2018 25th Asia-Pacific Software Engineering Conference (APSEC)

Multilingual Detection of Code Clones Using ANTLR Grammar Definitions

Yuichi Semura*, Norihiro Yoshida†, Eunjong Choi‡ and Katsuro Inoue*
*Osaka University, Japan, {y-semura, inoue}@ist.osaka-u.ac.jp
†Nagoya University, Japan, yoshida@ertl.jp
‡Nara Institute of Science and Technology, Japan, choi@is.naist.jp

***Abstract*—So far, many tools have been developed for the detection of code clones in source code. The existing clone detection tools support only a limited number of programming languages and do not provide any easy extension mechanism to handle additional language. However, from our experience in industry/university collaboration, we found that many practitioners need to analyze source code written in various languages. In this paper, we propose an approach for the multilingual detection of code clones using grammar files for a parser generator ANTLR. We extended a clone detection tool CCFinderSW with the proposed approach and then apply the extended CCFinderSW to ANTLR grammar files for 43 languages. As a result, the files for 39 out of the 43 languages can be analyzed correctly by the extended CCFinderSW.**

***Index Terms*—code clone, lexical analysis, parser generator, ANTLR**

I. Introduction

Programmers often copy and paste code so that they can reuse existing code fragments. This causes code clones (i.e, code fragments that are identical or similar code fragments to each other). Generally, a code clone is regarded as one of the factors that hinder software maintainability [1], [2], [3]. For instance, when a cloned code fragment contains a bug, a programmer should check all of its cloned fragments for the same bug. If the cloned fragments contain the same bugs, they should be modified for the same bug. To that end, he/she should know locations of all code clones in the source code. However, it is difficult for developers to recognize all code clone from large-scale software systems. To alleviate this problem, a multitude of code clone detection tools have been developed [4], [5], [6]. For example, Kamiya has developed a token-based code detection tool CCFinderX [4] that is widely used in academic research as well as industries [7], [8].

The existing clone detection tools support only a limited number of programming languages and do not provide any easy extension mechanism to handle additional language [4], [5], [6]. From our experience in industry/university collaboration, we found that many practitioners need to analyze source code written in various languages [9]. Such an extension mechanism to handle additional language saves effort and time for practitioners and tool developers [10].

A clone detection tool CCFinderSW has an extension mechanism to handle additional language on demand from practitioners. It enables practitioners to easily change the lexical mechanism for comment elimination and identifier replacement. This mechanism saves practitioners from troubles during the implementation of lexical analyzers. When users apply CCFinderSW to an additional language, they should prepare two files to define comment rules and reserved words of the language. However, in order to prepare a description file of comment rule definitions, it is necessary for users to learn grammars of the description file. Moreover, it is troublesome to create a description all reserved words of a target language. An extension mechanism to save these troubles is required CCFinderSW.

A parser generator is a programming tool that creates a lexer, parser or compiler based on the grammar definitions of the target language. Since such grammar definition file of a programming language has lexical information in the source code, by choosing what is necessary for code clone detection from among them and applying it to the tool, it is possible to create any easy extension mechanism to handle additional language. Regarding ANTLR, there is a Github repository 'grammars-v4'[1] which contains over 150 grammar definition files. Therefore, practitioners of a code clone detection tool can easily get a grammar definition file from the repository and give it to the tool.

In this paper, we propose an approach to automatically extract lexical information necessary for token-based code clone detection from grammar definitions of the parser generator. Besides, we extended a clone detection tool CCFinderSW that has a lexical information extractor from grammar definitions of a parser generator ANTLR.

II. Clone Detectors: CCFinderX and CCFinderSW

To introduce the mechanism of a token-based clone detection tool, we briefly explain CCFinderX and CCFinderSW. They identify not only Type-1 clones [11] (i.e. identical code fragments except for variations in whitespace, layout and comments) but also Type-2 clones [11] (i.e. syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.) by replacing identifiers related to types, variables, and constants with a special token.

Kamiya has developed a token-based code detection tool CCFinderX [4] that is widely used in academic research as well as industries. Practitioners can replace its lexical analyzer with another one depending on a target language.

[1] https://github.com/antlr/grammars-v4

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00088

CPS Conference Publishing Services

In other words, when we apply CCFinderX to additional languages, they should implement lexical analyzers. However, this implementation requires users' plenty of knowledge and takes more time and effort.

We developed CCFinderSW [9], a code clone detection tool based on CCFinderX. It contains a mechanism to tokenize source code written in many languages and to detect code clones and requires comment rules and reserved words as the input. This mechanism saves the time and effort of implementation of lexical analyzers of new languages. The code clone detection process of CCFinderSW is comprised of four steps, which are lexical analysis, transformation, detection, and formatting. Additionally, lexical analysis is comprised three detailed steps, which are comment elimination, tokenization, and identifier distinction.

The following subsections describe the detail of lexical analysis and transformation, and problems of CCFinderSW. With respect to the detection and formatting, please refer to [9].

A. Comment Elimination

The lexical analysis of CCFinderSW eliminates comments in the source code according to the type-1 and type-2 code clone definition. This tool requires comment rules and reserved words as the input. The user creates a description file of comment rule definitions and gives it to CCFinderSW at runtime. There are five kinds of configurable comment rules, that are line comment, multi-line comment, full line comment, full multi-line comment and string literals.

B. Tokenization

After the comments are eliminated from the source code based on the defined comment styles, each line of the source code is divided into tokens based on a lexical rule. The following four lexical rules are used for the tokenization. Note that a rule with a low number has a higher priority.

1) Each character literal or string literal corresponds to one token, respectively.
2) White spaces and line breaks are delimiters.
3) Each symbol is one token.
4) Other consecutive alphabetic or numeric strings are identified as one token.

C. Identifier Distinction and Transformation

In the identifier distinction process, the token generated by the tokenization process is distinguished between an identifier or a reserved word as preprocessing of transformation. The user creates a description file of reserved words and gives it to CCFinderSW at runtime.

In the transformation process, the token sequence is transformed in order to detect meaningful code clones. In detail, all the identifiers representing variable names and function names are replaced by the same token. Reserved words are character strings that are reserved by the programming language and cannot be used for variable and function names.

TABLE I
MAIN TOKENS USED IN ANTLR

Token	Description
'literal'	Match that character or sequence of characters.
[char set]	Match one of the characters specified in the character set.
.	The dot is a single character wildcard that matches any single character
~x	Match any single character not in the set described by x. In this paper, we call this token a NOT operator.
x*	Match zero or more occurrences of x.
x?	Match zero or one occurrences of x.
x*?	Match the shortest occurrences of x.
x|y	Match either x or y.

D. Problems of CCFinderSW

When users apply CCFinderSW to an additional language, they should prepare two files to define comment rules and reserved words of the language. However, in order to prepare a description file of comment rule definitions, it is necessary for users to learn grammars of the description file. Moreover, it is troublesome to create a description all reserved words of a target language. An extension mechanism to save these troubles is required CCFinderSW.

III. A PARSER GENERATOR: ANTLR

This section describes grammar definitions in ANTLR. The following example is a grammar definition file which represents a grammar of arithmetic operation. In line 1, **grammar Prog** defines the name of this grammar as **Prog**. From line 2, each line represents a rule that constitutes this grammar tree. There are two kinds of grammar rules, namely lexer rule and parser rule. The lexer rules define tokens that appear in source code. The parser rules define syntax grammars. In this example, *INT* and *WS* are lexer rules. *prog*, *expr*, *term*, and *factor* are parser rules.

```
1 grammar Prog;
2 prog: expr;
3 expr: term (('+'|'-') term)*;
4 term: factor (('*'|'/') factor)*;
5 factor: INT | '(' expr ')';
6 INT: [0-9]+;
7 WS: [ \t\r\n]+ -> skip;
```

Listing 1. An example of an ANTLR grammar file

Parr publishes documents of the lexical representation used in ANTLR [12]. Table I shows parts of the documents that are important to understand Section IV.

IV. IMPLEMENTATION OF A LEXICAL INFORMATION EXTRACTOR FROM GRAMMAR DEFINITION

In this section, we propose an approach to automatically extract lexical information necessary for token-based code clone detection from grammar definitions of the parser generator. Besides, we extended a clone detection tool CCFinderSW that had a lexical information extractor from grammar definitions of a parser generator ANTLR. We used Java for this development.

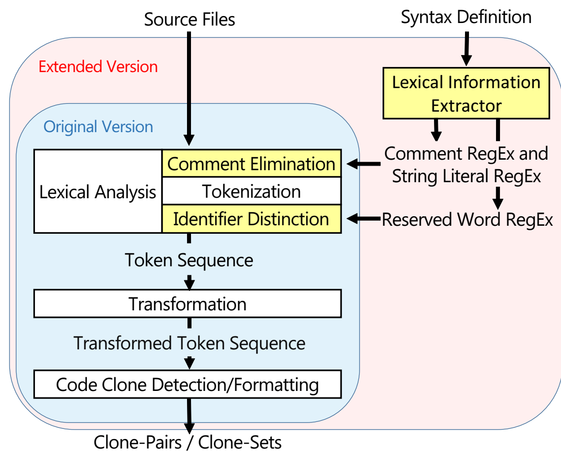

Fig. 1. An overview of the original and the extended versions of CCFinderSW

CCFinderSW detects code clones using lexical information extracted from grammar definitions. This section describes the implementation method of a new module, which is a lexical information extractor, and some of the changes in the existing processes.

Figure 1 depicts an overview of extended CCFinderSW. Areas highlighted in light yellow represents modules that we newly develop or extend.

A new module extracts lexical information from grammar definitions of ANTLR and outputs three Regular Expressions (*RegExes*) which represent comments, string literals, and reserved words. A *RegEx* of comments and a *RegEx* of string literals are used in the comment elimination process, and a *RegEx* of reserved words is used in the identifier distinction process. Therefore, we needed to extend the two process and enable them to accept *RegEx*es as inputs.

Because of the online repository 'grammars-v4', practitioners can easily obtain a grammar definition and give it to CCFinderSW. From an overall point of view, practitioners should give CCFinderSW source files and grammar definition of a target language. Also, they can choose between the existing and the new process flow.

We explain an overview of the processes in a lexical information extractor. A lexical information extractor contains a parser for grammar definitions of ANTLR that generates syntax trees.

First, it analyzes the tree of grammar definitions and extracts lexical information from the tree. Second, it transforms lexical definitions into *RegEx*es of comment rules, string literals, and reserved words. Finally, it gives the three *RegEx*es to the comment elimination process and the identifier distinction process.

The reason for transforming lexical definitions into a *RegEx* is as follows: A *RegEx* resembles a grammar definition in ANTLR, and is often used for search and replacement of character strings [13]. Therefore, comment elimination with a *RegEx* is suitable for this study. In addition to this, comment elimination with a *RegEx* allow CCFinderSW to handle more comment rules than the five existing comment rules.

Section IV-A describes the transformation of comment definitions into a *RegEx*, Section IV-B describes the transformation of string literal definitions, and Section IV-C describes the transformation of reserved words definitions.

A. Transformation of comment rule definitions into a RegEx

This section describes the implementation to extract comment rule and to transform them into a *RegEx*.

The process of transformation of comment rule into a *RegEx* is comprised of the following four steps:

Step A: Choose grammar definitions of comment rules from all rules.

Step B: Apply other grammar definitions to references in chosen definitions.

Step C: Transform applied definitions into *RegEx*es in available in Java.

Step D: Combine all transformed *RegEx*es into one *RegEx*.

Details of each step are as follows.

Step A First, the lexical information extractor chooses grammar definitions of comment rules. For this process, we set 4 standards so as to identify grammar definitions of comment rules based on our investigation. When a definition applies at least one of the standards, it is identified as a comment rule by this extractor. The standards are listed below.

1) A name of a definition contains 'comment', 'COMMENT' and so on.
2) A definition is linked to a 'skip' command.
3) A definition is linked to a 'channel(HIDDEN)' command.
4) A definiton is linked to a 'channel(X)' command. Moreover, X contains 'comment', 'COMMENT', and so on.

The following listing shows four grammar definitions of comment rules which are applied to standards. Also, they correspond to a multi-line comment written in C/C++.

```
1 Comment:'/*'.*?'*/';
2 Block1:'/*'.*?'*/'->skip;
3 Block2:'/*'.*?'*/'->channel(HIDDEN);
4 Block3:'/*'.*?'*/'->channel(BComment);
```

Listing 2. Notations for defining comments

Step B Second, the extractor applies recursively other grammar definitions to references in chosen definitions. In an example of a grammar definition file in Section III, the 'term' definition refers to the 'factor' definition. In such cases, the extractor applies a definition a reference destination to a reference source.

Step C Third, the extractor transforms applied definitions to *RegExes* available in Java. As a reason for that, there is some difference between descriptions of *RegExes* in Java and descriptions of grammar definitions in ANTLR. There are 3 patterns of descriptions in ANTLR which the extractor transforms into *RegExes*.

The first one is a single quotation. In grammar definitions in ANTLR, a literal appearing in the source code are enclosed

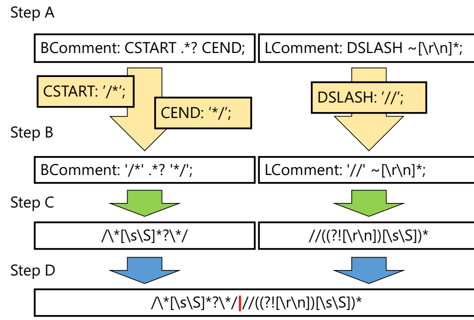

Fig. 2. An example of the transformation of comment rule definitions into a *RegEx*

in single quotations. In a *RegEx*, this single quotations is unnecessary and the extractor eliminates them.

The second one is a NOT operator. In grammar definitions in ANTLR, a '~x' matches any single character not in the set described by x, that is, '~' has a function of a NOT operator. In *RegExes*, there is no expression corresponding to a NOT operator. Accordingly, the extractor transforms a '~x' into a *RegEx* with the negative lookahead.

The third one is a dot. In grammar definitions in ANTLR, a dot is a single character wildcard that matches any single character. However, in *RegEx* used in Java, a dot matches any single character except the newlines. Because of the difference between two definitions of a dot, the extractor transforms a '.' into a '[\s\S]'.

Step D Finally, the extractor combines all transformed *RegExes* into one *RegEx*. That is, it outputs a logical sum of all generated *RegExes* with '|'. At the end of Step D, the transformation of comment grammar definitions into *RegExes* is completed.

Figure 2 is an example of the transformation of comment rule definitions into a *RegEx*. In Figure 2, the extractor chooses two definitions, which is labeled 'BCOMMENT' and 'LCOMMENT', and transforms them into a *RegEx*,

B. Transformation of string literal definitions into a RegEx

This section describes an approach for extract string literal and to transform them into a *RegEx*.

A string literal is a type of literal in programming for the representation of a string object within the source code, respectively. For instance, in Java program, characters enclosed in double quotes are identified as a string literal. CCFinderSW defines a string literal as one of the configurable comment rules. Also in this expansion, to implement similar the comment elimination process, it is necessary to give string literal grammar to CCFinderSW. Therefore, the lexical information extractor extracts string literal definitions and transforms them into a *RegEx*. Finally, it gives the *RegEx* to the comment elimination process of CCFinderSW.

In order to implement a lexical information extractor, we investigated notations of grammar definitions corresponding to string literals. The following listing shows notations of string literals which frequently appear in our investigation.

```
1 StringLiteral: QUOTE StringCharacters? QUOTE;
2 STRING : 'string';
```

Listing 3. notation for defining string literals

The grammar definition in line 1 represents a string literal grammar. The grammar definition in line 2 represents a reserved word 'string' which appears in the source code, although its name contains 'STRING'. Considering the notations of string literal definitions, we set a standard for extraction of string literal definitions. The standard defines a string literal definition which is not a reserved word definition and whose name contains 'string', 'STRING' and so on.

Then, the extractor identifies string literal definitions according to the standard and transforms them into a *RegEx*. The transformation of string literals is the same as comment rules, therefore detailed explanation is omitted.

C. Transformation of reserved word definitions into a RegEx

This section describes an approach to extract reserved words and to transform them into a *RegEx*. In the same as the others, we investigated notations of grammar definitions corresponding to reserved words. The following listing shows two different notations for defining a reserved word 'while'. Both of them are frequently appear in our manual investigation.

```
1 WHILE:'while';
2 WHILE:[wW][hH][iI][lL][eE];
```

Listing 4. Two different notations for defining a reserved word 'while'

In line 1 of this example, the definition which is named 'WHILE' links a string literal 'while'. This notation is widely used in grammar definitions which we investigate. In line 2 of this example, the definition is composed of character sets such as '[wW]'. This character set '[wW]' matches both uppercase and lowercase letter of 'w'. Hence, a *RegEx* '[wW][hH][iI][lL][eE]' matches 'while' and 'WHILE' and 'WhIlE', and so on. According to two notation of grammar definitions, we implement the extraction process of reserved words. A character set used in ANTLR is also used in a *RegEx*. Consequently, it transforms reserved word definitions into a *RegEx*.

Before our investigation, we defined that a reserved word contains only alphabets. However, since we found some reserved words which contain symbols such as '_' and '@', we add an option which designates characters composing reserved words.

After the extraction process, it transforms definitions of reserved words into a *RegEx*. The transformation of reserved words is the same as the others, therefore detailed explanation is omitted.

V. Evaluation

As an evaluation, we applied the proposed approach to a collection of ANTLR grammar files and confirmed the accuracy of the proposed approach. After that, we extended CCFinderSW with the proposed approach. And then, we

applied it to a collection of source files written in several different languages and checked the detected code clones.

A. Result of analyzing grammar files

We investigated how many grammar files in an ANTLR grammar repository 'grammars-v4'[2] are correctly analyzed. This repository contains over 150 grammar definition files.

For the evaluation, we selected 43 of 154 languages which are available in the advanced search[3] of the code search engine at GitHub because impractical languages such as esoteric languages (e.g., *Brainf*ck*) should be excluded from the eligible languages. And then, we manually identified the notations of comments, string literals, and reserved words from each of the grammar definition files of the 43 eligible languages. After that, we applied the proposed approach to the 43 grammar definition files. Finally, we checked whether the extracted *RegExes* are correct as comments, string literals and reserved words of each language based on the result of our manual identification.

The result shows that the proposed approach successfully extracted notations for comments, string literals, and reserved words in 38, 36 and 37 out of the 43 eligible languages respectively. As the total, all three *RegExes* are correctly analyzed in 39 out of the 43 eligible langueges.

The following listing shows the comment definitions in a grammar file 'Lua.g4'. *COMMENT* defines a comment notation of Lua and refers to *NESTED_STR*. *NESTED_STR* refers to itself recursively. Therefore, comments used in Lua is unable to be expressed by *RegEx*. Our method proposed in this paper is unable to extract comment notations from such grammar definitions.

```
1  COMMENT
2      : '--[' NESTED_STR ']' -> channel(HIDDEN)
3      ;
4  fragment
5  NESTED_STR
6      : '=' NESTED_STR '='
7      | '[' .*? ']'
8      ;
```

Listing 5. The comment definitions in the grammar file 'Lua.g4'

Please note that several of the eligible languages do not have any keyword/reserved word. The detailed result is available online[4].

B. Result of the detection of code clones

We used the extended CCFinderSW with the proposed approach to a collection of source files written in several different languages and checked whether the detection result is correct.

For this evaluation, we selected c, cobol85, cpp14, ecmasript, java9, python3 and visualbasic6 from the 39 correctly-analyzed languages in Section V-A. Nowadays many of systems which written in each of the seven selected languages have much legacy code.

[2]We collected ANTLR grammar files from 'grammars-v4' repository on December 14, 2017.

[3]https://github.com/search/advanced

[4]https://sites.google.com/site/yoshidaatnu/APSEC2018ERAResult.pdf

We downloaded source files which are written in the selected languages from Github repositories and then applied the extended CCFinderSW to the source files. After that, we randomly selected 20 pairs of code clones from the detection result for each selected language and then manually confirmed the all of the selected pairs for each language are correctly pairs of code clones.

VI. Summary

In this paper, we propose an approach to automatically extract lexical information necessary for token-based code clone detection from grammar definitions of the parser generator. Besides, we extend a clone detection tool CCFinderSW that has a lexical information extractor from grammar definitions of a parser generator ANTLR. In an evaluation, we indicate that the lexical information extractor in CCFinderSW can extract most rules of comment, string literals and reserved words in 43 languages. Also, we manually confirmed the all of the selected pairs for each of seven languages are correctly pairs of code clones.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP25220003, JP18H04094 and JP16K16034.

References

[1] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proc. of ICSM*, 1997, pp. 314–321.

[2] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[3] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. Morgan Kaufmann Publishers Inc., 2009.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. of ICSE*, 2007, pp. 96–105.

[6] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proc. of ICSE*, 2016, pp. 1157–1168.

[7] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, "Experience of finding inconsistently-changed bugs in code clones of mobile software," in *Proc. of IWSC*, 2012, pp. 94–95.

[8] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Applying clone change notification system into an industrial development process," in *Proc. of ICPC*, 2013, pp. 199–206.

[9] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "CCFinderSW: Clone detection tool with flexible multilingual tokenization," in *Proc. of APSEC 2017*, 2017, pp. 654–659.

[10] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa, "OCCF: A framework for developing test coverage measurement tools supporting multiple programming languages," in *Proc. of ICST 2013*, 2013, pp. 422–430.

[11] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[12] T. Parr, "antlr4/index.md at master antlr/antlr4," https://github.com/antlr/antlr4/blob/master/doc/index.md.

[13] J. E. Friedl, *Mastering regular expressions*. O'Reilly Media, 2002.