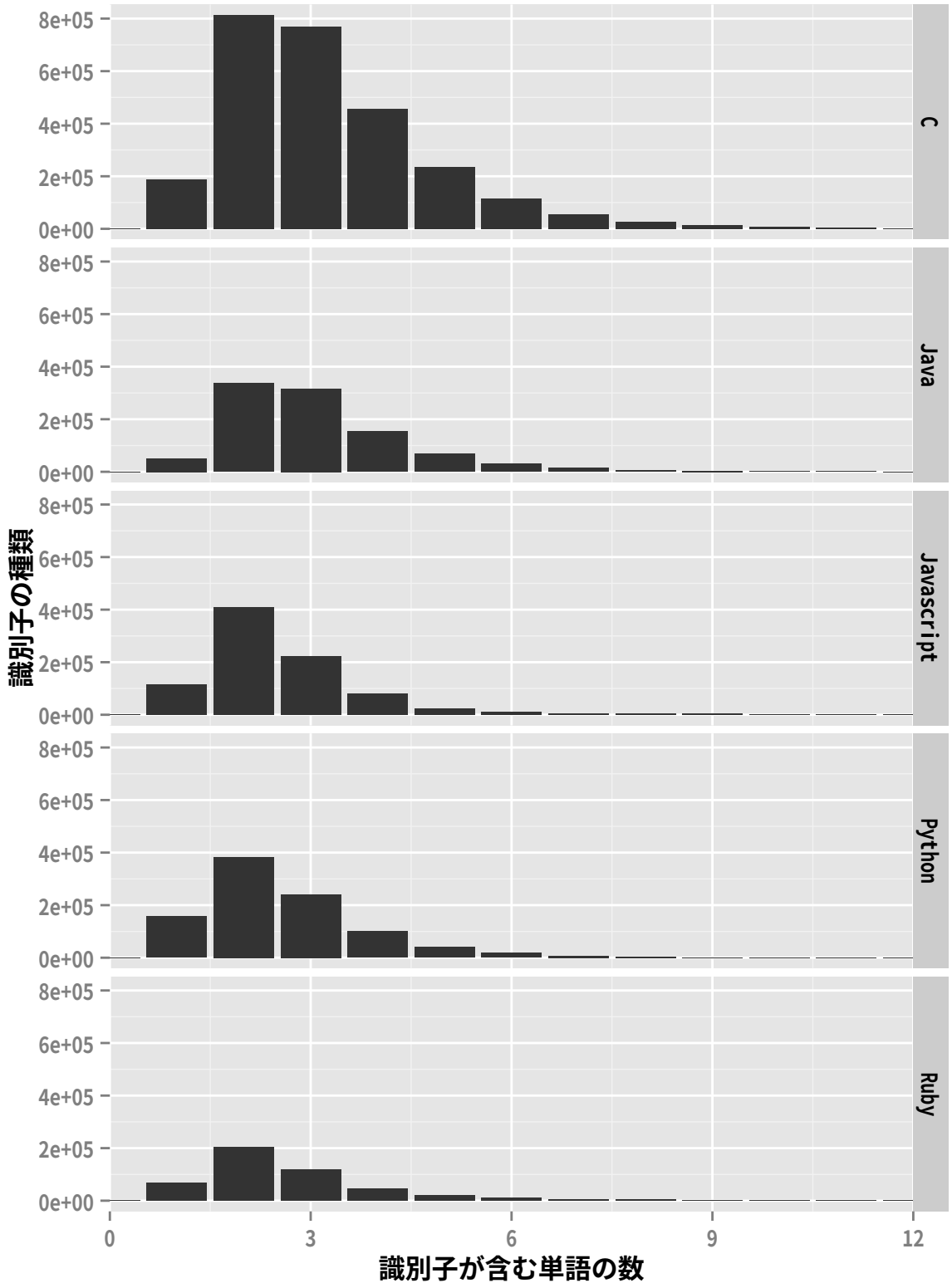

図 4.7　単語数毎の識別子の種類の数

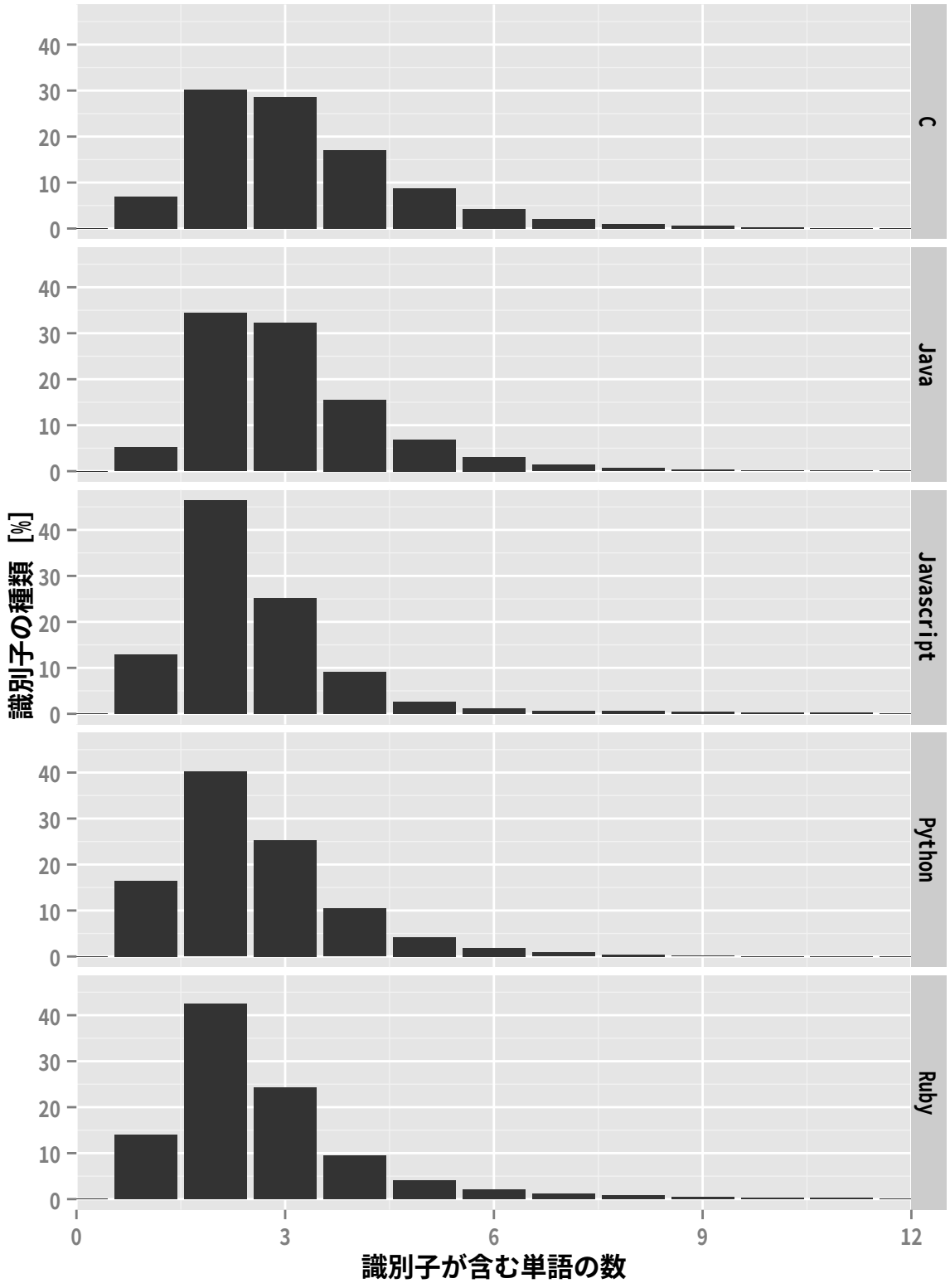

図 4.8　単語数毎の識別子の種類の割合

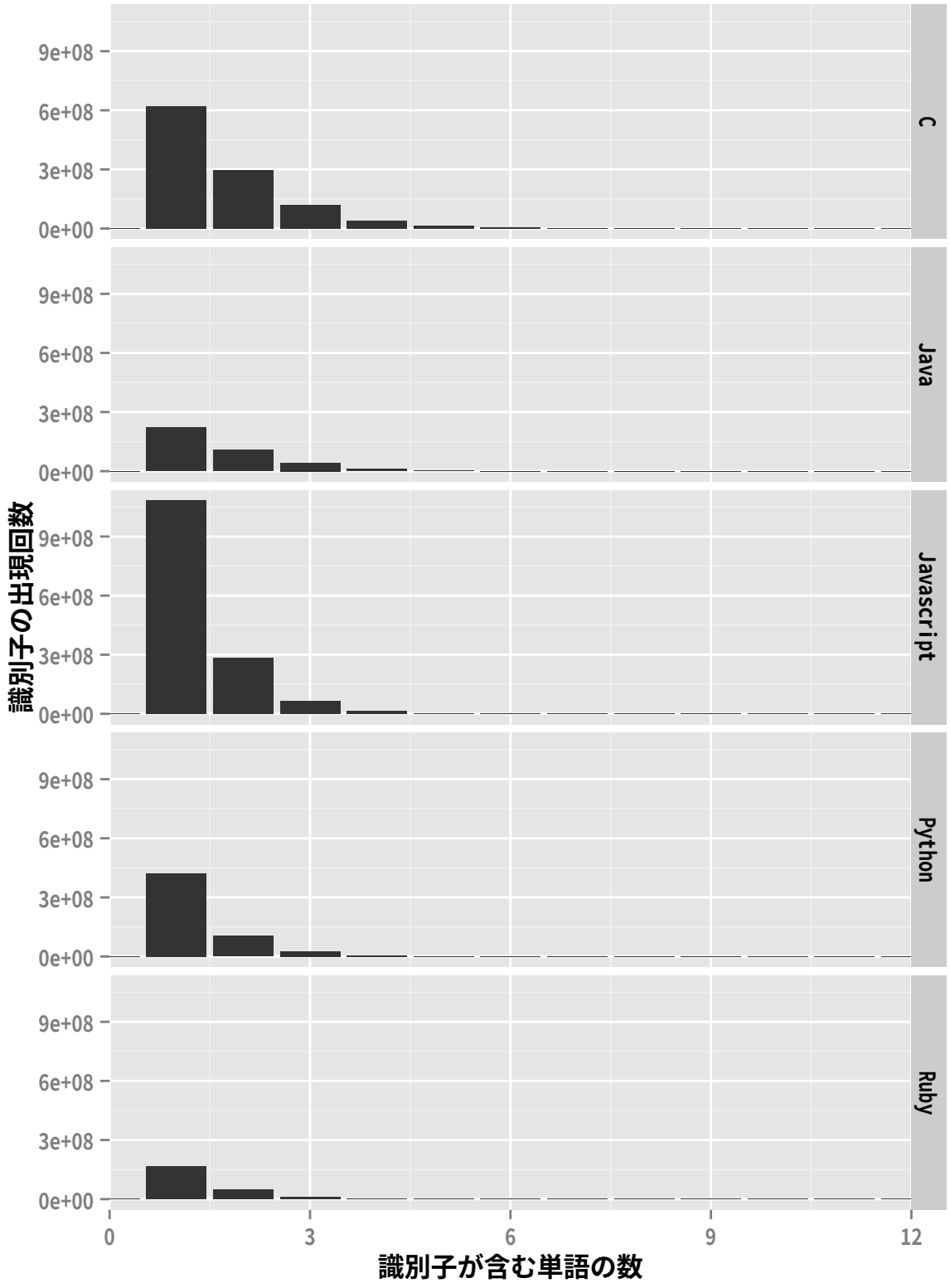

図 4.9　単語数毎の識別子の出現回数

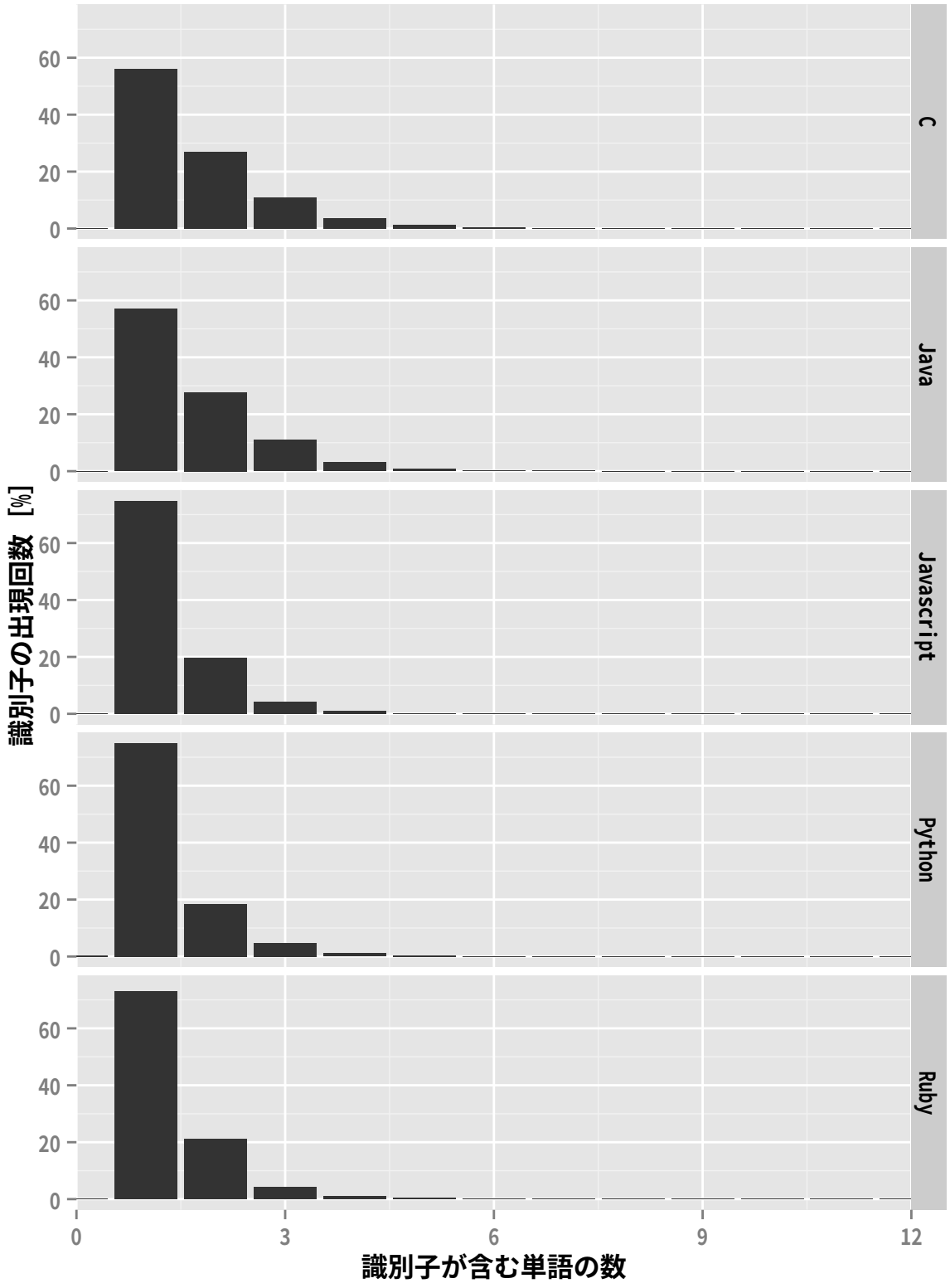

図 4.10　単語数毎の識別子の出現回数の割合

表 4.6　2 つの単語を含む識別子の単語の組み合わせ

C 言語　812,914 種類

1 単語目	68,492 種類		2 単語目	126,518 種類	
順位	数	単語	順位	数	単語
1	21,381	tp	1	9,651	H
2	13,039	p	2	6,771	2
3	6,934	m	3	5,883	1
4	4,632	is	4	2,811	t
5	4,360	set	5	2,741	3
6	4,317	get	6	2,662	Type
7	2,644	new	7	2,593	0
8	2,206	image	8	1,956	4
9	2,180	i	9	1,954	Data
10	2,099	num	10	1,943	Cout

Java 言語　338,092 種類

1 単語目	27,176 種類		2 単語目	51,842 種類	
順位	数	単語	順位	数	単語
1	17,962	tp	1	2,656	2
2	6,947	p	2	2,417	1
3	5,081	get	3	1,985	Type
4	4,049	set	4	1,628	List
5	3,355	m	5	1,585	Name
6	3,138	test	6	1,415	Test
7	2,768	is	7	1,233	Id
8	1,532	new	8	1,200	Map
9	1,486	add	9	1,164	Count
10	1,305	create	10	1,118	Info

Javascript 言語　409,307 種類

1 単語目	28,145 種類		2 単語目	51,587 種類	
順位	数	単語	順位	数	単語
1	9,792	str	1	3,450	2
2	6,421	$	2	3,108	1
3	5,206	this	3	1,622	Name
4	4,070	get	4	1,469	Id
5	3,625	G	5	1,464	3
6	3,538	is	6	1,444	Type
7	3,429	$intern	7	1,398	Data
8	2,898	set	7	1,398	Node
9	2,082	add	9	1,289	List
10	1,878	test	10	1,262	0

Python 言語　383,836 種類

1 単語目	34,511 種類		2 単語目	48,678 種類	
順位	数	単語	順位	数	単語
1	8,854	test	1	3,674	2
2	6,002	get	2	3,023	1
3	3,267	set	3	1,958	name
4	2,906	u	4	1,566	path
5	2,578	is	5	1,535	list
6	2,084	add	6	1,490	id
7	1,959	check	7	1,331	3
8	1,930	new	8	1,263	type
9	1,634	Test	9	1,233	file
10	1,408	do	10	1,232	data

Ruby 言語　204,214 種類

1 単語目	23,015 種類		2 単語目	27,620 種類	
順位	数	単語	順位	数	単語
1	4,449	tp	1	1,869	2
2	3,154	test	2	1,536	1
3	2,304	p	3	1,454	path
4	1,184	get	4	1,426	Test
5	1,140	new	5	1,419	name
6	993	add	6	1,294	id
7	914	create	7	925	type
8	895	set	8	894	Controller
9	841	is	9	883	file
10	805	to	10	874	url

表 4.7　3 つの単語を含む識別子の単語の組み合わせ

C 言語　769,068 種類

1 単語目	37,494 種類		2 単語目	50,846 種類		3 単語目	65,789 種類	
順位	数	単語	順位	数	単語	順位	数	単語
1	22,799	m	1	7,382	2	1	10,352	H
2	13,728	set	2	4,658	3	2	8,327	2
3	12,945	get	3	4,471	1	3	7,001	1
4	7,360	Get	4	4,261	To	4	6,337	t
5	6,156	pyx	5	3,261	get	5	5,007	Type
6	6,112	is	6	3,113	With	6	4,611	Name
7	5,305	s	7	2,791	k	7	4,007	Size
8	4,929	p	8	2,566	File	8	3,468	Data
9	3,618	Set	9	2,539	For	9	3,345	3
10	3,447	k	10	2,511	TYPE	10	3,314	Count

Java 言語　316,487 種類

1 単語目	18,245 種類		2 単語目	19,807 種類		3 単語目	19,686 種類	
順位	数	単語	順位	数	単語	順位	数	単語
1	31,248	get	1	3,533	To	1	4,651	Test
2	14,879	set	2	1,913	File	2	4,100	Name
3	12,587	test	3	1,705	Type	3	3,584	Type
4	11,473	m	4	1,637	List	4	2,713	Id
5	6,045	is	5	1,584	Data	5	2,694	2
6	3,573	create	6	1,430	Key	6	2,605	List
7	3,336	add	7	1,343	Test	7	2,318	1
8	2,105	on	8	1,312	2	8	2,155	Listener
9	1,973	Test	9	1,164	Set	9	2,144	Count
10	1,931	new	10	1,127	Class	10	1,924	Handler

Javascript 言語　222,038 種類

1 単語目	14,112 種類		2 単語目	18,322 種類		3 単語目	20,552 種類	
順位	数	単語	順位	数	単語	順位	数	単語
1	13,143	get	1	3,955	To	1	2,782	Name
2	6,055	set	2	2,262	2	2	2,529	2
3	4,548	on	3	1,465	1	3	2,038	1
4	4,429	is	4	1,015	On	4	1,841	Id
5	3,170	test	5	994	File	5	1,588	tree
6	2,850	create	6	990	From	6	1,397	Data
7	2,818	G	7	988	In	7	1,376	View
8	2,658	add	8	986	Event	8	1,332	Node
9	1,997	update	9	978	3	9	1,320	Element
10	1,809	new	9	978	For	10	1,300	Index

Python 言語　241,792 種類

1 単語目	18,076 種類		2 単語目	22,951 種類		3 単語目	25,604 種類	
順位	数	単語	順位	数	単語	順位	数	単語
1	28,604	test	1	3,056	2	1	2,463	2
2	13,682	get	2	3,009	to	2	2,186	name
3	5,336	u	3	1,555	1	3	1,930	1
4	4,441	set	4	1,453	Test	4	1,704	Error
5	3,106	Test	5	1,201	get	5	1,636	Test
6	2,332	is	6	1,150	3	6	1,519	list
7	1,929	add	7	1,093	from	7	1,392	path
8	1,745	check	8	965	file	8	1,368	id
9	1,695	on	9	845	To	9	1,208	file
10	1,434	create	10	843	for	10	1,004	data

Ruby 言語　116,940 種類

1 単語目	12,451 種類		2 単語目	13,288 種類		3 単語目	12,812 種類	
順位	数	単語	順位	数	単語	順位	数	単語
1	9,918	test	1	2,350	to	1	2,983	Test
2	1,844	get	2	1,483	with	2	2,074	name
3	1,323	set	3	1,228	for	3	2,004	path
4	1,222	create	4	976	2	4	1,365	id
5	1,196	Test	5	859	from	5	1,057	file
6	1,085	add	6	780	file	6	987	for
7	788	new	7	714	by	7	959	2
8	714	find	8	708	or	8	794	url
9	681	default	9	592	in	9	774	1
10	605	assert	10	510	without	10	757	Error

表 4.8　2 つの単語を含む識別子の単語の組み合わせ（出現回数）

C 言語　出現回数 296,248,652

	1 単語目			2 単語目	
順位	出現回数	単語	順位	出現回数	単語
1	6,182,884	p	1	6,597,181	t
2	5,705,894	pyx	2	6,325,819	1
3	4,635,064	Py	3	6,187,780	2
4	4,011,706	m	4	4,268,281	Object
5	3,798,781	G	5	3,334,421	String
6	3,424,322	get	6	3,282,101	u
7	3,312,702	char	7	3,025,554	T
8	2,549,173	Q	8	2,128,371	type
9	2,293,778	r	9	1,840,670	3
10	2,249,552	v	10	1,748,408	name

Java 言語　出現回数 107,364,691

	1 単語目			2 単語目	
順位	出現回数	単語	順位	出現回数	単語
1	10,520,411	get	1	2,831,101	Exception
2	2,988,455	set	2	2,562,753	Name
3	2,628,863	assert	3	2,019,749	String
4	1,804,199	m	4	1,985,512	Type
5	1,599,754	is	5	1,803,338	Equals
6	1,364,711	to	6	1,674,712	List
7	1,031,528	add	7	1,400,718	Id
8	966,069	new	8	1,370,067	1
9	941,511	IO	9	1,369,584	Value
10	898,931	Array	10	1,146,044	2

Javascript 言語　出現回数 285,322,028

	1 単語目			2 単語目	
順位	出現回数	単語	順位	出現回数	単語
1	8,632,445	v	1	7,415,253	Name
2	8,517,332	get	2	5,261,615	Node
3	6,994,731	is	3	4,643,659	2
4	5,554,728	set	4	4,472,737	Query
5	4,882,194	result	5	4,373,103	1
6	4,471,269	j	6	3,689,309	Class
7	4,175,576	add	7	3,660,039	Element
8	3,999,296	$	8	3,511,653	Type
9	3,474,852	node	9	3,131,774	Child
10	3,046,058	a	10	2,940,734	MATCH

Python 言語　出現回数 103,418,183

	1 単語目			2 単語目	
順位	出現回数	単語	順位	出現回数	単語
1	4,158,810	assert	1	2,457,386	Error
2	2,523,720	get	2	2,366,064	2
3	1,823,595	is	3	2,206,698	name
4	1,259,851	add	4	2,198,036	Equal
5	1,258,336	new	5	2,065,826	1
6	1,198,822	set	6	1,977,046	type
7	878,836	test	7	1,607,796	id
8	686,608	G	8	733,054	path
9	550,363	Value	9	694,694	list
10	532,431	make	10	685,747	dir

Ruby 言語　出現回数 47,805,697

	1 単語目			2 単語目	
順位	出現回数	単語	順位	出現回数	単語
1	2,390,704	assert	1	1,814,374	equal
2	2,061,121	to	2	1,407,685	name
3	675,242	should	3	899,824	id
4	626,202	attr	4	864,213	2
5	577,156	be	5	853,748	s
6	544,389	new	6	816,311	1
7	527,608	r	7	774,481	path
8	438,663	o	8	660,820	Error
9	432,317	current	9	627,457	to
10	408,497	is	10	492,279	type

表 4.9　3 つの単語を含む識別子の単語の組み合わせ（出現回数）

C 言語　出現回数 121,418,687

1 単語目			2 単語目			3 単語目		
順位	出現回数	単語	順位	出現回数	単語	順位	出現回数	単語
1	5,181,616	pyx	1	2,417,073	t	1	8,176,839	t
2	3,370,381	m	2	2,120,958	32	2	1,759,019	1
3	3,016,214	Py	3	1,491,942	2	3	1,339,886	2
4	2,433,568	uint	4	1,489,349	v	4	944,808	Object
5	2,361,092	get	5	1,484,106	Array	5	898,773	Type
6	1,805,257	Get	6	1,108,256	3	6	865,504	Name
7	1,369,420	set	7	1,021,209	L	7	779,130	Error
8	1,260,573	ngx	8	843,971	64	8	739,508	3
9	1,235,189	Q	9	827,005	8	9	690,616	ERROR
10	1,218,495	int	10	810,840	LOG	10	620,072	new

Java 言語　出現回数 42,306,188

1 単語目			2 単語目			3 単語目		
順位	出現回数	単語	順位	出現回数	単語	順位	出現回数	単語
1	6,183,253	get	1	641,268	Date	1	1,445,159	Exception
2	2,041,657	m	2	556,203	Of	2	1,023,414	Name
3	1,580,153	set	3	548,562	To	3	704,256	Time
4	780,324	is	4	464,004	Time	4	679,558	Type
5	552,880	add	5	314,403	Type	5	533,394	Id
6	464,998	on	6	310,830	List	6	483,548	Listener
7	405,705	Illegal	7	285,672	Input	7	411,651	Stream
8	361,686	I	8	270,654	Not	8	392,314	Value
9	361,233	create	9	267,228	Field	9	392,237	View
10	360,231	to	10	253,126	Text	10	370,687	List

Javascript 言語　出現回数 63,483,462

1 単語目			2 単語目			3 単語目		
順位	出現回数	単語	順位	出現回数	単語	順位	出現回数	単語
1	4,613,301	get	1	2,534,996	5	1	1,463,723	Case
2	2,578,766	p	2	1,222,657	To	2	1,371,447	2
3	2,328,261	is	3	1,196,038	Event	3	1,180,870	Property
4	1,942,268	to	4	1,165,738	Lower	4	1,030,752	Listener
5	1,728,013	v	5	1,130,095	Own	5	954,214	Array
6	1,632,710	set	6	946,083	m	6	819,000	1
7	1,415,087	has	7	741,991	2	7	723,262	Name
8	1,386,634	add	8	641,081	Char	8	708,415	Node
9	1,357,909	on	9	591,348	Pos	9	701,853	want
10	971,664	create	10	577,245	Text	10	649,494	Code

Python 言語　出現回数 26,379,568

1 単語目			2 単語目			3 単語目		
順位	出現回数	単語	順位	出現回数	単語	順位	出現回数	単語
1	1,734,956	get	1	574,947	to	1	536,625	code
2	746,933	test	2	366,090	2	2	512,166	Error
3	715,603	assert	3	193,677	object	3	502,935	type
4	572,627	set	4	191,480	Implemented	4	437,994	equal
5	487,513	is	5	187,470	frame	5	411,363	name
6	468,393	generate	6	165,230	1	6	259,480	id
7	264,567	add	7	158,876	from	7	238,118	Node
8	248,722	check	8	152,981	Not	8	212,164	2
9	215,031	new	9	147,021	Test	9	191,220	Equal
10	213,646	c	10	142,028	or	10	186,380	value

Ruby 言語　出現回数 9,552,961

1 単語目			2 単語目			3 単語目		
順位	出現回数	単語	順位	出現回数	単語	順位	出現回数	単語
1	403,153	assert	1	293,125	to	1	278,934	name
2	255,441	test	2	152,780	with	2	210,021	of
3	136,158	set	3	131,260	for	3	183,379	for
4	125,644	instance	4	127,205	variable	4	168,645	path
5	120,361	find	5	118,440	by	5	161,708	Error
6	116,603	get	6	112,698	or	6	150,557	id
7	116,270	add	7	105,746	not	7	110,459	file
8	93,733	create	8	98,010	from	8	94,072	n
9	91,558	be	9	92,546	18	9	77,762	method
10	81,060	default	10	83,058	file	10	75,726	key

では，すべての単語数の出現回数の合計で各単語数の出現回数を割り，各単語数の出現回数の割合を示している．

どのプログラミング言語も，単語数 1 の識別子が最もソースコード中に書かれている事がわかる．ここでも，Javascript，Python，Ruby のスクリプト言語と，C や Java 言語の間に傾向の違いがあり，スクリプト言語では単語数の多い識別子の利用が少ない傾向にある．

表 4.6 と表 4.7 に 2 つの単語を含む識別子と 3 つの単語を含む識別子において，識別子の分割を行った時の単語の位置（単語数 3 では先頭の単語（1 単語目），中央の単語（2 単語目），最後の単語（3 単語目））毎に単語の種類を集計した表を示す．

どのプログラミング言語も，先頭の単語では “get” や “set” がよく使用され，最後の単語では “2” や “1” といった数字がよく使用される傾向がある．

表 4.8 と表 4.9 に 2 つの単語を含む識別子と 3 つの単語を含む識別子において，識別子の分割を行った時の単語の位置毎に単語の出現数を集計した表を示す．

出現数においても，先頭の単語では “get” や “set” がよく使用され，最後の単語では “2” や “1” といった数字がよく使用される傾向がある．

4.3 PROMISE データセットの不具合予測

PROMISE で Jureczko らが公開しているデータセットとソースコードを組み合わせ，ソースコード中にある識別子の語長と単語数を用いて不具合予測を行い，他のメトリクスによる不具合予測の結果と比較を行う．

4.3.1 実験方法

実験は次に示す手順で行う．

1. PROMISE のデータセットとソースコードをリンク
2. ソースコードから識別子を抽出
3. 識別子を分割
4. 識別子の語長と単語数を集計
5. ソースコード毎に識別子の語長と単語数を正規化
6. それぞれのメトリクスを用いて不具合予測モデルを作成

7. 不具合予測結果の集計

手順 1 では，バージョンの等しいリポジトリからデータセットに含まれているファイル名と等しいファイルを取り出し，各メトリクスとソースコードを結びつけておく．

手順 2 では，第 3.1 節で解説した識別子の抽出手順を，字句解析器を生成できるソフトウェア lex で実装し，取得したすべてのソースコードに対して実行する．

手順 3 では，第 3.2 節で解説した識別子の分割のうち，アンダーバー記号を用いた単語の分割，数字を用いた単語の分割，キャメルケースを用いた単語の分割の 3 つの手順を実装し，すべての識別子に対して実行する．

手順 4 では，抽出した識別子の文字数や分割した識別子の単語の数を調べ，ソースコードのファイル毎にまとめておき，集計を行う．

手順 5 では，語長や単語数毎の識別子の出現回数からコード行数との相関をなくすため，ソースコード毎にすべての識別子の出現回数の合計で各語長や各単語数の出現回数を割り，それぞれ正規化する．

手順 6 では，機械学習アルゴリズム RandomForest[15] を用いて不具合予測モデルを作成する．

不具合予測モデルを作成には次の 4 つのメトリクスを用いる．

- コード行数

  ソースコードのコード行数をファイル毎に集計した値．
- CK メトリクス [16]

  Chidamber-Kemerer Metrics．オブジェクト指向の設計の複雑さ表す尺度．クラス内部の複雑度を示す WMC（Weighted Methods for Class：クラスの複雑度）と LCOM（Lack of COhesion of Methods：クラスの凝集性の欠如の度合い）．継承の複雑度を示す DIT（Depth of Inheritance：クラス継承の深さ）と NOC（Number of Children：小クラスの数）．結合の複雑度を示す CBO（Coupling Between Objects：クラスが依存する外部クラスの数）と RFC（Response For Class：クラスの応答の数）．以上の 6 つの値を用いる．
- 語長

  識別子の語長毎に識別子の出現回数を集計し正規化した値．

- 単語数識別子の単語数毎に識別子の出現回数を集計し正規化した値.

モデルの学習は，各プロジェクトの最新バージョン以外を使用し，モデルの評価に各プロジェクトの最新バージョンを使用する．表 4.10 に，各プロジェクトの訓練データとテストデータのバージョンとファイル数を示す.

手順 7 では，不具合予測の結果から予測結果の評価指標を計算する．また，作成したモデルには結果にばらつきが出るため，不具合予測モデルの作成とモデルの適用を 100 回繰り返し，得られた結果の平均値を実験結果とする.

モデルによる予測結果は表 4.11 の 4 つの値にクロス集計され，予測結果の評価指標として，正確度 (Accuracy)，適合率 (Precision)，再現率 (Recall)，F 値 (F-measure) を用いる．それぞれの値は以下の計算式で求める.

$$\text{正確度} = \frac{TN + TP}{TN + TP + FN + FP} \tag{4.1}$$

$$\text{適合率} = \frac{TP}{TP + FP} \tag{4.2}$$

$$\text{再現率} = \frac{TP}{TP + FN} \tag{4.3}$$

$$F\text{ 値} = \frac{2 \times \text{適合率} \times \text{再現率}}{\text{適合率} + \text{再現率}} \tag{4.4}$$

正確度は，実測が不具合有りで予測も不具合あり，実測が不具合無しで予測も不具合無しと予測できた割合を示しす．適合率は，予測で不具合有りと判断し，実測でも不具合有りであった割合を示し，正確性の指標となる．再現率は，実測が不具合有りで，予測でも不具合有りと判断できた割合を示し，網羅率の指標となる．適合率と再現率は，一般的にトレードオフの関係にある．そこで，F 値を使用する．F 値は，適合率と再現率の調和平均であり，値が高いほど良い予測ができたといえる.

4.3.2 実験結果

不具合予測の結果をプロジェクト毎に表と図にまとめた.

表 4.12 に，不具合予測の結果を表 4.11 に対応した値で示す.

True negative と True positive の値が正しく予測を行えたファイル数を表す．したがって，False negative と False positive の値が小さいほど予測が良くできているとい

表 4.10　訓練データとテストデータ

プロジェクト	訓練データのバージョン	訓練データのファイル数	テストデータのバージョン	テストデータのファイル数
Ant	1.4 1.5 1.6	819	1.7	741
Jedit	3.2 4.0 4.1 4.2	1,208	4.3	487
Lucene	2.0 2.2	420	2.4	330
Poi	1.5 2.0 2.5	923	3.0	438
Velocity	1.4 1.5	419	1.6	229
Xalan	2.4 2.5	1,438	2.6	875
Xerces	1.2 1.3	891	1.4	328

表 4.11　予測結果の集計

		予測	
		不具合無し	不具合有り
実測	不具合無し	True negative（TN）	False positive（FP）
	不具合有り	False negative（FN）	True positive（TP）

える．プロジェクトによって，予測結果の傾向に違いがり，予測が不具合無しよりのものと不具合有りに偏っていることがある．

表 4.13 と図 4.11 に，不具合予測結果の評価指標の値を示す．

F 値に着目すると，Jedit と Xerces では予測がうまくできていないことがわかる．

図 4.12 に，コード行数と他のメトリクスの差をとった評価値を示す．

コード行数を基準とすることで，プロジェクト毎の差を小さくし，メトリクス間の値の差に注目できる．プロジェクトによりメトリクスの得手不得手があることがわかる．

表 4.12　不具合予測の結果

プロジェクト：Ant

メトリクス	True negative	False negative	False positive	True positive
コード行数	502	72	95	70
CK メトリクス	513	61	79	86
語長	512	62	82	83
単語数	497	77	74	91

プロジェクト：Jedit

メトリクス	True negative	False negative	False positive	True positive
コード行数	413	63	6	5
CK メトリクス	445	30	6	5
語長	448	27	6	4
単語数	447	28	7	4

プロジェクト：Lucene

メトリクス	True negative	False negative	False positive	True positive
コード行数	56	70	104	99
CK メトリクス	49	77	64	138
語長	48	78	51	151
単語数	54	72	57	145

プロジェクト：Poi

メトリクス	True negative	False negative	False positive	True positive
コード行数	86	70	118	162
CK メトリクス	90	66	87	193
語長	91	65	81	199
単語数	80	76	83	197

プロジェクト：Velocity

メトリクス	True negative	False negative	False positive	True positive
コード行数	68	82	20	58
CK メトリクス	69	81	10	67
語長	73	77	8	69
単語数	77	73	7	71

プロジェクト：Xalan

メトリクス	True negative	False negative	False positive	True positive
コード行数	269	194	128	282
CK メトリクス	306	157	117	293
語長	257	206	114	296
単語数	278	185	111	299

プロジェクト：Xerces

メトリクス	True negative	False negative	False positive	True positive
コード行数	115	2	192	17
CK メトリクス	118	0	182	27
語長	116	1	179	30
単語数	117	1	183	26

表 4.13　不具合予測結果の評価値

プロジェクト：Ant

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.773	0.427	0.493	0.458
CK メトリクス	0.809	0.519	0.583	0.549
語長	0.804	0.502	0.571	0.534
単語数	0.795	0.549	0.542	0.546

プロジェクト：Jedit

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.858	0.455	0.074	0.127
CK メトリクス	0.924	0.455	0.140	0.214
語長	0.932	0.454	0.155	0.231
単語数	0.928	0.364	0.124	0.185

プロジェクト：Lucene

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.470	0.488	0.583	0.531
CK メトリクス	0.570	0.683	0.641	0.661
語長	0.606	0.747	0.659	0.700
単語数	0.605	0.715	0.667	0.690

プロジェクト：Poi

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.569	0.578	0.698	0.632
CK メトリクス	0.649	0.688	0.745	0.715
語長	0.666	0.711	0.754	0.732
単語数	0.633	0.701	0.719	0.710

プロジェクト：Velocity

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.550	0.744	0.411	0.530
CK メトリクス	0.597	0.869	0.453	0.595
語長	0.625	0.895	0.473	0.619
単語数	0.650	0.910	0.493	0.639

プロジェクト：Xalan

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.631	0.687	0.592	0.636
CK メトリクス	0.685	0.714	0.650	0.681
語長	0.633	0.721	0.589	0.649
単語数	0.660	0.728	0.617	0.668

プロジェクト：Xerces

プロジェクト	正確度	適合率	再現率	F 値
コード行数	0.408	0.084	0.898	0.154
CK メトリクス	0.442	0.129	1.000	0.228
語長	0.446	0.144	0.938	0.249
単語数	0.439	0.128	0.964	0.226

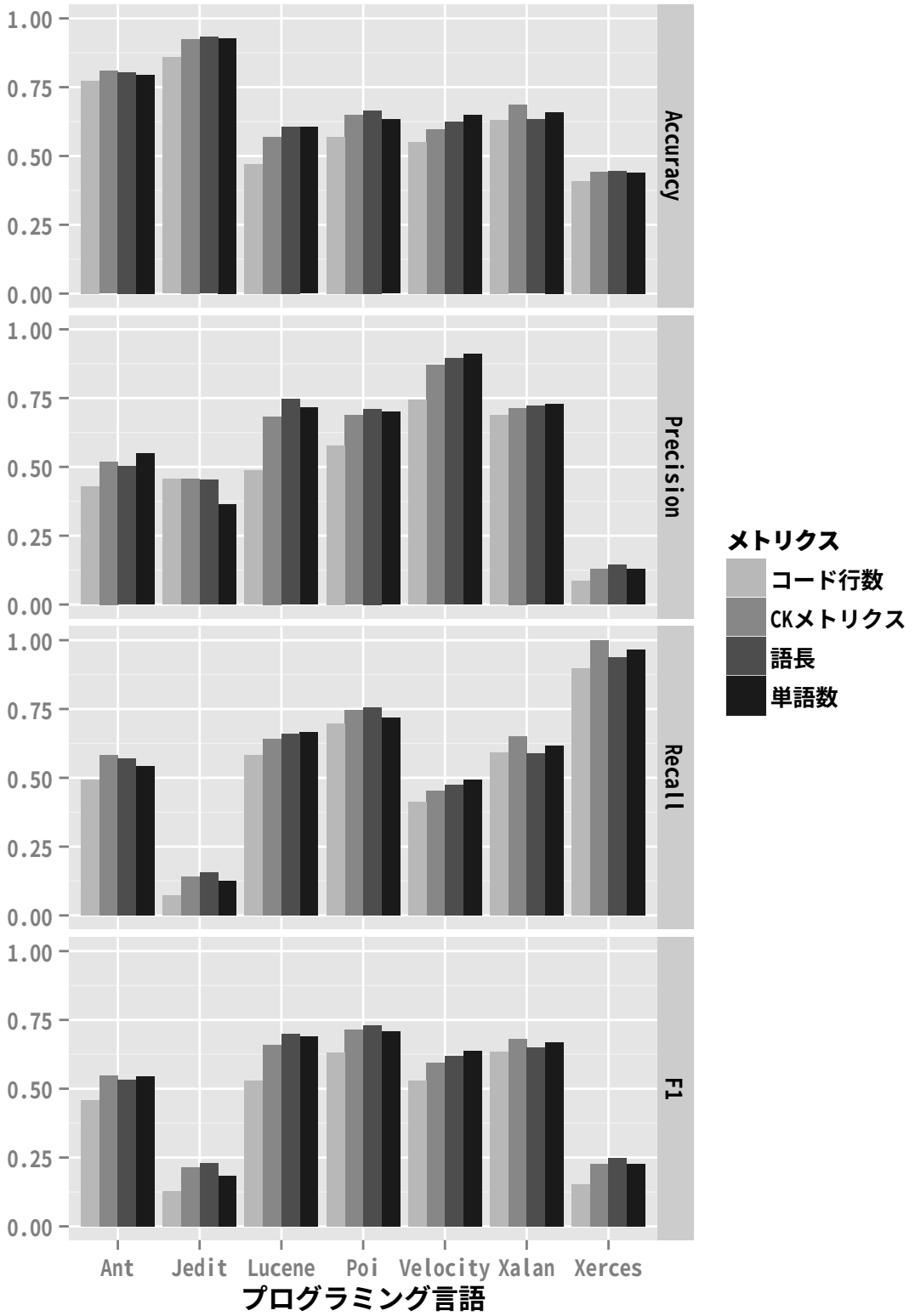

図 4.11　不具合予測結果の評価値

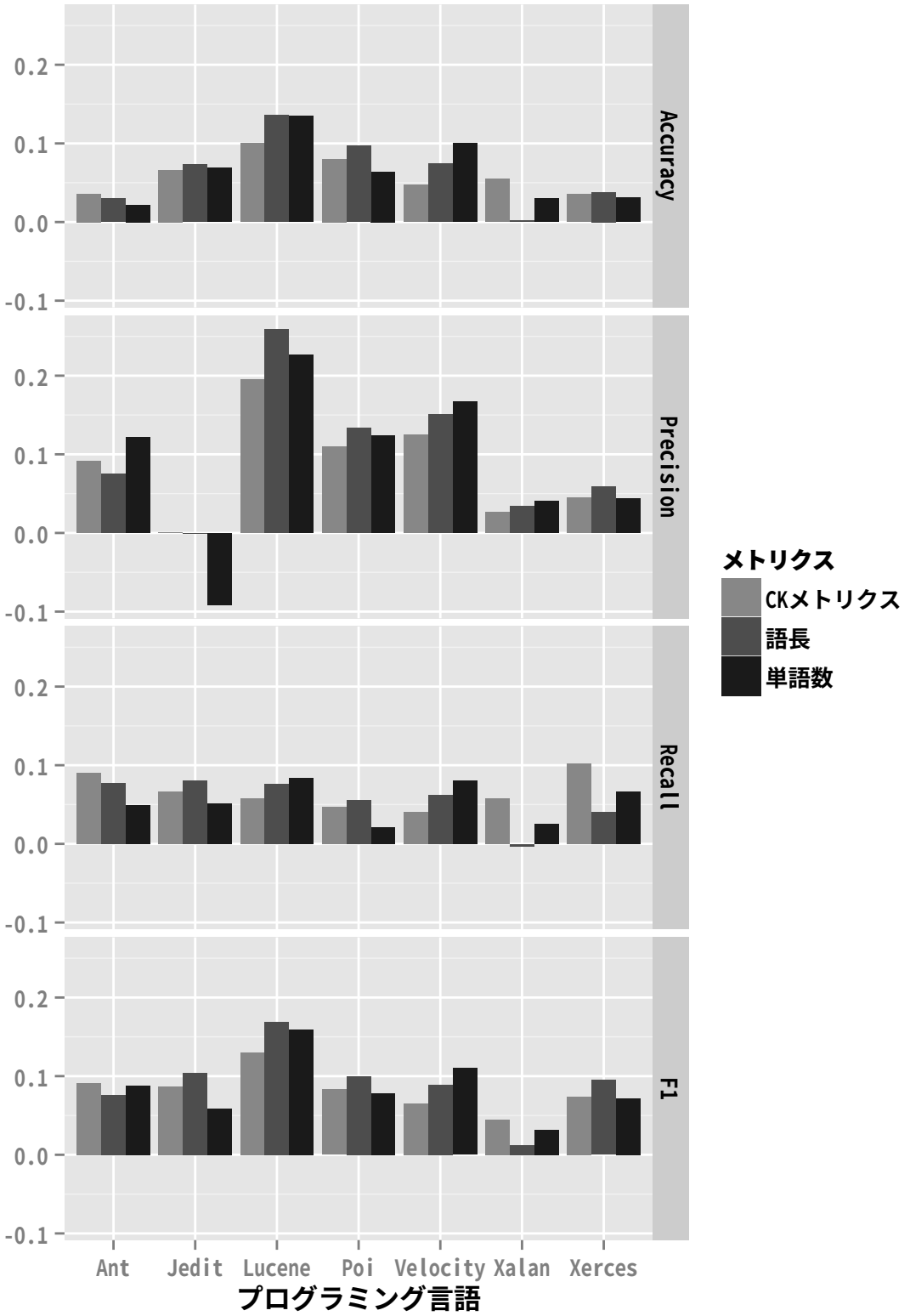

図 4.12　不具合予測結果の評価値（コード行数からの差）

5. 考察

5.1 妥当性の検証

まず初めに，実験結果の妥当性について検証を行う．

プログラムの不備：実験の過程で，識別子の抽出，語長と単語数の計算，ソースコード中の識別子の集計などを作成したプログラムを用いて処理しているが，このプログラムに不備があった場合，識別子の語長や単語数の傾向の分析，不具合予測の実験結果に間違いが存在する恐れがある．

データセットの不備：識別子の語長と単語数の傾向の分析では，オープンソースソフトウェアのソースコードを用いており，そこから抽出された識別子は開発者の不備により意図した識別子と違うものが記述されている恐れがある．不具合予測の実験では，PROMISE データセットにあるソースコードの不具合有無の情報を用いているが，その値が本当の不具合有無と異なっている恐れがある．

ソースコードの自動生成：ソースコードが自動生成ツールなどにより生成された場合，識別子は人間が読む必要がないため，識別子は意味のない文字列の組み合わせとなっている場合がある．この場合，識別子の分割を正しく行えないことや，語長と単語数の傾向に偏りをもたらす恐れがある．

5.2 設問への考察

実験結果の妥当性について注意し，研究設問を通して実験結果に対する考察を示す．

RQ1 ソフトウェア開発者は何文字の識別子をよく用いるか

ソフトウェア開発者が識別子の命名を行うときにどの語長の識別子を用いるか知りたいため，語長毎の識別子の種類をプログラミング言語毎にまとめた図 4.3 と図 4.4 を参考にする．

最も識別子の種類の多い語長は，Java が語長 13，C が語長 12，Javascript と Python と Ruby が語長 11 となっている．スクリプト言語では語長の短い識別子が多い傾向が見られ，識別子の命名時に語長の短い単純な識別子が用いられ易いことが考えら

れる．逆に，C や java では，語長 25 を超えても識別子の種類は多く，識別子の語長が長くなっても丁寧に命名を行う傾向があると考えられる．

短い語長の識別子の種類が極端に多い場所が見られるが（Java では語長 6 と 7，Javascript では語長 3，Python では，語長 5），これは，ソースコードのファイルサイズを小さくする目的や，機械的に意味のない文字列を組み合わせて生成された変数が影響していると考えられる．

特に，Javascript では，しっかりと単語を組み合わせた識別子を用いて記述されたソースコードを，Web の読み込み時間を短くするためにファイルの軽量化を行い，識別子を語長の短いものに置き換えている場合がある．そのため，語長の短い順に全ての識別子に使用可能な文字を組み合わせて識別子を命名しソースコードに記述している場合がある．

RQ2 ソフトウェア開発者は識別子に幾つ単語を含ませるか

ソフトウェア開発者が識別子の命名を行うときにどの単語数の識別子を用いるか知りたいため，単語数毎の識別子の種類をプログラミング言語毎にまとめた図 4.7 と図 4.8 を参考にする．

どのプログラミング言語も，単語数 2 の識別子の種類が最も多く，2 つの単語を組み合わせて識別子が，手軽に必要な意味を込めた命名が行えていると言える．単語数 3 の識別子の種類が単語数 2 の次に多いが，スクリプト言語である Javascript と Python と Ruby では，C と Java に比べ割合が少なく，単語数 4 より単語数 1 の識別子が多いなど，少ない単語数の識別子を使用したコーディングが多いことがわかる．これは，識別子に意味を持たせることが難しいため，ソースコードの理解が難しくなると考えられる．

C と Java では単語数が多くなっても識別子の種類が多く，丁寧に意味を多く持たせた識別子を使用し，理解しやすいソースコードを書いていると考えられる．

RQ3 ソースコード中で最も出現回数の多い識別子の語長と単語数は何か

まず，語長では，図 4.5 と図 4.6 から，どのプログラミング言語も，語長 1 から 10 の識別子の出現回数が多く，短い識別子が多くソースコードに記述される事がわかる．これは，繰り返し使われる識別子の語長を短くすることと，ソースコードの 1 行の長さを短くしたり，ファイルサイズを軽減することができる為と考えられる．

単語数では，図 4.9 と図 4.10 から，どのプログラミング言語も単語数 1 の識別子が最も出現している事がわかる．これは，短い語長をよく使用するのと同じ理由があると考えられる．

RQ4 よく識別子に使われる単語は何か

まず，識別子の種類で集計した表 4.6 と表 4.7 から，どのプログラミング言語でも，1 単語目に使用する単語として “get” と “set” があり，識別子の命名において基本となる単語といえる．同じように “new” や “add”，“is” なども，識別子において一般的なものと考えられる．

英単語では無い文字列として，“tp” や “t”，“m”，“u” などが見られるが，これらはハンガリー記法を用いた例と考えられる．

2 単語目に用いられることが多い単語として，数字の 2 や 1 がある．これらは，同じ意味を持った識別子を単純に区別するために識別子に追加した結果と考えられ，ソースコードの品質を考えると悪影響があると考えられが，命名における負担を軽減するために高い頻度で行われてしまっていると考えられる．数字の中では，1 という数字より 2 という数字がよく使われており，同じ意味の識別子を追加するときに，元の識別子の末尾に数字の 2 を追加するという事が行われており，元の識別子に数字の 1 を付けない場合がある為と考えられる．

次に，識別子の出現回数で集計した表 4.8 と表 4.9 をから，C や Javascript，Python，Ruby で英単語ではない文字や数字が多く使われている傾向がわかり，Java では 2 語目以降の単語は先頭の文字が大文字など，キャメルケースによる命名がしっかりと行われていることがわかる．Java の開発者は識別子をしっかりと命名していることが考えられる．

Javascript の 2 つの単語を持つ組み合わせでは，1 単語目に “j” があり，2 単語目に “Query” という単語があり，これは，Javascript で jQuery というライブラリが頻繁に使われていることがわかる．また，Javascript の単語で現れる “$” は，jQuery で使われる記号であり，GitHub に公開されている Javascript のプロジェクトは jQuery を使っているものが多いと考えられる．

識別子の種類と単語数は，大規模なプロジェクトがあるとその影響を大きく受けることが考えられ，単語の組み合わせに大きく影響を与えている可能性がある．

RQ5 識別子の語長を用いてソースコードの品質を予測できるか

表 4.12，表 4.13，図 4.11 の語長の項目より，プロジェクトによっては上手く予測できていない場合もあるが，それなりに不具合を予測できていると言える．

コード行数を使った不具合予測と比較した図 4.12 の語長の部分から，どのプロジェクトにおいても殆どの値がコード行数の評価値より語長の評価値が高く，比較的良く不具合を予測できていると考えられる．

RQ6 識別子を構成する単語の数を用いてソースコードの品質を予測できるか

表 4.12，表 4.13，図 4.11 の単語数の項目より，プロジェクトによっては上手く予測できていない場合もあるが，それなりに不具合を予測できていると言える．

コード行数を使った不具合予測と比較した図 4.12 の語長の部分から，どのプロジェクトにおいても殆どの値がコード行数の評価値より単語数の評価値が高く，比較的良く不具合を予測できていると考えられる．

語長と単語数の評価値を比較すると，どちらにも良く予測できたプロジェクトとあまり予測できないプロジェクトがあり，どちらか一方が良いとは言えない結果となっている．

RQ7 識別子の語長や単語数を用いた予測と他の不具合予測ではどちらが良いか

コード行数を使った不具合予測と比較した図 4.12 から，CK メトリクスによる不具合予測はコード行数による予測より良い結果を出しており，CK メトリクスは不具合予測を良くできていると言える．

CK メトリクスの結果と語長や単語数の結果を比較すると，どちらにも良く予測できたプロジェクトとあまり予測できないプロジェクトがあり，ここでも，どちらか一方が良いとは言えない結果となっている．

しかし，CK メトリクスはソースコードの構文を解析しクラスの構造や関係などを調べ計算する必要があるのに対して，語長や単語数はソースコードの字句解析を行うだけで計算可能なため，語長や単語数を用いることで比較的手軽な不具合予測になるのではと考えられる．

6. 関連研究

ソースコード中の識別子がソースコードの理解に与える影響についての研究が行われている．Caprile らは，関数名の字句，構文，意味の構造を解析し，関数名の分類を行い，プログラムの理解や記述支援に関して研究している [17]．更に，ソースコード中の識別子に対して，識別子の標準化（字句と構文の自然言語に直す）を行い，ソースコードの理解を容易にするための研究も行なっている [18]．Rilling らは，ソースコードの分かりやすさを評価するためにソースコード中の識別子を利用したメトリクスを定義している [19]．Lawrie らは，識別子を頭文字，省略語，完全な単語で表記した時のプログラマの理解度に対する実験を行い，プログラムの習熟度や性別による識別子の理解度の違いを示している [3]．更に，識別子の命名の特徴について開発年代毎に数値化し分析を行い，識別子の命名規則について検証している [20][21]．Binkley らは，人間の脳の短期記憶と識別子の語長に関する研究．プログラマにさまざまな語長の識別子を読ませ理解度に対する分析している [22]．Eshkevari らは，Eclipse-JDT と Tomcat を実験対象として，識別子のリネームがソースコードに与える影響を分析している [23]．

識別子の命名規則を定義し，ソースコードの品質を改善するという研究が行われている．Lawrie らと Deissenboeck らと Liblit らは，現存のコーディングスタイルガイドに対して評価を行い，識別子の簡潔さと一貫性のある構造について定義を行い，簡潔さと一貫性のある識別子の命名について研究している [24][1][25]．Butler は，Java のクラス名の構造を調査し，ソースコードの理解を高めるクラス名の命名について提案している [26]．

ソースコード中の識別子に注目し，識別子とソースコードの品質や不具合についての研究が行われている．Butler らは，識別子の命名規則とソースコードの品質について実験している．ソースコードの品質は FindBugs などの静的解析ツールを用いる測定し，識別子の命名規則との関連を分析している [2]．山本らは，ソースコード中の特定の識別子とソフトウェアバグの間に関係があるか分析し，変数名のバグ密度を定義し実験している [27]．Chen らは，ソースコードに対してトピックモデルを構築し，トピックモデルを用いた不具合予測の実験をしている [28]．

第 3.2 節の識別子の分割で示した，小文字だけや大文字だけで構成された文字列

の識別子に対する分割手法の研究が行われている．更に，識別子に用いられる短縮語を自然な単語に戻す手法の研究も行われている．Feild らは，辞書を用いた greedy アルゴリズム，ニューラルネットを用いたアルゴリズムによる識別子分割手法を提案している [7]．Lawrie らは，ソースコードにおいて，識別子の省略表記がプログラムの理解を難しくしているとし，ソースコードに一緒に現れる単語を集め省略語を展開し元の単語に戻す手法を提案している [29]．更に，分割制度の評価指標として Google の単語間の出現確率を用いている識別子分割アルゴリム GenTest と識別子の標準化アルゴリズム Normalize を提案している [9][10]．Enslen らは，ソースコードに含まれる識別子を学習して識別子を分割する，識別子分割アルゴリム Samurai を提案している [8]．Guerrouj らは，speech recognition を応用した識別子分割・標準化アルゴリズム TIDIER を提案している [11]．更に，木構造を利用した識別子の分割や標準化を行う識別子分割・標準化アルゴリズム TRIS を提案している [12]．

7. 結言

本研究では，ソースコード内で識別子が重要と捉え，識別子命名の要素のうち語長と単語数に着目し分析を行った．その過程で，ソースコード中の識別子の抽出，単語を組み合わせて作られた識別子の分割，識別子の語長と単語数を使った不具合予測の手法を提案した．

多数のソースコードから大量の識別子を抽出し，語長や単語数，識別子に用いる単語を分析することで，ソフトウェア開発者の識別子の命名の傾向についで分析することができた．

識別子の語長と単語数によるソフトウェアの不具合予測モデルを構築し，実際に語長と単語数が不具合予測に使用出来ることを示した．識別子の語長と単語数を使った不具合予測は，語長や単語数がソースコードから比較的手軽に計算できるため有効な手法であると示せた．

今後の課題として，識別子を関数名やクラス名，変数名など役割に分類した上での分析を行うことで，より深く識別子について分析できるのではと考えられる．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修准教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻梁軍偉君，出原真人君，中井道君，椋代凛君，大西哲朗君，学生生活を通じて筆者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” Software Quality Control, vol.14, pp.261–282, Sept. 2006.

[2] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, “Relating identifier naming flaws and code quality: an empirical study,” Proc. of the Working Conf. on Reverse Engineering, pp.31–35, IEEE Computer Society, Sept. 2009.

[3] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “What’s in a name? a study of identifiers,” Proceedings of the 14th IEEE International Conference on Program Comprehension, pp.3–12, IEEE Computer Society, Washington, DC, USA, 2006.

[4] 川本公章，水野　修，“ソフトウェアモジュールにおける識別子の語長と不具合出現に関する分析,” 情報処理学会研究報告 ソフトウェア工学 (SE)，vol.2011-SE-173，pp.1–8，July 2011.

[5] K. Kawamoto and O. Mizuno, “Do long identifiers induce faults in software? — a repository mining based investigation —,” Proc. of 22nd International Symposium on Software Reliability Engineering (ISSRE2011), Supplemental proceedings, pp.1–6, Nov. 2011. Hiroshima, Japan.

[6] K. Kawamoto and O. Mizuno, “Predicting fault-prone modules using the length of identifiers,” Proc. of 4th International Workshop on Empirical Software Engineering in Practice (IWESEP 2012), pp.30–34, Oct. 2012.

[7] H. Feild, D. Binkley, and D. Lawrie, “An empirical comparison of techniques for extracting concept abbreviations from identifiers,” Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006), pp.81–85, Nov. 2006.

[8] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, “Mining source code to automatically split identifiers for software analysis,” Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, pp.71–80, IEEE Computer Society, Washington, DC, USA, 2009.

[9] D.J. Lawrie, D. Binkley, and C. Morrell, “Normalizing source code vocabulary,” Proceedings of the 2010 17th Working Conference on Reverse Engineering, pp.3–12, 2010.

[10] D. Lawrie and D. Binkley, “Expanding identifiers to normalize source code vocabulary,” Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, pp.113–122, IEEE Computer Society, Washington, DC, USA, 2011.

[11] L. Guerrouj, M.D. Penta, G. Antoniol, and Y.-G. Guéhéneuc, “TIDIER: An identifier splitting approach using speech recognition techniques,” Journal of Software Maintenance - Research and Practice, p.31, 2011.

[12] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M.D. Penta, “Tris: a fast and accurate identifiers splitting and expansion algorithm,” Proceedings of the 19th Working Conference on Reverse Engineering, pp.103–112, Oct. 2012.

[13] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, The PROMISE Repository of empirical software engineering data, （オンライン）, 入手先 ⟨http://promisedata.googlecode.com⟩（参照 2013-2-1）.

[14] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp.9:1–9:10, ACM, New York, NY, USA, 2010.

[15] L. Breiman, “Random forests,” Machine Learning, vol.45, pp.5–32, Oct. 2001.

[16] S.R. Chidamber and C.F. Kemerer, “A metrics suite for object oriented design,” Software Engineering, IEEE Transactions on, vol.20, no.6, pp.476–493, June 1994.

[17] B. Caprile and P. Tonella, “Nomen est omen: Analyzing the language of function identifiers,” Proceedings of the Sixth Working Conference on Reverse Engineering, pp.112–122, IEEE Computer Society, Washington, DC, USA, 1999.

[18] B. Caprile and P. Tonella, “Restructuring program identifier names,” Proceedings of the International Conference on Software Maintenance (ICSM’00), pp.97–107, IEEE Computer Society, Washington, DC, USA, 2000.

[19] J. Rilling and T. Klemola, “Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics,” Proceedings of the 11th IEEE International Workshop on Program Comprehension, pp.115–124, IEEE Computer Society, Washington, DC, USA, 2003.

[20] D. Lawrie, H. Feild, and D. Binkley, “Quantifying identifier quality: an analysis of trends,” Empirical Software Engineering, vol.12, pp.359–388, Aug. 2007.

[21] D. Lawrie, H. Feild, and D. Binkley, “An empirical study of rules for well-formed identifiers: Research articles,” Jounral of Software Maintenance and Evolution: Research and Practice, vol.19, no.4, pp.205–229, July 2007.

[22] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, “Identifier length and limited programmer memory,” Science of Computer Programming, vol.74, no.7, pp.430–445, May 2009.

[23] L.M. Eshkevari, V. Arnaoudova, M.D. Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of identifier renamings,” Proceedings of the 8th Working Conference on Mining Software Repositories, pp.33–42, ACM, New York, NY, USA, 2011. Waikiki, Honolulu, HI, USA.

[24] D. Lawrie, H. Feild, and D. Binkley, “Syntactic identifier conciseness and consistency,” Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp.139–148, IEEE Computer Society, Washington, DC, USA, 2006.

[25] B. Liblit, A. Begel, and E. Sweetser, “Cognitive perspectives on the role of naming in computer programs,” In Annual Psychology of Programming Workshop, pp.53–67, Sept. 2006.

[26] S. Butler, “Mining java class identifier naming conventions,” Software Engineering (ICSE), 2012 34th International Conference on, pp.1641–1643, June 2012.

[27] 山本博之，亀井靖高，松本真佑，門田暁人，松本健一，“ソフトウェアバグと変数名の関係の分析 (ソフトウェア解析),” 電子情報通信学会技術研究報告. KBSE, 知能ソフトウェア工学，vol.109，no.307，pp.67–71，Nov. 2009.

[28] T.-H. Chen, S.W. Thomas, M. Nagappan, and A.E. Hassan, "Explaining software defects using topic models," Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp.189–198, 2012.

[29] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp.213–222, IEEE Computer Society, Washington, DC, USA, 2007.