UNIVERSITÄT
BERN

Automatic Token Classification

An attempt to mine useful information for parsing

Bachelor Thesis

Joël Guggisberg

from

Busswil BE, Switzerland

13. December 2015

Prof. Dr. Oscar Nierstrasz

Phd. Jan Kurš

Software Composition Group

Institut für Informatik und angewandte Mathematik

University of Bern, Switzerland

Abstract

Developers need software models to make decisions while developing software systems. Static software models are commonly imported by parsing source code. But building a custom parser is a difficult task in most programming languages. To build such a custom parser the grammar of the programming language is needed. If the grammar is not available, which is the case for many languages and notably dialects, it has to be inferred from the source code. Automatically finding the keywords of those languages can help the process of inferring a grammar because many keywords identify beginnings and endings of the basic building blocks of programs.

We tested four heuristics of finding keywords in source code of unknown languages: i) the most common words are keywords; ii) words that occur in most of the files are keywords; iii) words at the first position of the line before an indent are keywords; and iv) words at the beginning of a line are keywords.

With our third method we achieved the best results. It found 26 of the 50 Java keywords, 10 out of 17 of the Shell keywords and 9 out of 20 of the Haskell keywords. Our data suggests that the more source code is available the more precise the results get. Adding more source code produced the most improvements when using the first or second method.

Contents

1 Introduction

1.1 Motivation

It is widely accepted that software developers spend more time reading code than writing it [3]. Reading code not only promotes program comprehension but helps the developer understand the impact of their changes on the existing system. Nevertheless there are numerous questions developers ask that cannot simply be answered by reading code, such as *"which code implements this feature?"*, or *"what is the impact of this change?"*, and for which dedicated analyses are needed [6, 7].

That is where software models come into place. Dedicated platforms exist to analyse software systems, such as Moose [4] and Rascal [2]. These platforms help the developer make decisions by analysing and presenting problem dependent aspects of the underlying software system. Software models are commonly imported by parsing source code. But building a custom parser is a difficult task in most programming languages [5].

To build such a custom parser the grammar of the programming language is needed. If the grammar is not available —which is the case for many languages and especially dialects— it has to be inferred from the source code [1]. Inferring a grammar and building a parser can take several person-months of effort. Inferring might be easier if we first find the keywords in the raw source code [5].

In our bachelor thesis we argue that it is possible to automatically infer keywords

from common programming languages. We explore and test four methods to identify keywords:

Global method expects keywords to appear the most frequently over all source code;

Coverage method expects keywords to appear in most files;

Newline method expects keywords to appear the most frequently at the beginning of a newline;

Indent method expects keywords to appear at the beginning of a newline before an indent.

To make the process of finding keywords applicable to unknown languages our program assumes as little as possible about the language's syntax. The result of this thesis is a tool that is able to automatically start tokenizing and classifying tokens in unknown code. We show that with our tool and the first method we are able to extract 26 of the 50 Java keywords, 10 out of 17 of the Shell keywords and 9 out of 20 of the Haskell keywords.

1.2 Goal and Focus

The goal of our thesis is to develop a tool which can automatically find the keywords of unknown source code. The process of automatically finding keywords can be split up into two subproblems:

1. **Tokenizing**. A file loaded into memory is just a sequence of characters. To find the keywords we have to split the sequence into tokens: which are basic units of meaning. It seems obvious to our perception that `package autoca.*; ..` should be split into *"package"* and *"autoca"*. But for our tool we need to find a set of rules so it can identify the beginning and end of tokens across different languages.
2. **Analysing**. From the tokenizing stage we get a list of all occurring tokens. The analysis stage now tries to identify which of the tokens are keywords. Since we do not know how to find the keywords methods have to be found and tested. Additionally the tool should be able to evaluate these methods.

With the introduced methods to find keywords also assumptions about commonalities between programming languages had to be made. We tried to keep the assumptions as broad as possible in such a way that the focus stayed on the task of finding keywords but the final program still could easily be extended to further minimize the assumptions:

- For a language to be similar to another language means that they have common constructs and since they have common constructs developers structure their code similarly.
- It is given that most programming languages that are close descendants of the same language are similar to each other.
- There is typically enough source code available for the methods to be tested on.
- Formatting of the code contains indents and dedents as it is good practice amongst developers.
- It is known what the comments look like in the code.

To visualize what is described in the texts we are using the Java code snippet seen in Listing 1 throughout the thesis. In Listing 1 Java keywords are indicated in blue.

```
1  package autoca.mode;
2  import Tokenizer;
3
4  public final class AnalyseMode
5     implements IOperationMode  {
6
7     private static final Logger logger =
8        Logger.getLogger(AnalyseMode.class);
9
10    public AnalyseMode(JSONInterface data){
11       try{
12          this.db = new Database(data);
13       } catch (SQLException e) {
14          logger.error("Analyse Mode",e);
15       }
16    }
17 }
```

Listing 1: Java code snippet with the keywords marked in blue

2
Methods

In this chapter we describe four methods proposed by us. In our context a method defines a process to find the keywords inside code. The result of a method is a table with all found tokens and their respective ranking.

Each description of a method starts with a short explanation. To visualize what the method does the explanation is followed by our Java code snippet. In the snippet tokens affected by the method are highlighted in orange. Delimiters have been left out for readability and only the tokens are left in the example. They are separated by whitespaces. Strings are considered to be single tokens. After the code snippet we appended the first three rows of the result table of each method.

There are two techniques to explore:

1. Exploiting indentation to identify structural elements
2. Analyzing recurring names to distinguish the potential of tokens to be keywords.

Based on these two techniques we introduce four methods:

2.1 Global Method

The global method assumes that tokens that occur most frequently on a global scale in the source code are keywords.

Explanation: Consider a large amount of source code of a specific unknown language. It is likely the tokens that occur most would be the keywords since in different projects different names would be used for classes and variables. Different names means fewer occurrences per name overall which then results in loss of influence in the result. Our prediction is that the more data is added the better the result of this method.

Listing 2 shows that Global method affects all tokens found in a file.

```
1  package autoca mode
2  import Tokenizer
3
4  public final class AnalyseMode
5     implements IOperationMode
6
7     private static final Logger logger
8        Logger getLogger AnalyseMode class
9
10    public AnalyseMode JSONInterface data
11       try
12          this db new Database data
13       catch SQLException e
14          logger error "Analyse Mode" e
```

Listing 2: Java example Global

Table 2.1 shows the first three results of the result table of the Global method.

Token	Count
AnalyseMode	3
public	2
data	2
	⋮

Table 2.1: Global result table

2.2 Coverage Method

The coverage method counts in how many files a token appears: the more often it appears the more likely it is expected to be a keyword.

Explanation: In the Global method it is possible that a single file can change the whole result since in this file could be a complex algorithm or data manipulation which reuses a token a significant amount of times, because of this, Coverage only counts each occurrence of a token once per file. The risk of discounting highly specialized keywords is a downside of the coverage method.

```
package autoca mode
import Tokenizer

public final class AnalyseMode
   implements IOperationMode

   private static final Logger logger
      Logger getLogger AnalyseMode class

   public AnalyseMode JSONInterface data
      try
         this db new Database data
      catch SQLException e
         logger error "Analyse Mode" e
```

Listing 3: Java example Coverage

The result is similar to the case of the Global example except that the tokens that appear multiple times in the same file are only added once to to the result table per file.

Token	Count
IOperationMode	1
mode	1
public	1
	⋮

Table 2.2: Coverage result table

2.3 Newline Method

The Newline method counts how many times a token appears at the beginning of a newline: the more frequently it appears the more likely it is expected to be a keyword.

Explanation: Just by glancing at a piece of code with highlighted keywords it seems that most of the keywords occur at the first position of a new line. As one might expect the downside is also that some keywords seem unreachable like *static* or *implements* and *extends*.

```
1 package autoca mode
2 import Tokenizer
3
4 public final class AnalyseMode
5    implements IOperationMode
6
7    private static final Logger logger
8       Logger getLogger AnalyseMode class
9
10   public AnalyseMode JSONInterface data
11      try
12         this db new Database data
13      catch SQLException e
14         logger error "Analyse Mode" e
```

Listing 4: Java example Newline

Token	Count
public	2
private	1
Logger	1
	⋮

Table 2.3: Newline result table

2.4 Indent Method

The Indent method counts how many times a token appears at the beginning of a newline before an indent: the more frequently it appears the more likely it is expected to be a keyword.

Explanation: A more precise way than the Newline method is to rely on indentation as a tool of a visual syntactical identifier for programmers. Assume that most programmers keep their code clean and visually appealing. Also there are programming languages that use Indentation for block structures. The assumption that programmers keep their code clean and the knowledge that the structure after an Indent is defined by the first token on the line leading the indentation concludes the indentation method. It has similar disadvantages as the Newline method. An additional disadvantage is that code could not be formatted at all or indentation has no syntactic value in a programming language.

```
1 package autoca mode
2 import Tokenizer
3
```

```
4 public final class AnalyseMode
5    implements IOperationMode
6
7    private static final Logger logger
8       Logger getLogger AnalyseMode class
9
10   public AnalyseMode JSONInterface data
11      try
12         this db new Database data
13      catch SQLException e
14         logger error "Analyse Mode" e
```

Listing 5: Java example Indent

Tokens in this example are separated by whitespaces. Delimiters are excluded by the parser except for the string token to show that it is counted as a whole token and the words inside the string do not matter. The Indent method now sums up token indicated in orange with the same value . The ones with the highest sum value are considered to be keywords.

Token	Count
public	2
private	1
try	1
	⋮

Table 2.4: Indent result table

3 Implementation

This chapter gives a rough overview of the development process. We discuss how we improved the prototype named Lexica to the final version called AuToCa.

3.1 Technology

As programming language to develop AuToCa we used Java, as build automation Gradle and as database the MySql Database H2.

3.2 Prototype: Lexica

Lexica was a first prototype to verify our ideas. To start off we limited the method to Global and Coverage. We implemented its parser with fixed lookahead. The parser decided on the fly what to do with the current token found. So when a token is found it is inserted directly into the database. The result tables are updated accordingly. Special cases, comments and syntactical structures were hard coded in the algorithm. Adapting to a new language meant changing the whole parsing algorithm.

There are two classes. The Scanmode class handles scanning and analysing. The

Database class is responsible for controlling the access to the result tables in the H2 database. Figure 3.1 shows the UML diagram of Lexica. An occurrence of a token in the Scanmode calls *insertToken()* which updates the result table of each method respectively.

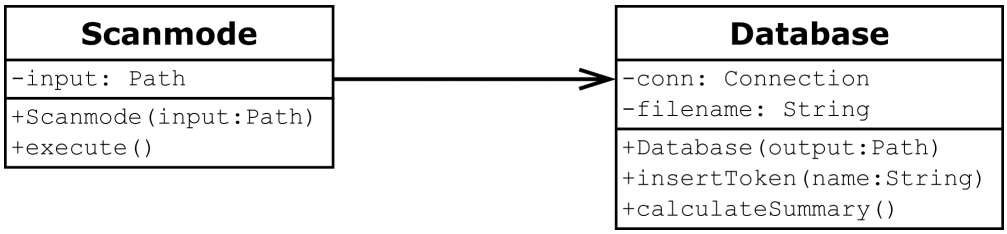

Figure 3.1: Class diagram of prototype: Lexica

Problems

After the first test runs on code the disadvantages of the prototype became clear.

- **Adjustability:** Several steps were fit into one algorithm, scanning and analysing. This was bad because to adjust an analysis the same code had to be rescanned.
- **Runtime:** With additional input the runtime increases extensively. The database access and indexing was on a per occurrence base, which is a problem because opening the connection and rebuilding the indices takes time.
- **Extendability:** The code of the algorithm in the Scanmode class reached such a complexity that extending was impossible without rewriting it.

3.3 New Approach: AuToCa

Knowing all the disadvantages of the prototype helped us get to know the problem set and so define the specification for AuToCa. Our design approach improved everything that had gone wrong in the first one. Figure 3.2 shows the class diagram of AuToCa and how we split the whole process up into a TokenizeMode and an AnalyzeMode.

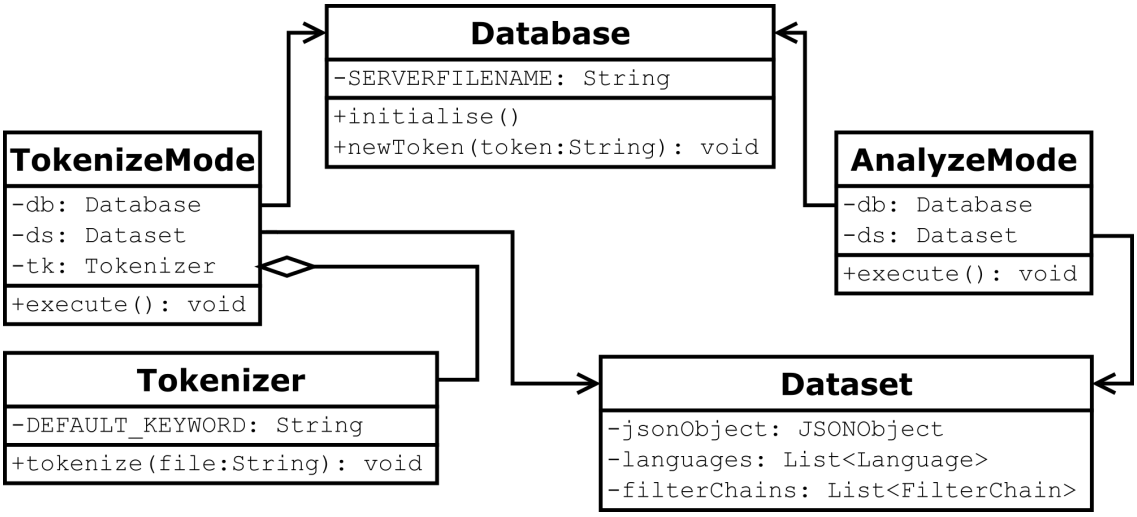

Figure 3.2: Simplified class diagram of AuToCa

TokenizeMode From a prepared configuration file the tokenizer loads the settings on how a delimiter, string and other tokens are to be recognized as. These settings come in the form of a RegEx. Every setting is adjustable. For example it is possible to include new characters inside tokens or change the syntax of comments. Then the tokenizer loads a code file and tokenizes its char sequence. Upon finding a token the tokenizer inserts it into the database. Tokens like delimiters, strings or Newlines are tagged as *#delimiter*, *#string* and *#Newline*. When the tokenizer is finished the extracted tokens are stored inside the *Occurrence* table without loss of important data. Theoretically it would be possible to reconstruct all tokenized files.

AnalyzeMode The analyser can now apply the methods on the *Occurrence* table inside the database. Methods are applied by using MySql queries which is convenient as MySql queries can easily be adjusted and improved. In case there is a method exceeding the functionality of a MySql query we can still export the data back into Java and handle the operation there.

Database Accesses to the database are now controlled in such a way that they are only done when necessary. After each separate part of the execution the connection is closed: which empties the connection cache and improves overall performance. At the heart of our design is the table schema of the database (see Figure 3.3). Both the tokenizer and analyser access it. There are three types of attributes in the database. *Ids* of the type `AUTO_INCREMENT` or `MEDIUMINT` holding the integer representation of values. *Files* of the type `VARCHAR` holding the filenames and *token* of the type `VARCHAR` holding the token values.

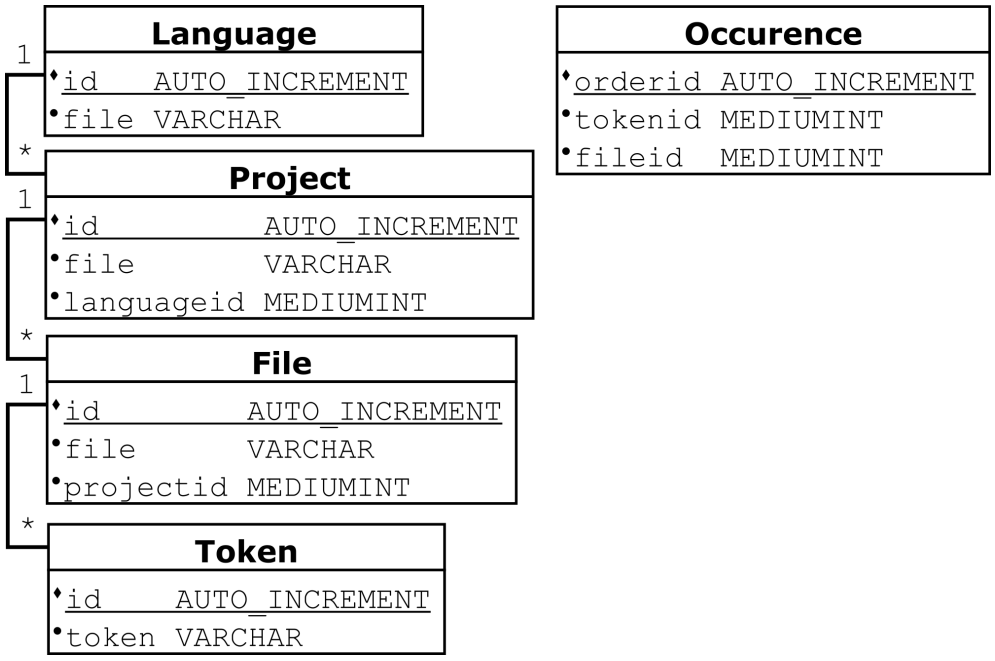

Figure 3.3: Database scheme

The requirements for the schema were minimal-redundancy, structural completeness of data and ease of maintainability. Our database schema allows us to store an occurrence of a token with no redundancy. Also the data most needed is kept inside the same table. Strings such as file names and token values are mapped to integer ids. This improves performance and reduces disk space or memory needed. The most important table is the occurrence table. Each entry of the table describes the exact position of a token with an id. The id reveals at which position in a file, project and dataset a token occurred.

3.3.1 Database Example

To see what the structure of the database looks like we can now insert our sample code from before (Listing 1) into AuToCa.

As a result we get an H2 database (see Figure 3.4). The table that brings everything together is the *Occurrence* table which stores all the tokens in the order they appeared during tokenizing. ORDERID is the primary key of the table. Each occurrence of a token has a unique integer value. With the TOKENID we can look up the token value in the *Token* table. With FILEID the file value in the *File* table. If we need to know in which project a token appeared we can first look up the file in the *File* table and from there the project int the *Project* table with the PROJECTID.

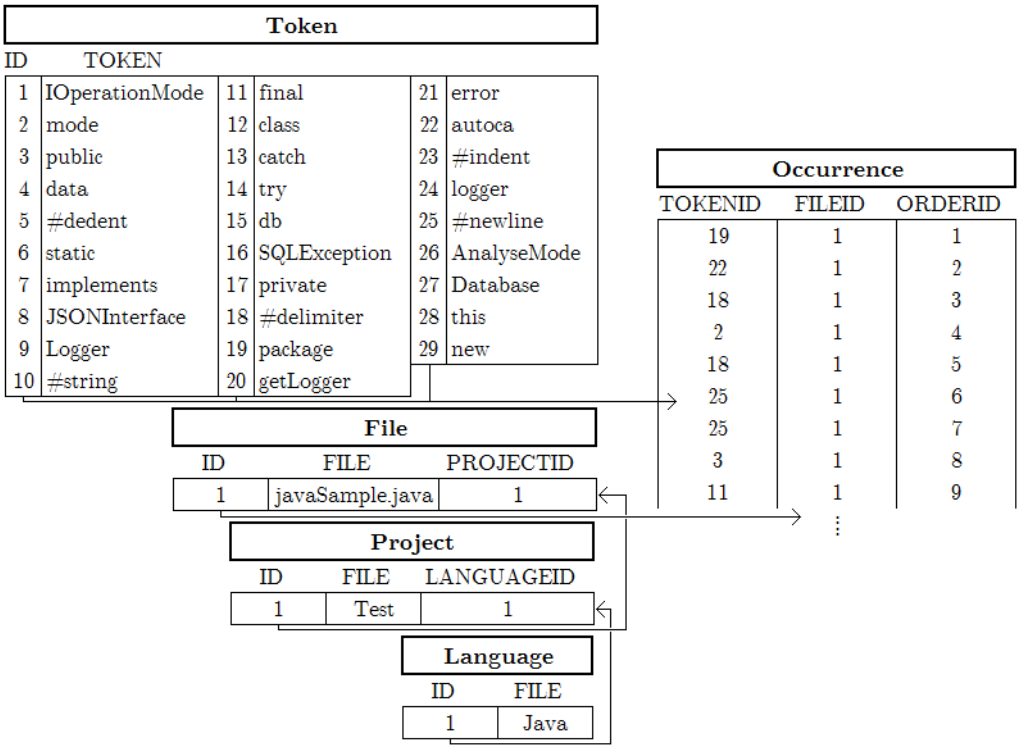

Figure 3.4: Database of the Java snippet Listing 1

The reason we introduced Ids was to save space in case strings appear redundantly. The size used on the hard disk by a string quickly exceeds the one of an integer. Also since integers have a fixed byte size we can calculate how much space entries in the occurrence table will require.

Methods run their analysis on the structure in Figure 3.4. Results are then stored in

separate tables introduced by the method. To add another method we could just add another result table and would not have to change any of the existing tables.

For example the Indent method applied on our Java snippet Listing 1 produces the table seen in Table 3.1.

Token	Count
public	2
private	1
try	1
catch	1

Table 3.1: Indent method result table

Since the Java snippet code file is very small the count of the tokens is low. In these results *public* is rated the most likely to be a keyword. *Private*, *try* and *catch* have the same likeliness. The Global, Coverage and Newline methods have similar result tables with tokens ranked differently.

3.3.2 Run Time Analysis

Several times in the development we ran into problems with the runtime. The compiled code ran but during execution something was causing slow downs. Using VisualVM from JDK we were able to detect the problems and solve them. There were two kinds of problems: repeated database accesses and out of memory problems during MySql query execution. Repeated database accesses were eliminated by pre storing results, prepared statements or only opening the connection once during a closed part of the execution. To get rid of memory problems we had to split up Sql queries into smaller parts and rejoin the result. We also tried to improve the performance using in-memory tables but the data space was to large to fit inside our system's resources.

3.3.3 Testing

To test our implementation we prepared sample source files for each language with special cases that could occur during parsing. After we had calculated the methods by hand we compared the results to those from AuToCa.

Test runs showed that there would most likely always be minor bugs in the process. As we started to scan bigger amounts of data there were always new test cases which had to

be added.

4

Evaluating the methods

In this chapter we are going to tokenize and analyse bigger Java datasets. To evaluate the results from the methods we will first introduce two evaluation variables: precision and rank. With those variables we will be able to draw conclusions. With the conclusions we will try to improve the process of automatically finding keywords.

4.1 Evaluation

Now we have an optimized program that produces a result table for each method. A result table contains the tokens found by the method sorted from highest count to lowest. The methods suggest that the tokens with the highest count are most likely to be keywords. The question is: How are we going to measure the success of a method?

Programming languages only have a fixed number of keywords. Intuitively it makes sense to define a cutoff point n, where we cut off the top n tokens from the result table. First we set the cutoff point to 50 which is the amount of Java keywords. This gives us 50 tokens found by each method which we expect to be keywords. To evaluate whether those 50 tokens really are keywords we compared them to the actual keywords of Java. This comparison gives us two groups of tokens: the ones that are keywords (also known as true positives) and the other ones that we expect to be keywords but are not (false

positives). The amount of tokens in the two groups true positives and false negatives yield our first variable:

Precision tells us the percentage of keywords in the top results to the cutoff point n.

$$\textbf{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}} = \frac{\text{keywords}}{\text{keywords+tokens}}$$

This measurement is important because it combines the positively identified keywords with the falsely identified ones.

After the first few comparisons of precision we realised that measuring precision only at one set point raised questions.

1. Where precisely in the top n results are the keywords?
2. Are the remaining keywords right after our cutoff point?
3. Which tokens are in the top n results?
4. Which method is the best at finding a specific token?

To answer the first two questions we changed the one set cutoff point to multiple cutoff points. Whenever an actual Java keyword appears in the result table we set a cutoff point and calculated the precision. For Java this gave us 50 precisions for each method see Table 4.1.

It is common to use recall as well. Recall is defined as the percentage of found keywords to all keywords we wanted to find. But in our case recall would not answer any of the questions we have. So to answer questions three and four we introduced a new variable:

Rank indicates the position of each keyword in the result table. *e.g.*, Table 4.3. Let's say we are interested in which of the methods yields the best result for the keyword *public*. The rank variable lets us quickly compare the results between the methods. Here the rank for *public* in the Indent method is first and second in the Newline method.

Evaluation example

To show what the evaluation variables look like in table form we will now evaluate the Java snippet from Listing 1.

Table 4.1 shows the precision table of the Java snippet. The first column indicates the name of the method. The columns *"1 to n"* are the precisions of the cutoff points from the first to the *n*th keyword (*n* is the number of keywords in the language). To clarify the precision values we calculate the fourth entry (0,571) in the row of the newline method. Using Table 4.2 we see that the fourth keyword found is *this*. We use *this* as a cutoff point which means that we ignore all results below it. Now we apply the precision formula on page 20 by dividing the sum of the keywords above the cutoff point (4) by the sum of all results above the cutoff point (7). This results in a precision of 0,571.

Method	1	2	3	4	5	6	···	n
Indent	1	1	1	1	0	0		0
Newline	0,5	0,667	0,75	0,571	0,625	0,667		0
Coverage	0,333	0,333	0,429	0,364	0,417	0,462		0
Global	0,167	0,222	0,3	0,286	0,333	0,333		0

Table 4.1: Precision table of Java snippet

Here are some example on how to express the values in this table:

- The precision for the Indent method at the cutoff point of the fourth keyword found in the top results is 1 (or 100 percent).
- The precision for the Newline method at the cutoff point of the second keyword found in the top results is 0.667 (or 66.7 percent).

Note: The Indent method drops from 100 percent directly to zero percent because it is not possible to find more than four keywords.

TOKEN	COUNT	KEYWORD
#newline	6	NO
public	2	YES
private	1	YES
implements	1	YES
logger	1	NO
Logger	1	NO
this	1	YES
catch	1	YES
try	1	YES

Table 4.2: Newline method result table of Java snippet

It is hard to compare the results of the methods by looking at numbers in a table. To visualize the results we generated graphs (see Figure 4.1) from the precision tables. The horizontal axis of the graph indicate the cutoff points from 1 to 50. The vertical axis indicates the precision. The actual data is a set of dots but we drew lines to see overlapping results better.

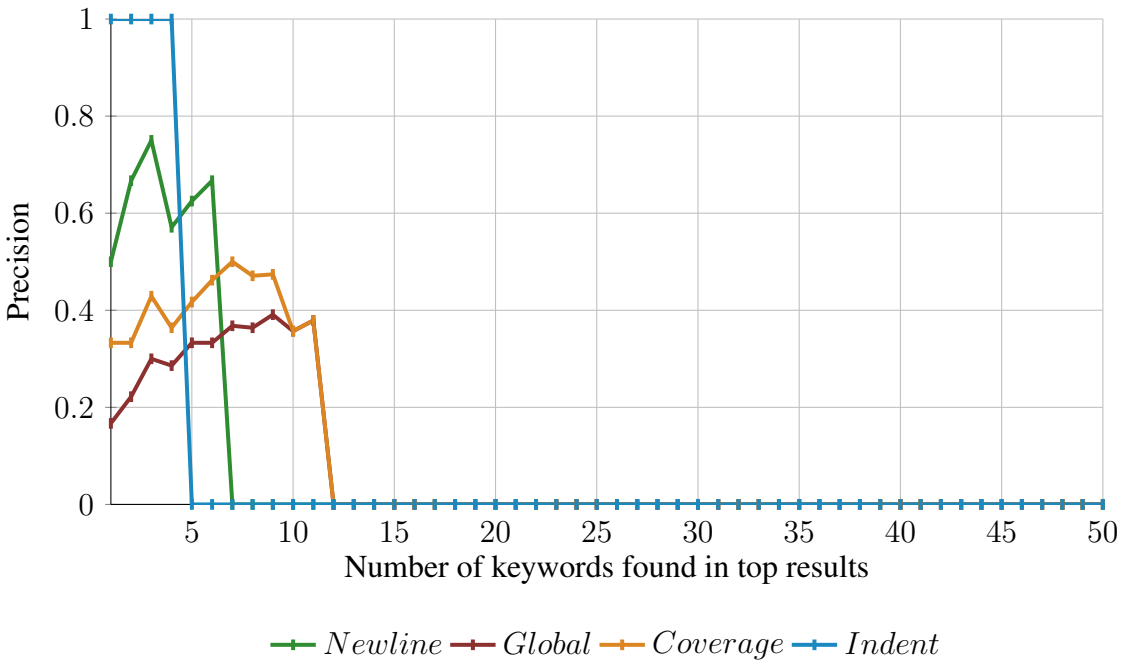

Figure 4.1: Precision graph example

Table 4.3 shows the rank variable. The first column is a list of all keywords of the analysed language. The following columns are the ranks which the keywords achieved in the result table given the method in the first row. Null value is given if the keyword was not found by the method.

- In the Java code snippet six keywords were found of 50. The rest did not appear or were not found in the code: indicated as null.
- The Indent method only found four keywords compared to the other three methods which found six.
- The rank of the keyword *public* is second for the Newline method and first for the rank method. This means that in the Newline method result table *public* appears at position two and in the Indent method result table at position one.

KEYWORD	Newline	Global	Coverage	Indent
public	2	6	3	1
private	3	22	17	2
implements	4	15	7	null
this	7	28	28	null
catch	8	18	13	3
try	9	19	14	4
abstract	null	null	null	null
continue	null	null	null	null
		⋮		

Table 4.3: Java snippet rank

4.2 Detailed evaluation of Apache Tomcat 7.0.39

In order to help the reader understand the results we perform a detailed and explained analysis of the Apache Tomcat project in this section. Table 4.4 shows the statistics of the Apache project.

Java Keywords	50
Files	1'609
Token Occurrences	1'908'227
Unique Tokens	21'184
Size MB	14

Table 4.4: Statistics of the Apache project

To improve the results it is important to know what the tokens that were falsely identified as keywords look like. That is where the plain result tables of the methods help. Table 4.5 shows the top part of the plain results of the Apache project of the Global method.

16618	public	1924	private	673	case	312	return
14898	if	1906	protected	605	log	265	String
4310	else	1880	for	432	synchronized	177	static
2561	catch	986	while	393	finally	157	new
2455	try	722	throw	356	digester	148	throws

Table 4.5: Global Apache Tomcat 7.0.39

In Table 4.5 the number to the left of the token indicates the number of times it was found. Tokens with a hash tag are already recognized and marked by the tokenizer. We inserted them into the database to keep the structural integrity of the data. As an example the Indent method needs the indents to find keywords. If we would have removed the *#Indents* already recognized by the tokenizer the Indent method would not work properly.

Table 4.6 shows the top part of the results from the Indent method:

834708	#delimiter	28822	#string	12744	null	7003	Override
295727	#Newline	20684	public	11576	import	6555	this
57813	#Indent	17656	if	9955	new	6403	org
55906	#dedent	16551	String	8925	int	5961	private
33334	#comment	13003	return	8854	void	5788	apache

Table 4.6: Indent Apache Tomcat 7.0.39

Both results seem to already contain quite a few Java keywords. To further inspect how good they are we inspect the evaluation table of the precision variable (Table 4.7). With this table conclude some pretty important information. The Indent method finds 11 keywords with a precision of 100 percent and gives the best result for 26 keywords with 53,1 percent. There is also a downside to the Indent method. It only finds 39 tokens while Coverage and Global both find 47.

Method	11	···	26	···	39	40	···	47	48
Indent	1		0,531		0,061	0		0	0
Newline	0,688		0,464		0,044	0,031		0	0
Global	0,5		0,371		0,108	0,102		0,002	0
Coverage	0,5		0,481		0,153	0,097		0,003	0

Table 4.7: Precision Apache Tomcat 7.0.39

Figure 4.2 shows the graph of the Apache precision table.

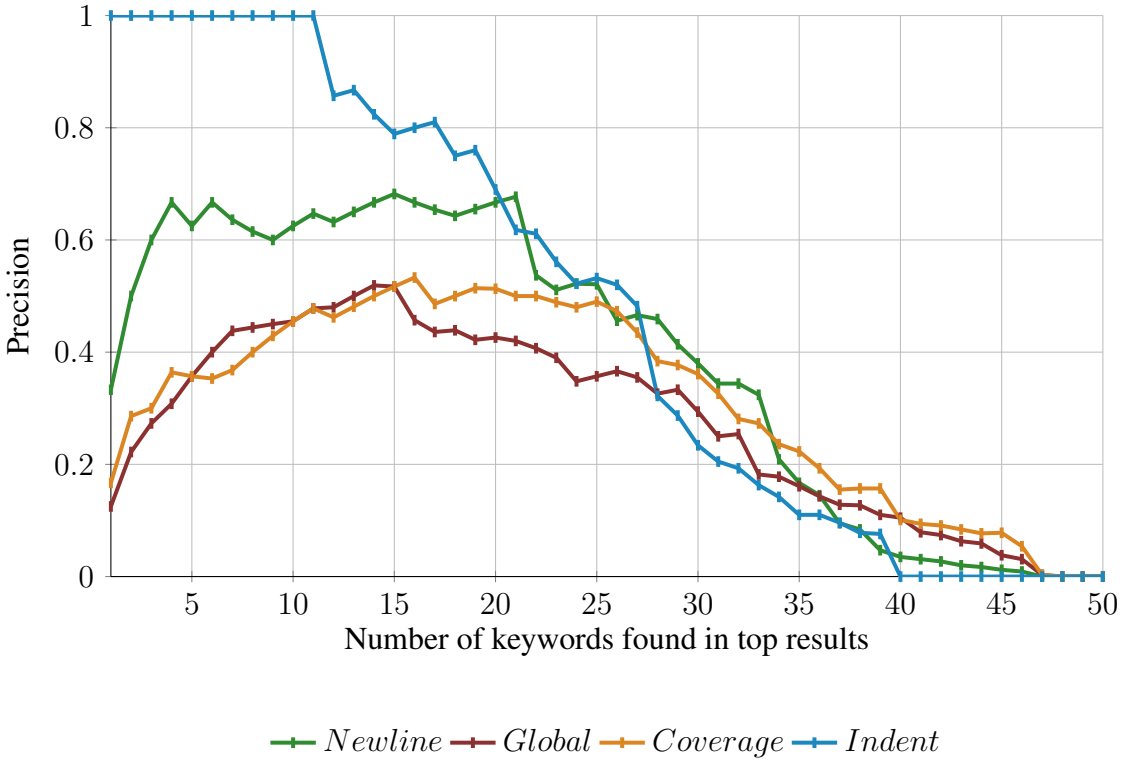

Figure 4.2: Precision graph Apache project

The question is: how are we going to improve those results? We had two ideas:

Filtering we try to find commonalities between the wrongly identified keywords and eliminate them by filter.

Adding more data lets us filter out project specific keywords. This gives us a more average result over all the data.

4.3 Filters

Filters should be adjustable, easy to remove and reorderable in different sequences. With those properties in mind we changed our analysis mode and introduced filter chains. The chain pattern was used to design the class structure. A filter chain can be specified in the configuration file. The chain starts with an input like the result table from a method. Then various filters can be applied to those results. At the end of a filter chain an output filter is applied to its result. With those filters we can assemble a test and run different filter chains on the same data set.

By looking at the result tables we notice four different groups of wrongly identified keywords: remains of the tokenizer, project specific keywords, keywords of excessive length and tokens with upper case letters *e.g.*, method literals.

Scan mode filter All remains of the tokenizer are already tagged with a hash tag. So removing them is just a matter of removing all tokens with hash tags.

Upper case filter Java keywords only contain lower case letters. By eliminating tokens with upper case letters we should get better results.

Length filter Keywords in Java are rather short. The longest one is synchronized with 12 characters and the shortest are 2 characters. Filtering too long words and too short words could help improve the results.

Intersection filter While scanning multiple projects each one introduces project specific expected keywords to the end result. What stays the same between the results of the projects are the keywords of the language. The filter has one variable which indicates how many projects a token has to be in to appear in the end result.

Filtered Apache Results

Figure 4.3 shows the unfiltered and filtered precision graph of the Apache project. Only the intersect filter has not yet been applied since it needs more than one project to work. The fair dashed colors indicate the unfiltered and the bright ones the filtered results. The filters highly improved the top 25 token of Global(red), Coverage(orange) and Newline(green). Indent can now find 11 keywords with 100 percent precision. Another improvement is that Coverage, Newline and Indent now score a solid precision above 65 percent : finding 25 of the 50 tokens. At 33 tokens Newline spikes again with 50 percent precision but at 47 tokens Coverage is the best solution.

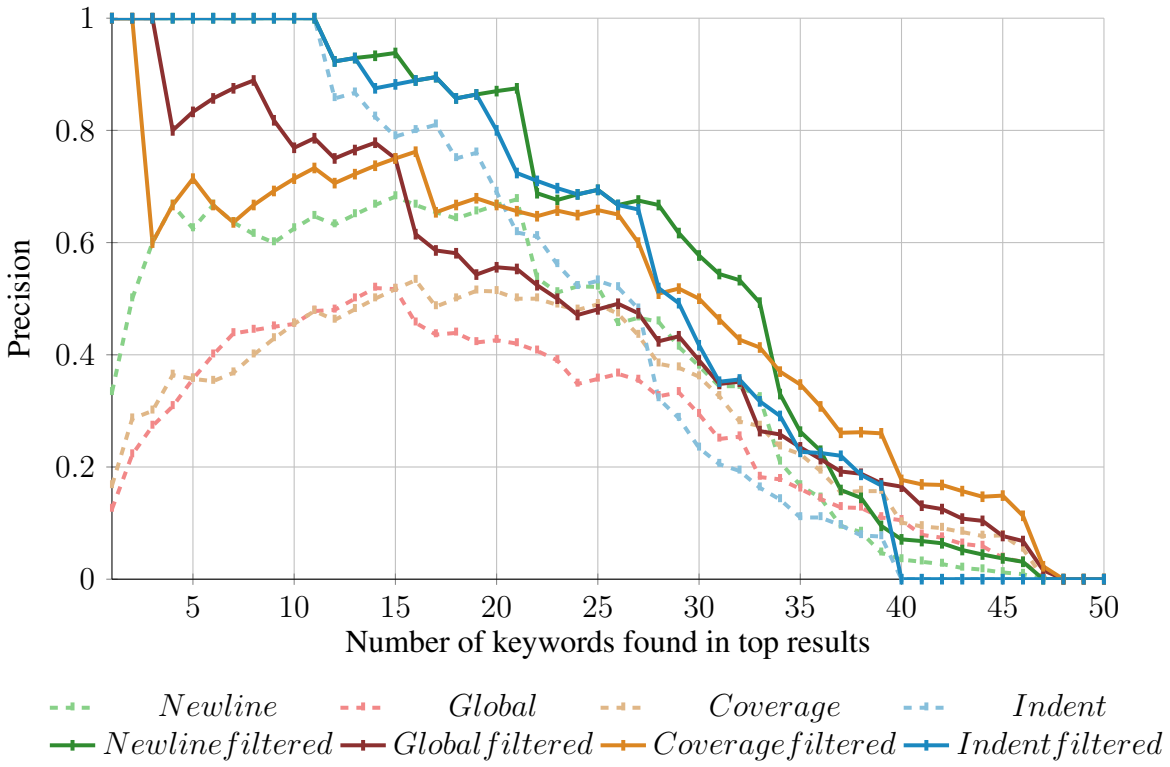

Figure 4.3: Filtered precision graph Apache project

To find out more about the results let's look at the falsely identified keywords in the result tables of Coverage and Newline. Is it possible to filter more? Table 4.8 shows the result table of the Newline method on the left and the result of the Coverage method on the right side. This table indicates that many of the highly ranked tokens which are not keywords will be very hard to get rid of such as: *log*, *out*, *result*, *io*. They are very common words in many projects and just by glancing over the results almost indistinguishable from the

keywords.

But there is hope since terms like *apache*, *catalina*, *digester* or *tomcat* clearly are in the result because we only scanned the Apache project. With more data and the intersection filter which removes tokens that only appear in few projects we should be able to get rid of these and get better results.

TOKEN	COUNT	KEYWORD	TOKEN	COUNT	KEYWORD
public	20461		public	1602	
if	14357		package	1600	
return	11624		org	1398	NO
	⋮		apache	1397	NO
log	1883	NO	class	1391	
throw	1845		import	1363	
for	1815		void	1169	
package	1600		java	1142	NO
throws	1596		return	1087	
out	1218	NO	util	1037	NO
new	1169		static	1002	
while	941		throws	980	
sb	897	NO	final	979	
super	889		private	968	
case	841		new	968	
boolean	826		null	945	NO
break	794		int	927	
result	644	NO	if	925	
writer	588	NO	extends	827	
buf	581	NO	this	814	
request	518	NO	boolean	715	
digester	501	NO	io	713	NO
context	489	NO	javax	702	NO
ctxt	481	NO	tomcat	693	NO
long	479		catalina	684	NO
tomcat	413	NO	else	618	

Table 4.8: Falsely identified keywords: on the left side the Newline and on the right the Coverage method.

4.4 Adding more Data

It is likely that scanning just a single project like Apache does change the result compared to what we would get when scanning random source code on the internet. This is due to the fact that Apache is a well known and managed project with style guides which is not the case for most of the code out there. So are the results of the Apache project falsified? Does more data improve results or worsen them? To get a more average result we added randomly selected Java projects from Github. Compared to the Apache project the Java projects from Github contain about 47 times more occurrences.

Projects	179
Files	100'764
Token Occurrences	89'018'672
Unique tokens	415'052
Size MB	554

The intersection filter is influenced by the amount of data added. The setting deciding in how many projects a keyword appears improves the results the more data we add. Figure 4.4 compares the results from the Apache project to the results with the random data. The fair colors are the filtered Apache results from the chapter before and the bright colors the new data.

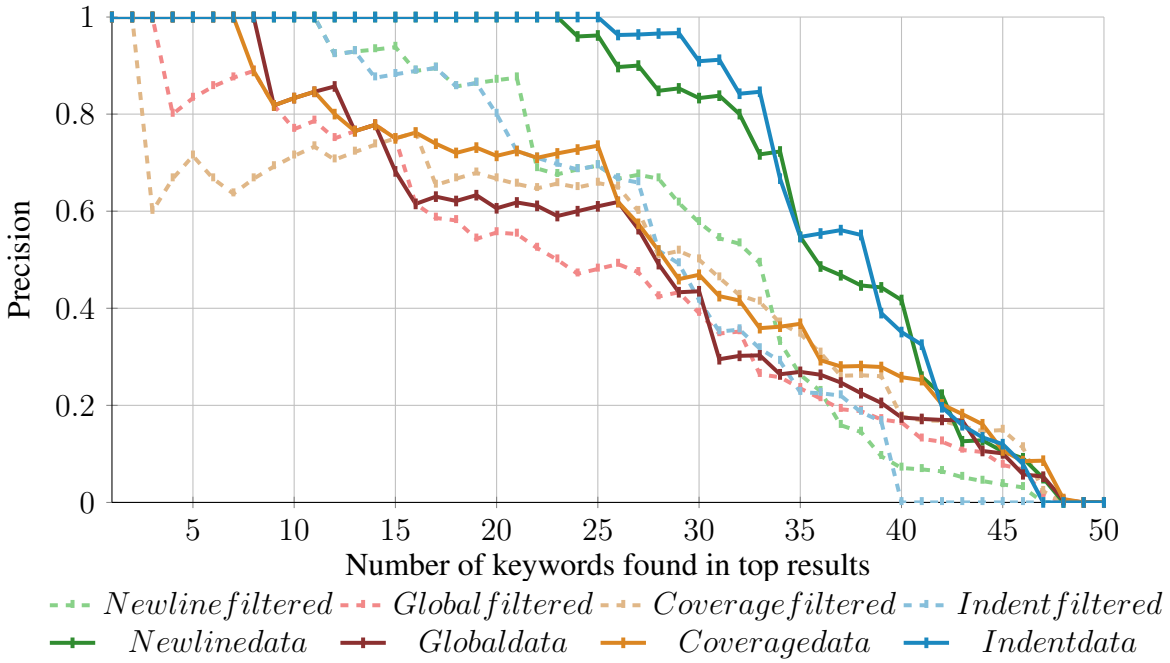

Figure 4.4: Comparison: Apache vs 179 random projects

The Newline and Indent method both made big improvements in precision. They both can find 25 tokens with 100 percent precision. But after 33 they both drop quickly. At 41 they still reach around 30–40 percent precision. Interestingly Global and Coverage seem to have only improved slightly.

Adding the intersection filter

With a dataset of 172 projects it is likely that highly ranked tokens which are not keywords appear in more than one project. Setting the value of the intersection filter to a higher one could improve results. If we set the value of the intersection filter higher: tokens must appear in more projects to be considered keywords.

Figure 4.5 shows the Global method with different values for the intersection filter. Darker colors mean the token has to appear in more projects to be considered a token. As we can see the intersection filter does improve the results if we want to find fewer than 38. But it comes with a trade off. If there is a token that is harder to find it will get deleted from the result set. This is shown in the range between 39 to 48.

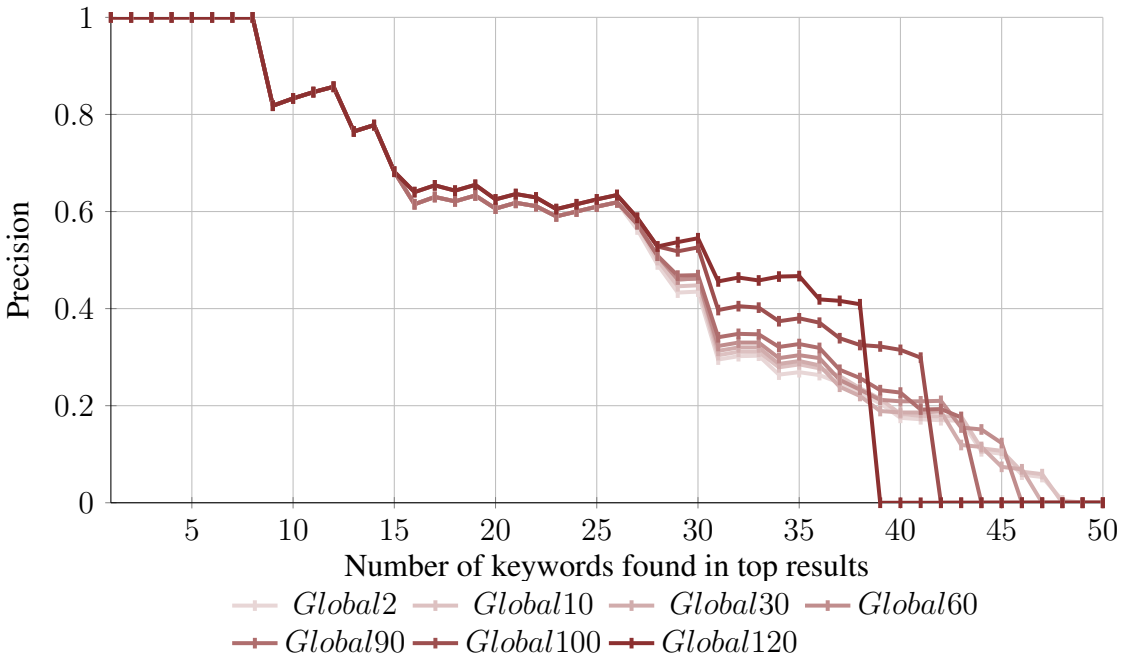

Figure 4.5: Intersection filter on the filtered Global method

Table 4.9 shows the precision data:

FILTER	37	38	39	40
Global 2	0,247	0,225	0,205	0,175
Global 10	0,261	0,236	0,214	0,183
Global 30	0,239	0,22	0,189	0,186
Global 60	0,253	0,233	0,212	0,209
Global 90	0,274	0,257	0,232	0,227
Global 100	0,339	0,325	0,322	0,315
Global 120	0,416	0,409	0	0

Table 4.9: Precision Global with different intersection values

Figure 4.6 indicates that the same effect applies to our current best method Newline. Compared to the Global method we see a stronger bend at Newline 120 and only slight improvements in between.

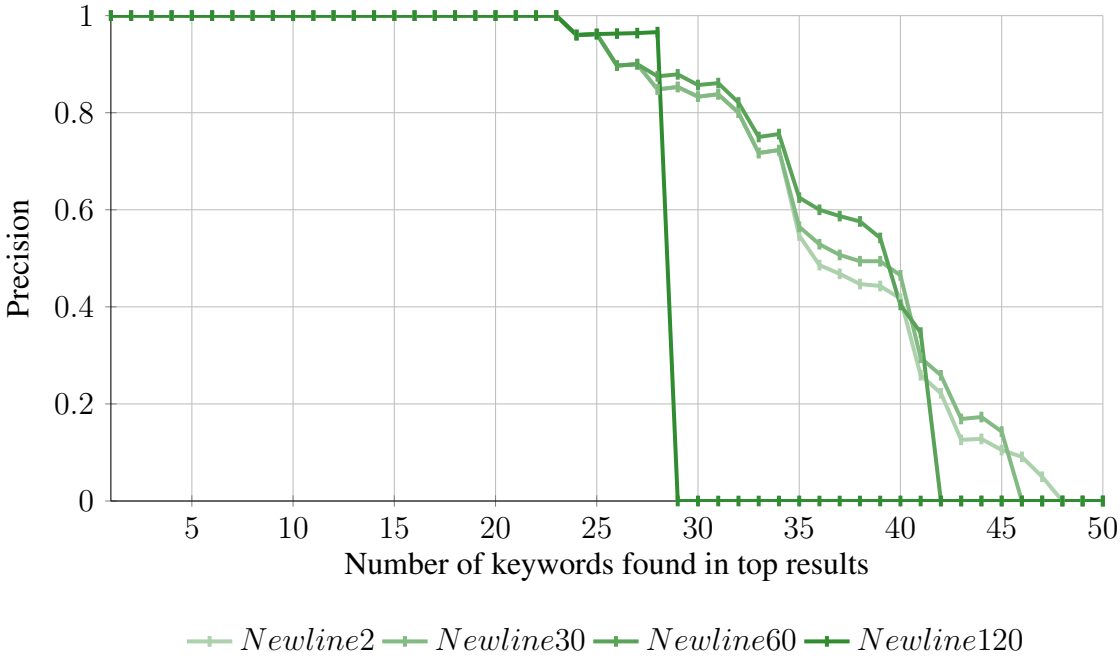

Figure 4.6: Intersection filter on the filtered Newline method

In conclusion the Intersection filter improves the results when kept at a low value. The tokens left over in the results will be more likely to be keywords. When moving it to a higher value there will be a significant trade off in the form of lost keywords.

5

Applying AuToCa to new languages

In this chapter we are going to test AuToCa on different programming languages. The idea was to select the most commonly used languages for which enough source code is available. Since an assumption was that programming languages are similar in syntax we selected C, C++ and Python. To have a broader sample we also tested completely different ones like Haskell and Clojure.

There is a subsection for each language. It consists of a short introduction comparing the language to Java with a short piece of sample code. Followed by statistics on the dataset from Github and finally the results and the conclusion. The results are always a comparison diagram of fit and normal.

Fit means that if the language had a different comment syntax or special characters inside the token we adapted the configuration file accordingly. Normal means that we used the settings adjusted for Java.

At the end of this chapter we will be discussing the problems that occurred while scanning the new languages.

5.1 Adapting AuToCa to new Languages

In Chapter 4 we adapted AuToCa to find the keywords specifically for Java. Now that we try to find keywords for different languages it will be interesting to see what problems occur with AuToCa when new source code is scanned and analysed. As part of the diagnosis we will also be adapting the AuToCa configuration to the new languages and compare the results to the standard Java one in the hope of reaching new conclusions.

5.2 Python

```
#Style Table
""" An table reader and writer.
"""

from __future__ import absolute_import,
from ...table.column import col_getattr

class BasicHeader(core.BaseHeader):
    start_line = 0
    comment = r'\s*#'
```

Listing 6: Python Sample

Python is similar to Java in the way that all keywords are written in lower case and contain no special characters. Instead of curly braces to structure content Python uses Indents which sounds promising for the Indent method. The dataset we collected has the following statistics.

Python Keywords	31
Projects	241
Files	40’343
Token Occurrences	61’845’944
Unique tokens	465’367

Figure 5.6 shows the precision graph of Python. The results come pretty close to the ones of Java. Though compared to Java, Python has only 29 keywords. The difference is that the filters are not fit to Python. We can see that the fit changes in the configuration do not always improve the result but also do not worsen them significantly. The reason for the varying results are the comments which have a different syntax in Java and Python.

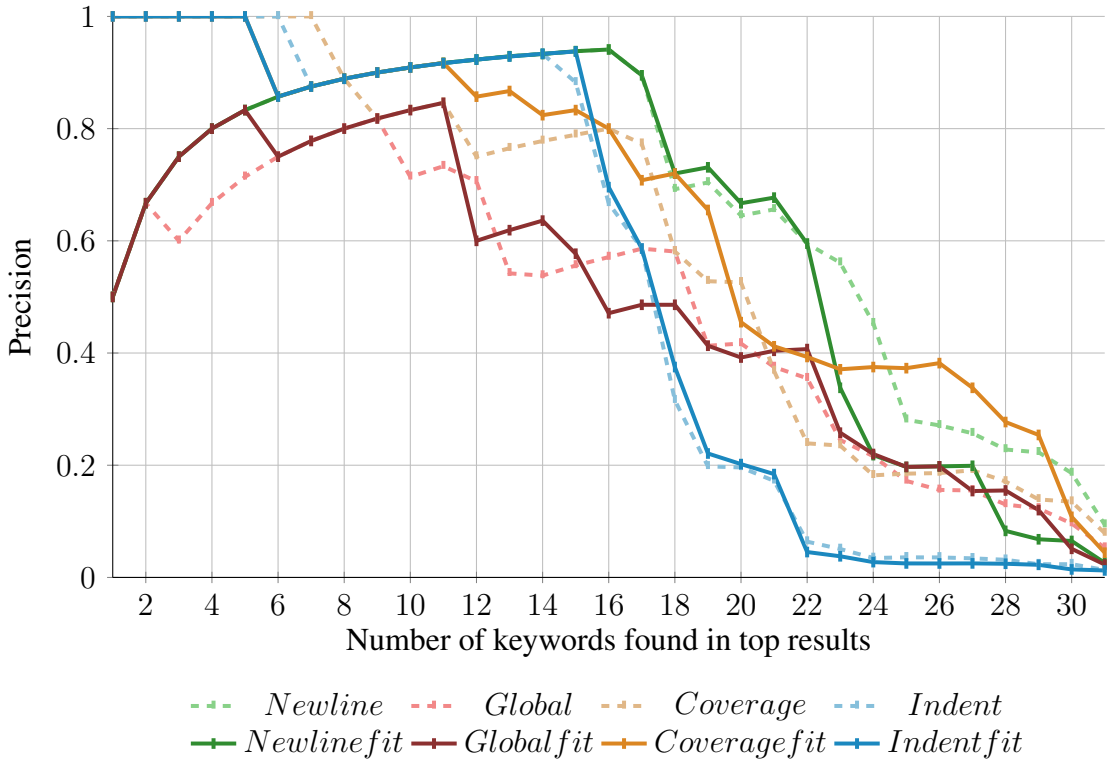

Figure 5.1: Python data set precision results of the fit configuration in fair colors compared with the not configured in darker colors.

5.3 C and C++

```c
#include <string.h>
/*C Sample*/
FILE *fp;

int main ( void )
{
    unsigned int ra;

    if(fp==NULL) return(1);
    fread(data,1,sizeof(data),fp);
}
```

Listing 7: C Sample

C and C++ are syntactically closely related to Java. They have the same kind of comments and use curly braces to structure code. A big difference is that keywords in C and C++ can contain special characters and numbers. There are also keywords with upper case letters. Table 5.3 shows the statistics of the C data set.

C Keywords	44
Projects	171
Files	18’370
Token Occurrences	67’709’888
Unique tokens	623’050

What we found with C is that the different C standards can cause problems when finding keywords. We used the C11 standard for the evaluation because it has the most keywords. In the graph keywords from 33 upwards are mostly C11 standard introduced keywords. With the ANSI C recognized keywords we achieved similar results to Java. The fit result made not a big difference because the comments have the same syntax. If the fit result did perform worse it was because we added underscores to the setting of the keywords.

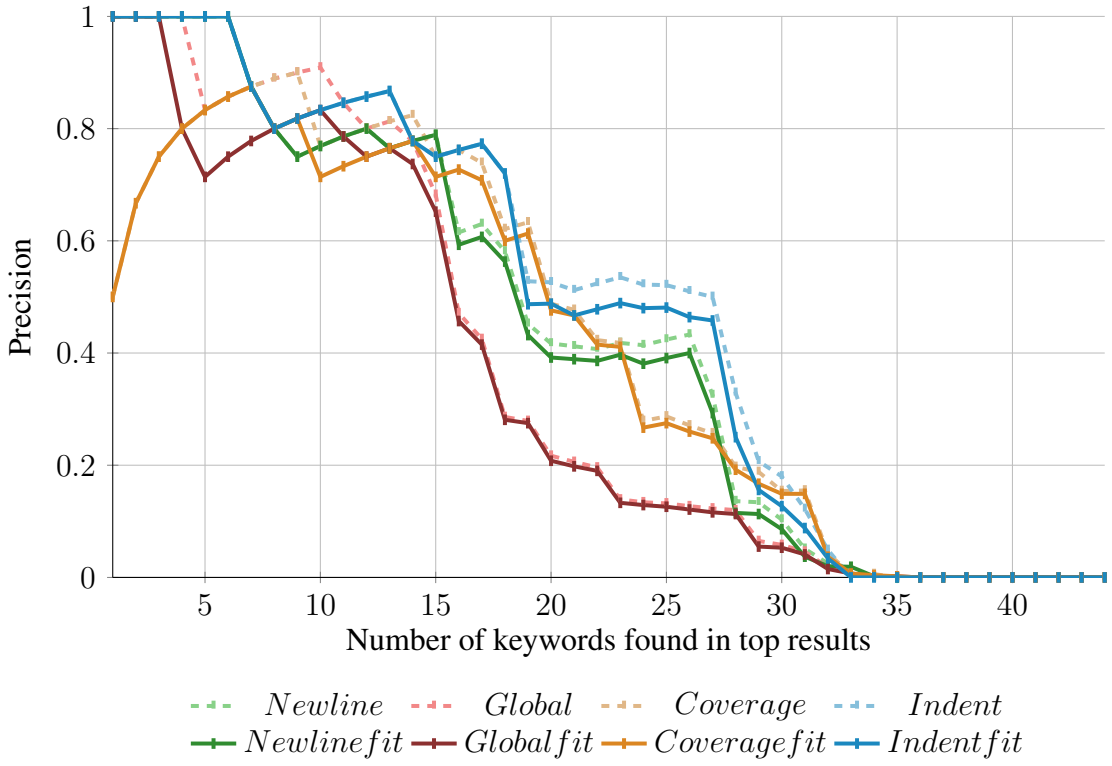

Figure 5.2: Precision fit vs normal C

and now the C++:

C++ Keywords	86
Projects	158
Files	23’718
Token Occurrences	65’035’401
Unique tokens	461’033

What is special about C++ is that in the current state AuToCa produces an unusable table if the settings are not fitted to C++. The reason is that the keywords contain not only special characters but also if scanned without correction contain multiple instances of the same token e.g. char → char, char16_t → char, char32_t → char. So we only did a scan with the configured settings to contain special characters in keywords.

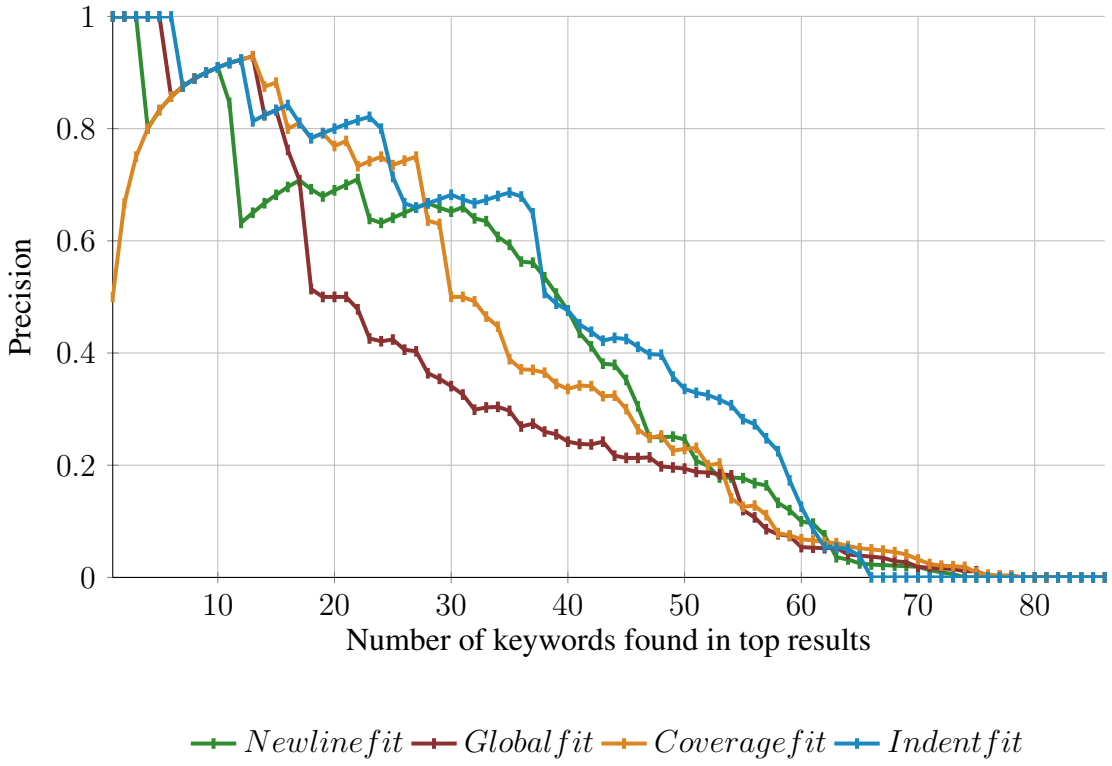

Figure 5.3: Precision fit C++

5.4 Shell

```
# Find the proper handle
local __customtoolpath=$3

function find_tool () {
   # For each possible
   for i in "${__possibletoollocations[@]}"; do
      local __commandlinetool=$(type -p $i)
   done
}
```

Listing 8: Shell Sample

Shell or Bash (Bourne-again shell) keywords are written in lower case letters and do not contain special characters. Bash is not object oriented. Multiline comments are not in the language but can be achieved with a hack: this is a problem because the hack cannot be detected with a regular expression. In the data set many projects contain only 1–2 files Table 5.4.

Shell Keywords	17
Projects	951
Files	7’971
Token Occurrences	4’009’554
Unique tokens	51’105

With shell we got the best result with the Indent method. None of the other methods were able to get close. The fit of the configuration generally improved the result.

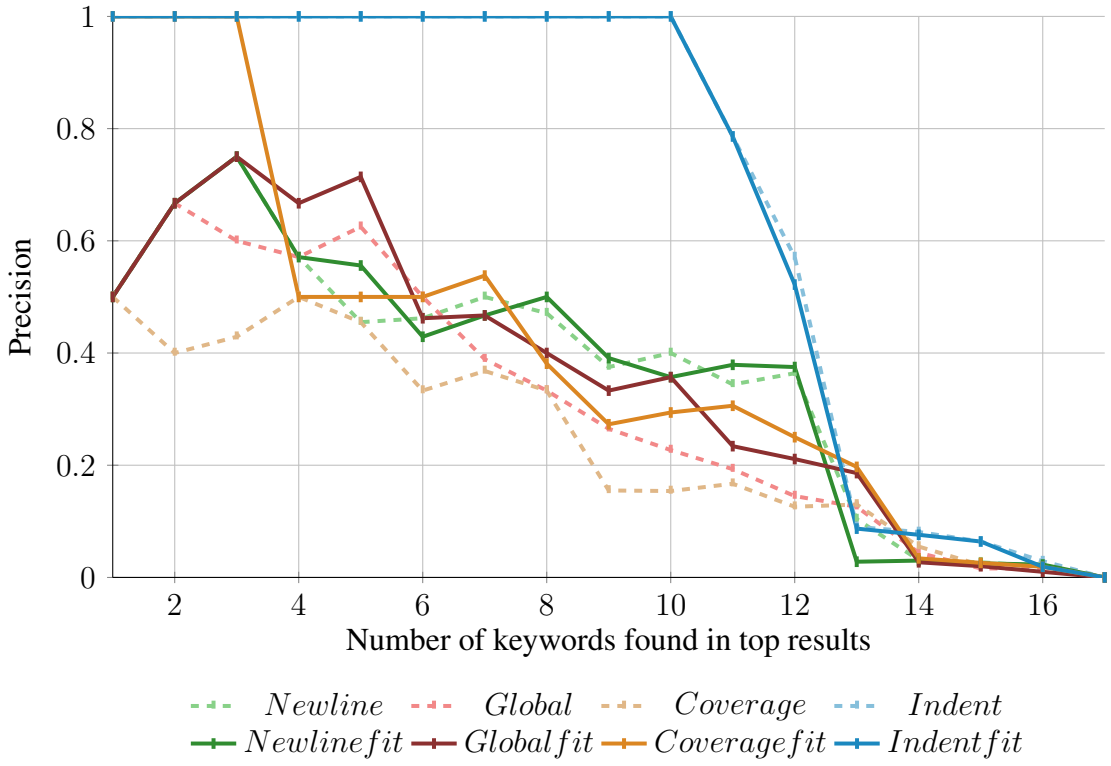

Figure 5.4: Precision fit vs normal Shell

5.5 Haskell

```
1 -- Main
2 main :: IO ()
3 main = do
4  args <- getArgs
5  case args of
6    ("help":_) -> putStrLn notice
```

Listing 9: Haskell Sample

What makes Haskell interesting is that it is a purely functional language. The keywords are in lower case and kept short.

Haskell Keywords	20
Projects	409
Files	20’087
Token Occurrences	18’822’050
Unique tokens	238’753

The results for Haskell were surprisingly good —almost as good as Java with both settings fit and normal. Indent four 9 out of 20 keywords with 100 percent precision.

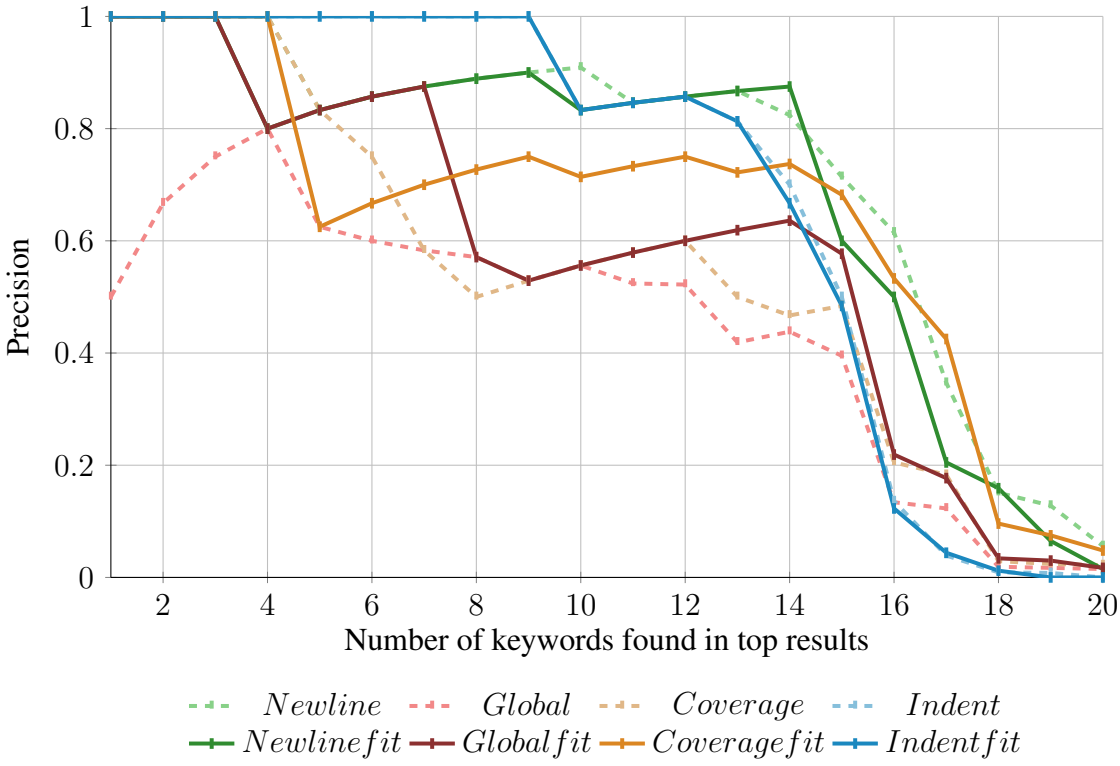

Figure 5.5: Precision fit vs normal Haskell

5.6 Scheme

```
1 ; Define this Syntax
2
3 (define-syntax pmatch
4   (syntax-rules (else guard)
5     ((_ v (e ...) ...)
6      (pmatch-aux #f v (e ...) ...))
```

Listing 10: Scheme Sample

In Scheme, a dialect of Lisp, keywords are completely different from Java keywords. Scheme keywords tend to be long and contain special characters to separate parts. In fact there are only few *"real"* keywords and a standard library of functions. To compensate for this we selected a set of base functions and keywords to see if we could find them.

Scheme Keywords	55
Projects	296
Files	11'582
Token Occurrences	17'967'765
Unique tokens	183'845

With Scheme we had the first big problems applying AuToCa. The special characters in the keywords resulted in a huge database. Only after fitting the configuration file we were able to scan the dataset. Still the Indent method did really well compared to the others.

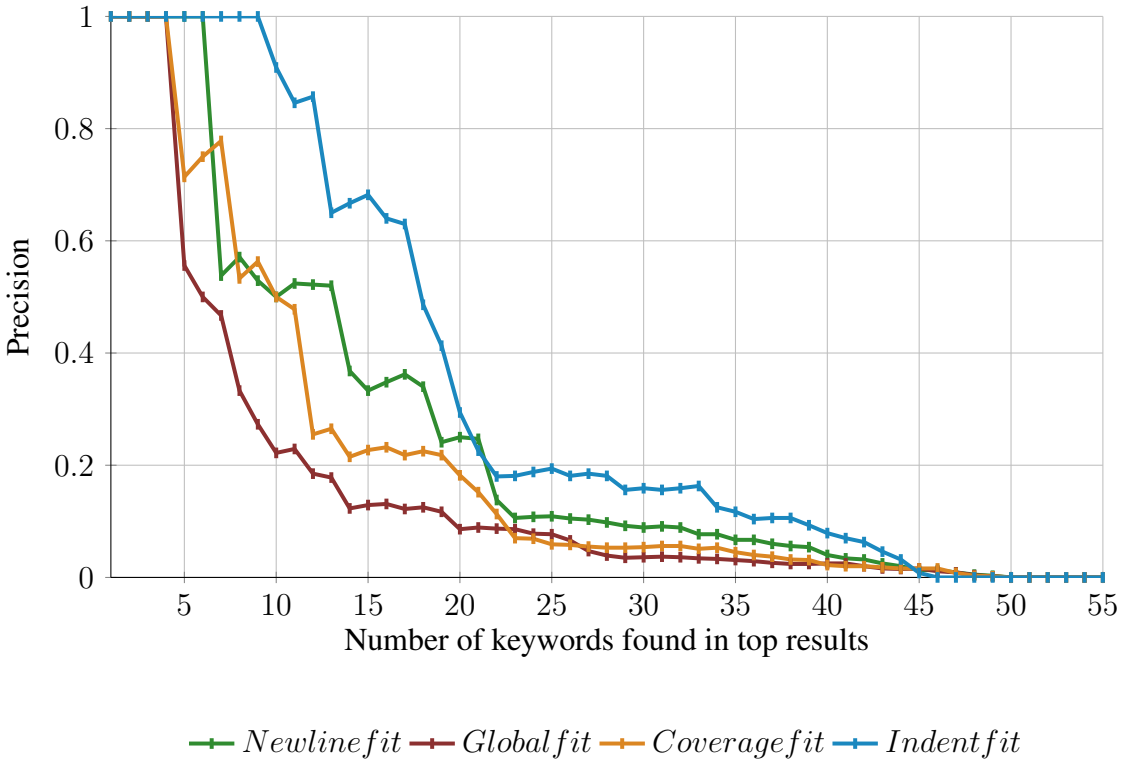

Figure 5.6: Precision fit Scheme

5.7 Clojure

```
(defn save-solution [user-id problem-id code]
  (update :solutions{:user user-id:problem
 problem-id}{:$set {:code code}}:upsert true))
```

Listing 11: Clojure Sample

Like Scheme Clojure is a dialect of Lisp. Clojure does not really have keywords but core functions. It is impossible to evaluate if AuToCa can find Clojures keywords if there are none. It is still interesting what the output of AuToCa looks like.

Clojure Keywords	0
Projects	987
Files	16’631
Token Occurrences	16’551’211
Unique tokens	57’113

defn	78078	ns	29437	in	18827
let	70631	core	28230	first	17687
name	52867	get	27902	deftest	17107
is	50286	type	26682	path	16854
fn	42905	def	24690	string	16777
clojure	42610	TRUE	24618	this	16752
map	40756	value	21687	count	16752
if	39485	to	20460	set	16587
test	36309	file	20407	body	16282
id	35780	not	20066	line	16042
as	35283	args	19926	key	16016
nil	34551	when	19354	keys	15945
str	32587	with	19200	or	15751

Table 5.1: Global method Clojure

defn	73122	fact	5349	catch	2247
let	59826	with	4987	import	2233
if	25043	cond	4892	binding	2200
deftest	16609	do	4745	it	2182
is	14073	doseq	4523	for	2056
when	13907	use	3356	reduce	1880
ns	12982	try	3129	str	1850
fn	9815	loop	3038	are	1828
def	9641	map	3013	defprotocol	1585
testing	7785	name	3011	extend	1552
require	7168	and	2775	doc	1545
defmacro	6637	or	2752	defproject	1448
defmethod	6399	assoc	2484	throw	1423

Table 5.2: Indent method Clojure

5.8 Problems while adapting

Many problems occurred while we adapted.The following problems have to be addressed if we want to find a method that works better:

Indents are basically a set number of white spaces. If the number of white spaces in an indent is not the same as the one we set in the configuration of AuToCa the Indent method will fail. It might be the case that there is a mix of indenting styles in a code file. We only counted indents with a fixed number of white spaces and ignored the rest.

Keywords with special characters settings in one language can disturb the results in another one. To solve this analysis on what is a token has to be done beforehand. There are many variations of special characters in keywords. But it seems that there are special characters that are more likely to be found in a keyword than others. Heuristics might be able to help guess what those rules are by presenting the user with subsets of solutions with different kinds of settings.

Comments different languages have different comments. There are many variants of commenting styles. Comments can be long and have a big influence on the results.

Escape characters strings in code can be long and have an influence on the results e.g an escaped character inside a string can cause a whole file to fail. To detect an escape character the escape sequence has to be known.

Dialects and Versions of programming languages different dialects can have different character sets for keywords or introduce new keywords.

6 Conclusion and Future Work

6.1 Conclusion

Finding keywords can help the process of inferring a grammar, which is an important part of building a parser. In this thesis we have tested four methods to find keywords in source code of an unknown language. If those methods really would yield usable results was unknown.

Our goal was to develop a tool which could automatically find the keywords in any source code. To test the methods on how to find the keywords we had to develop a tool which lets us easily implement and verify new methods. It should allow us to build tests which automate the calculation needed for the evaluation of a method. We achieved this with AuToCa. The four methods we wanted to test were basic in the sense that they are not complex algorithms.

The methods were Global, Coverage, Newline and Indent. The Indent method suggests that a keyword always appears at the first position on the line before an Indent; the Global method that keywords appear most frequently; the Coverage method that keywords appear in most files and the Newline method that the first token of a line is a keyword.

Our results indicate that Indent is the best method overall. But it also has a big limitation namely that it can only identify keywords that appear at the beginning of lines.

It is surprising that such a simple method can already find most of the keywords in similar languages. It makes it easy to imagine what could be achieved with a more complex method for example by combining Indent with another method that could fill its weaknesses.

We ran into several issues configuring AuToCa to other languages that are similar to Java. Special Characters, dynamic typing and Indents can all be result breaking.

6.2 What is missing?

AuToCa should be able to give feedback on changes of single configuration values *e.g.*, the optimized setting for the intersection filter. To get this feedback time consuming filter chains have to be tested with only one setting changed on each configuration. This ends in redundant calculations.

In some of the randomly downloaded source code there were large tables in normal source files. The tokenizer slowed down quite a bit on those files because it was traversing random character sequences as if they were code. The tokenizer ended up tokenizing single characters and numbers which is time consuming. A time limit mechanism should have been added to AuToCa which in case a single file takes too long just skips it. Since those files would not affect the results we deleted them for simplicity. Another way of removing those files could have been to limit the byte size of a file. Files bigger than 1 Megabyte are rare.

Duplicate files in open source projects are highly possible. With a redundancy test for source code files duplicate scanning could have been prevented.

6.3 Future

The problem of finding keywords is clearly a statistical one. Deeper knowledge of statistics and machine learning would give a new perspective on the whole topic and could even be the solution.

An important part of improving AuToCa would be to have less assumptions and the same results. There are different ways of achieving improvements for example by automatically recognizing comments. Through natural language detection it is theoretically possible to identify comments without knowing anything about the programming language. Another improvement could be to automatically recognize code of the same language by statistical analysis.

One thing we did not take into account is the project size. In the intersection filter every project has the same value. Logically this does not make sense if we consider that the results from the Apache project have the same value as the ones from a project with a single code file. Clearly Apache brings the better average to the table. This should be factored in.

At the end we had the results of the four methods Global, Coverage, Newline and Indent. It would be interesting to see if even better results could be achieved by normalising and intersecting these results.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Alpana Dubey, Pankaj Jalote, and SanjeevKumar Aggarwal. Inferring grammar rules of programming language dialects. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *Grammatical Inference: Algorithms and Applications*, volume 4201 of *Lecture Notes in Computer Science*, pages 201–213. Springer Berlin Heidelberg, 2006.

[2] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009.

[3] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[4] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[5] Oscar Nierstrasz and Jan Kurs. Parsing for agile modeling. *Science of Computer Programming*, 97, Part 1(0):150–156, 2015.

[6] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012.

[7] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

7

Anleitung zu wissenschaftlichen Arbeiten

This chapter consists of an additional documentation and a user guide on how and where to get the results presented in our thesis.

7.1 Introduction

AuToCa is a tool to test methods on how to find the keywords in source code of an unknown programming language. Methods are partly hard coded but the filters that are applied on the results are configurable. To analyse a method we can run it with different settings. In the configuration file we describe the different kinds of settings we want to run. Also in the configuration file we describe the path to the source code. In our resources we have the configuration files and the dataset with which we achieved the results in the thesis.

Using the command line we can specify the location of the configuration file and choose which mode we want to run. There are three modes:

- **TOKENIZE** tokenizes the source code and stores it in the database.
- **ANALYZE** analyzes the tokens in the database using the configuration file.
- **BOTH** sequentially executes TOKENIZE and then ANALYZE.

7.2 Getting started

This section gives an overview on how to replicate the results from the thesis with the preconfigured configuration files.

What do I need to run AuToCa?

- The only runtime dependency needed is Java 8.

Where can I get AuToCa?

- Our repository is stored on Github, from there you can clone it using git and the command:
  `clone https://github.com/jogug/AuToCa.git`
- Or you can download the .zip file from:
  `https://github.com/jogug/AuToCa/archive/master.zip`

How to run the already compiled autoca.jar?
As mentioned in the introduction you need a configuration file and source code to run AuToCa. The prepared configuration files are located under *AuToCa/resources/configuration/*. The data is compressed in testprojects.part01.rar-part09.rar under *AuToCa/resources/*. Extract the data into *AuToCa/resources/testprojects/*. Next start up the terminal and navigate to the root directory of AuToCa.

Now execute AuToCa by modifying the following command:

```
java -jar Xms -Xmx [jar] mode configuration [path]
```

- `Xms`, `Xmx` specify the minimum and maximum memory allocation pool for a Java Virtual Machine
- `[jar]` is the name of the jar
- mode can be `both`, `tokenize` or `analyze`
- configuration can be `path` or `default`, it defines whether the default configuration file is loaded or one from a path
- `[path]` if in the previous parameter path is selected a path to the configuration file is needed here.

e.g: `java -jar -Xms1024M -Xmx3048M autoca.jar both path C:/Users/name/ws/AuToCa/resources/configuration/java.cfg`

When AuToCa is finished, the results can be viewed using the H2 database. You can read more about this in Section 7.4

How to build AuToCa from source code?

- Install Gradle and build it directly using the command: `./gradlew build` in the root directory *AuToCa/*. On Windows we used git bash to execute the command.

7.3 How to create a configuration file

The configuration file is a Json file with a fixed structure. For the most part the configuration file is self–explanatory, here are the settings that need directions:

languages contains basic information about the languages to be scanned.

- *projectSizeLimit*: limits the size of the source code to a desired limit.
- *minAmountOfProject*: minimal amount of different projects in the source code.

database contains all information concerning the database and tables.

- *outputLocation* the output location of the database file

filterchains contains the order of the filters applied to a results of a specific method. A filter chain has three settings:

- *resultName*: the name of the result table
- *languages*: the names of the languages the filter chain is performed on. The language name must be the same as the one defined in *DefaultLanguages*
- *filters*: A list of filters to be applied from bottom to top.
  - *Always has to start with:* NewlineFilter, GlobalFilter, RealIndentFilter or CoverageFilter.
  - *Followed in any order by:* IntersectFilter, SubStringFilter, UpCaseFilter.
  - *Ends with:* Output.

tokenizer contains the regular expression for the different tokens.

tokenhandler contains tokenizer flags for special tokens and the maximum and minimum token length.

Configuration file example

Listing 12 is an example of a configuration file. Configuration files are stored under `AuToCa/resources/configuration/` with the ending `.cfg`.

```
{
"language": [
   {  "name":        "Java",
      "filePattern":    "*.java",
      "projectsPath":   "../AuToCa/
    resources/testprojects/Thesis/",
      "tokenPath":      "../AuToCa/
    resources/java_keywords.txt",
      "projectSizeLimit":  "6000000000",
      "minAmountOfProjects": "1"}
],

"database": {
   "FILENAME":    ".thesis_apache_autoca",

   "DRIVER":      "org.h2.Driver",
   "USER":        "sa",
   "PASSWORD":    "",
   "LOGINPREFIX": "jdbc:h2:",

   "outputLocation":       "resources/
    databases/",

   "TEMPORARY":   "temporary",
   "TEMPFILTER":  "temporary_filter",
   "OCCURENCE":   "occurences",
   "TOKEN":    "tokens",
   "FILE":        "files",
   "PROJECT":     "projects",
   "LANGUAGE":    "languages",
   "RESULTTABLE": "resulttable",
   "RANK":        "rank_",
   "PREFIXSTAT":  "LStat_",
   "PRECISION":   "precision_",
   "SUMMARY":     "summary_"
},

"filterchains": [
   {  "resultName": "_Newl_Int_Up_Sub",
      "languages": [
         {"name":    "Java"}
      ],
      "filters": [
         {"name": "Output",
```

```
41              "save": "true"},
42             {"name": "SubStringFilter",
43              "subString": "#"},
44             {"name": "UpCaseFilter"},
45             {"name": "NewlineFilter"}
46          ]
47       },
48       {  "resultName": "_Glo_Int_Up_Sub",
49          "languages": [
50             {"name":    "Java"}
51          ],
52          "filters": [
53             {"name": "Output",
54              "save": "true"},
55              {"name": "SubStringFilter",
56              "subString": "#"},
57             {"name": "UpCaseFilter"},
58             {"name": "GlobalFilter"}
59          ]
60       },
61       {  "resultName": "_Cov_Int_Up_Sub",
62          "languages": [
63             {"name":    "Java"}
64          ],
65          "filters": [
66             {"name": "Output",
67              "save": "true"},
68              {"name": "SubStringFilter",
69              "subString": "#"},
70             {"name": "UpCaseFilter"},
71             {"name": "CoverageFilter"}
72          ]
73       },
74       {  "resultName": "_RealInd_Int_Up_Sub",

75          "languages": [
76             {"name":    "Java"}
77          ],
78          "filters": [
79             {"name": "Output",
80              "save": "true"},
81              {"name": "SubStringFilter",
82              "subString": "#"},
83             {"name": "UpCaseFilter"},
84             {"name": "RealIndentFilter"}
85          ]
86       }
87 ],
88
```

```
"tokenizer": {
   "DEFAULT_LS":            "\n",
   "DEFAULT_WORD":          "[a-zA-Z]\\w+",

   "DEFAULT_STRING":           "(?s)
   \".*?\"",
   "DEFAULT_MULTI_COMMENT":    "(?s)
   /\\*.*?\\*/",
   "DEFAULT_SINGLE_COMMENT":   "(?s)//.*?\
   n",

   "PYTHON_LIKE_COMMENT":      "(?s)#.*?\n
   ",

   "WHITESPACE":            "[ \t]+",
   "START_OF_LINE":         "(?m)^[ \t]*",
   "NEWLINE":               "\n",
   "TABSPACE":              "    ",
   "EMPTYLINE":             "(?m)^[ \t]*\n"

},

"tokenhandler": {
   "DEFAULT_MAX_TOKEN_LENGTH":   "100",
   "DEFAULT_MIN_TOKEN_LENGTH":   "1",

   "DBNEWLINE":"#newline",
   "DEDENT":   "#dedent",
   "INDENT":   "#indent",
   "STRING":   "#string",
   "COMMENT":  "#comment",
   "DELIMITER":"#delimiter",
   "LONGWORD": "#longword"
}
}
```

Listing 12: Configuration file example

7.4 The h2 database

1. To access the results you need to download the H2 package. To run it you can extract it anywhere you like.
2. Now start the H2 database by executing the h2 script `h2.bat` in the bin directory.
3. Open a browser and go to http://localhost:8082.
4. The autoca database is called as configured in the configuration file under *DB/-FILENAME*.

   Modify the JDBC URL field to point to your autoca database, *e.g.*,

   Omit the .h2.db ending. You can leave everything else on its default values.
5. With the precision table you can now plot the diagrams using your favourite plotter(*e.g.*, pgfplot, R, Excel) as seen in the thesis.