Finding Code-Clone Snippets in Large Source-Code Collection by `ccgrep`

Katsuro Inoue[1], Yuya Miyamoto[1], Daniel M. German[2], and Takashi Ishio[3]

[1] Osaka University, Osaka, Japan
{inoue,yuy-mymt}@ist.osaka-u.ac.jp
[2] University of Victoria, Victoria, Canada
dmg@uvic.ca
[3] Nara Institute of Science and Technology, Ikoma-shi, Japan
ishio@is.naist.jp

Abstract. Finding the same or similar code snippets in the source code for a query code snippet is one of the fundamental activities in software maintenance. Code clone detectors detect the same or similar code snippets, but they report all of the code clone pairs in the target, which are generally excessive to the users. In this paper, we propose `ccgrep`, a token-based pattern matching tool with the notion of code clone pairs. The user simply inputs a code snippet as a query and specifies the target source code, and gets the matched code snippets as the result. The query and the result snippets form clone pairs. The use of special tokens (named meta-tokens) in the query allows the user to have precise control over the matching. It works for the source code in C, C++, Java, and Python on Windows or Unix with practical scalability and performance. The evaluation results show that `ccgrep` is effective in finding intended code snippets in large Open Source Software.

Keywords: Code Snippet Search · Pattern Matching · Clone Types

1 Introduction

Finding and locating the same or similar code snippets in source code files is a fundamental activity in software development and maintenance, and various kinds of software engineering tools or IDEs have been proposed and implemented[19].

A (code) clone is a code snippet that has an identical or similar snippet, and a pair of such snippets is called a (code) clone pair[6]. A large body of scientific literature on clone detection has been published and various kinds of code clone detection tools (detectors) have been developed[18, 20]. These code clone detectors are candidates for finding similar code snippets, but most of those are designed to detect all of the code clone pairs in the target, which are generally excessive to the user who wants to search for a specific query snippet.

It has been reported that `grep`[8], a character-based pattern matching tool, is widely used in the software engineering practice to find lines with a specific keyword[14, 21], although making a query for a code snippet that spans multiple lines needs some skill and effort.

In this paper we propose a tool, named `ccgrep` (*code clone grep*), to find code snippets by using the notion of clone detection and pattern matching. Search queries can be simply code snippets, or code snippets enhanced with meta-tokens having a leading $ that can provide flexibility to narrow or broaden the search query. `ccgrep` is not an ordinary code clone detector that finds all code clone pairs in the target program and is a code snippet finder that reports code snippets composing code clone pairs against the query snippet.

`ccgrep` works on Windows or Unix as a simple but reliable clone detector and pattern matching tool for C, C++, Java, and Python. `ccgrep` has been applied to various applications, and it showed high scalability and performance for large source-code collection. `ccgrep` is an Open Source Software system and can be obtained from GitHub[4].

2 Motivating Example

Some uses of the ternary operator (e.g., `exp1 ? exp2 : exp3` meaning the result of this entire expression is `exp2` if `exp1` is true, otherwise the result is `exp3`—available in C, C++ and Java) are considered bad practice[23]. For example, the use of `a < b ? a : b` is arguably harder to read than using `min(a,b)`. Therefore, it might be desirable to replace the ternary operator with a function or macro that returns the minimum value. The following is an example found in the file `drivers/usb/misc/adutux.c` in the Linux kernel (v5.2.0).

```
amount = bytes_to_read < data_in_secondary ?
                                bytes_to_read : data_in_secondary;
```

This line of code should be replaced with a more readable expression (note that the macro `min` in Linux guarantees no side effects):

```
amount = min(bytes_to_read, data_in_secondary);
```

We might consider that finding all occurrences of such usage of the ternary operator could be done by clone detectors. A popular clone detector NiCad[7] reports 646 block-level clone classes for the `drivers/usb` files by the default setting, but no snippet with the ternary operator case is included in the result because it is too small to be detectable.

Alternatively, we would try it with `grep` but it is not easy. For example, simply executing "`grep '<'`" for all 598 files (total 51,6394 lines in C) under /drivers/usb produces 16335 matching, including many undesired patterns such as "`if (a<b)`", "`for (i=0; i<x; ...)`", or "`#include <linux/...>`". We could narrow the matches by concatenating `grep` like,

[4] https://github.com/yuy-m/CCGrep

```
grep '<' -r . | grep '?' | grep ':'
```

However, it still produces 149 matches. Perhaps more problematic is that the expressions could span multiple lines. While it is possible to create a complex regular expression to find these expressions, it would be time-consuming and potentially error-prone.

Ideally, we would like to be able to specify a simple and easy-to-create-and-understand query to find these types of snippets. Therefore in this paper, we propose `ccgrep` and its query is written simply as:

```
a < b ? a : b
```

In a nutshell, this query specifies that a variable (represented by a) should be followed by `<` and then the second variable (represented by b), followed by a `?`, followed by the same first variable found, followed by `:`, followed by the second variable. Also, white spaces and comments should be ignored. This query would match all type 2 clones (mentioned in Sec.3.3) with consistent variable names such as `x<y?x:y` but it would not match `x<y?x:z`.

As a practical application, we have used this query to identify 3 instances of such an expression in Linux's `drivers/usb` and submitted patches to replace them with `min`. Two of those patches have been accepted already into Linux.

3 Overview of Code Clone Query by ccgrep

3.1 Basic Features

The input of `ccgrep` is the query and the target of the source code files in the same programming language. The output is a list of the matched code snippets in the target. The query and the matched code snippets form clone pairs. The query is a code snippet of single or multiple lines and is composed of the regular tokens in the language and the extended tokes with meta symbols having a leading \$. We will describe these based on the classification of the clone types. Formalization of the matching is presented in Appendix and also in [11].

3.2 Query for Type 1 Clone

A Type 1 code clone pair is two code snippets possibly with different spacing, line break, or comment. To find type 1 cloned snippets, a code snippet in the programming language is directly given as the query, with a leading \$ for each identifier or literal. Note that in the following examples, we will use Java as the programming language.

Query: `int $a= $0;`

Target: `int a=0 /* some comments */;` *Match*

Target: `int b=0 ;` *Not Match*

In this case, the query matches a code snippet with a comment, but it does not match the latter case of identifier `b`. The users do not worry about the white spaces and comments in the query.

3.3 Query for Type 2 Clone

A Type 2 code clone pair is two code snippets with the difference of the replacement of identifiers and literals, in addition to the difference of type 1 clones.

In type 2 matching, a user-defined identifier in the query matches any identifier in the target. The same also applies to literal. This "normalization" of the user-defined names allows very flexible pattern matching to find different identifiers or literals. By default, `ccgrep` executes so-called *Parameterized match*[3] or *P-match* for short, such that if two identifiers (or literals) in the query are the same, then the corresponding tokens in the target must be consistently the same. These normalization and p-match are formally explained in Appendix.

Query: `a = 0; a = a + b;`
Target: `y = 0; y = y + c;` *Match*

Target: `y = 0; y = z + c;` *Not Match*

In the former case, `a` consistently corresponds to `y`, but in the latter case, it does not[5].

3.4 Query for Type 3 Clone

A Type 3 code clone pair is two code snippets with a difference of some statements of addition, deletion, or change, in addition to the distinction of type 2. We employ wild-card tokens in the query, which extend the matching from the original seed tokens. The seed snippet and the matched snippet form a code clone pair of type 3. We can replace a token in the seed snippet with '$.' that matches any single token.

Seed: `a = 5 ;`
Query: `a = $. ;`
Target: `a = b ;` *Match*

'$$' is a wild-card token to match zero or more tokens before the next token matches.

Seed: `a = 10 ;`
Query: `a = $$ ;`
Target: `a = b+c+10 ;` *Match*
Target: `a = f(g,h) ;` *Match*

The following is a more complex example.

Seed: `a= f(q); if(a<0){a=-a;}`
Query: `a= $f(p); $$ if(a<0){a=-a;}`
Target: `b= f(q); if(b<0){b=-b;}` *Match*
Target: `b= f(q); c= c+10; d=20; if(b<0){b=-b;}` *Match*

[5] This can be changed by an option to allow inconsistent matching.

3.5 Finding Various Code Snippets

Combining the regular tokens and meta-tokens in the query, we can find many different kinds of code patterns in the target, from simple to complex ones.

Method *XYZ* with no parameter
Query: `$XYZ( )`

Method *XYZ* with 0 or more parameters
Query: `$XYZ($$)`

Method *print* with variable *buf* as the 1st parameter
Query: `$print($buf, $$)`

Any method definition
Query: `T f($$){$$}`

Note that type names are treated as identifiers and then `T` matches any type name.

Getter method
Query: `T f(){return this.v;}`

Setter method
Query: `T1 f(T2 v1){this.v1=v2;}`

***if* statement**
Query: `if ($$){$$}`

***for* statement using control variable**
Query: `for(T i=0; i<$$; i++){$$}`

In addition to finding these patterns, one of the usable use-cases would be a copy-and-paste code search. A developer finds a bug in a system and locates the snippet that causes the defects. She would want to find the same or similar occurrences of the bug in the system, then she copies the buggy snippet and runs `ccgrep` with the pasted snippet as the query. Then she instantly gets type 2 clone snippets. She does not need to set up a heavy clone detector, nor does she need to do tedious analysis of the unnecessary detection results.

4 Architecture of ccgrep

The architecture of `ccgrep` is presented in Figure 1.

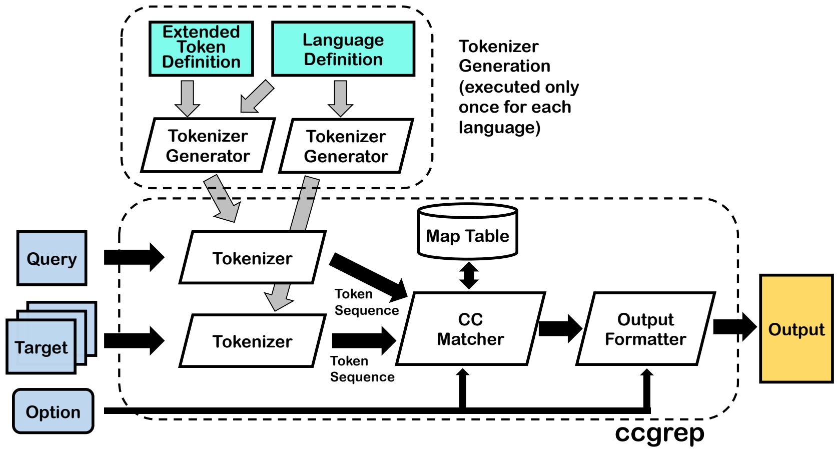

Fig. 1. Architecture of ccgrep

Tokenizer Generators: Parser generator ANTLR is used to generate two kinds of tokenizers. For the target tokenization, only the language definition is used to recognize the regular tokens, but for the query tokenization, the definition of the meta-tokens and that of regular tokens are used. This process has been executed only once for each target language.

Tokenizers: Each tokenizer removes white spaces and comments from the input files and decomposes the code into tokens. The query tokenizer accepts the meta-tokens starting with $ and the regular tokens defined by the language, but the target tokenizer accepts only the regular tokens. The tokenizer for the target files is executed in parallel for each file, along with the following CC Matcher.

CC Matcher: This performs a naive sequential pattern matching algorithm between two token sequences for the query (of the length m) and the target (of the length n), whose worst-case time complexity is $O(mn)$[9]. For type 2 code clone matching, we record the position for each identifier and literal in Map Table to check proper P-matching. The table contents are flushed for each query. Option controls the normalization level, input language, output form, and many others.

Output Formatter: This process constructs the output for the successful matching result. Based on the input option, we can view the match result, like `grep`, in the form of the file name associated with the matched top line as the default, or as many other styles such as full matched lines, only the number of lines, or so on.

`ccgrep` is written in Java associated with the ANTLR output, and it is very easily installed and executed on various Unix or Windows environments with a single JAR file (about 1M byte) containing all necessary libraries.

5 Evaluation

The goal of the evaluation is to show that our proposed approach can find various kinds of intended code snippets effectively and efficiently. This goal could be decomposed into the following three research questions.

RQ1 : **Query Expressiveness** Are queries for various types of code clones expressible by `ccgrep`?
RQ2 : **Accuracy of ccgrep** Does `ccgrep` accurately find various types of code clones already detected by other approaches?
RQ3 : **Performance of ccgrep** What is the execution time of `ccgrep`? Is the token-based naive sequential pattern matching approach fast enough in practice?

5.1 RQ1: Query Expressiveness

As shown in previous sections, it is obvious that our approach can easily create various query patterns for type 1 matching, type 2 matching with P-match, and type 2 matching with non-P-match, by specifying a code snippet associated with appropriate options. In addition, we can specify the name of an identifier or literal, if we place $ before the name.

A type 3 code-clone snippet is one with a few statement addition, or deletion, or change for a seed snippet. Thus the query for type 3 matching could be made from the seed by adding meta-tokens such as $., $$, or $*, deleting some regular tokens in the seed, or modifying some regular tokens with $., $$, or other meta-tokens.

Therefore, the queries for type 1 to 3 code clones can be effectively created from a code snippet at hand.

5.2 RQ2: Accuracy of ccgrep

For evaluation of query-matching (or information retrieval) systems, recall and precision values, computed by comparing the matched results with the oracles for the queries, are popularly employed[2]. Here in our approach, however, the query to CC matching has no ambiguity and it reports the matching result rigorously as expected and specified by the query with options. In such a sense, the result is always the same as the oracle, i.e., the recall and precision are always 1. Thus, instead of using recall and precision, here we simply investigate if `ccgrep` works accurately in the sense that code clones already reported by other approaches could be found by our approach.

For this purpose, first, we have employed BigCloneBench[24] which is a huge collection of various kinds of code clones. We have extracted all pairs classified as type 1 and type 2 code clones from BigCloneBench, and for each clone pair $(sp1, sp2)$, we have checked if $sp2$ is successfully found in the result of `ccgrep` for $sp1$ as a query with appropriate options, and vice versa. Table 1 shows the numbers of type 1 and 2 clones found by `ccgrep`.

Table 1. Checked Clones in BigCloneBench

Clone Type	Clone Pairs	Found	Not Found
Type 1	48116	48111	5
Type 2	4234	4232	2
Total	52350	52343	7

As we can see in Table 1, most type 1 and 2 clones are found accurately. There were several cases of not-found clones, and we have investigated further those cases and recognized that those cases are faults of the classification of BigCloneBench, some of which should be classified into type 3, and some others are not clones. Thus, we can say that all of the proper type 1 and 2 clones in BigCloneBench were perfectly found by `ccgrep`.

For type 3 clones, since BigCloneBench contains huge type 3 data and we cannot make the queries for those, we have instead used CBCD data[16], that contains 11 type-3 clone sets taken from the source code of Git, the Linux kernel, and PostgreSQL. We have crafted type 3 queries from one of the code snippets in each clone set as the seed and have checked if those queries accurately match the other snippets in the same clone set. We have confirmed that all the crafted queries accurately match other snippets in each clone set.

As far as our investigation, all the matches are controlled by the query and are performed accurately as we have expected.

5.3 RQ3: Performance of ccgrep

It is interesting to know that our approach, i.e., token-based and naive sequential pattern matching, can be implemented fast enough for practical use. We have examined various queries for `ccgrep` with the target source files of Antlr and Ant in Java, and CBCD data (Git, PostgreSQL, and Linux Kernel) in C, and have measured the performance of `ccgrep`. Following are employed queries. All execution was made with the default setting of `ccgrep` except for the language option.

qA:
```
a < b? a: b
```
Find ternary operation to give a smaller value.

qB:
```
T1 f(T2 a) { return $$; }
```
Find function definition immediately returning a value.

qC:
```
f($$, $$, $$);
```
Find three parameter function.

qD:
```
for(a = 0; a < $$; a++) { $$ }    $|
a = 0; while(a < $$) { $$ a++; }
```
Find `for` or (represented by `$|`) `while` statement with a control variable.

Table 2. Target and Execution Result by `ccgrep`

Target		Antlr	Ant	Git	PgSQL	Linux
Lang.		Java	Java	C	C	C
#file		678	1,272	339	904	15,123
#line		59,511	138,396	90,495	177,174	3,756,212
qA	#found	0	2	8	3	48
	time(sec.)	**1.12**	**1.32**	**1.11**	**1.43**	**9.46**
qB	#found	159	161	7	27	543
	time(sec.)	**1.15**	**1.33**	**1.10**	**1.47**	**10.15**
qC	#found	1,710	2,487	5,717	10,603	187,653
	time(sec.)	**1.20**	**1.38**	**1.13**	**1.55**	**12.01**
qD	#found	1	13	442	621	10,754
	time(sec.)	**1.19**	**1.52**	**1.10**	**1.49**	**11.06**

Antlr: Antlr4 v.4.7.2, Ant: Apache Ant v.1.10.5, Git: v.1.6.4.3,
PgSQL: PostgreSQL v.6.5.3, Linux: Linux kernel v.2.6.14rc2

Table 2 shows the size metrics of the target, the number of found snippets, and the execution time of each query on a workstation with Intel Xeon E5-1603v4 (@2.8GHz $\times$ 4), 32GB RAM, and Windows 10 Pro for WS 64bit.

As we can see from Table 2, the execution times are about 1-10 sec. even for a few million lines of Linux kernel target. We would think that those are fast and acceptable as a daily-use tool. The execution times for qA to qD are very stable for each target. For example, in the case of Linux, they are about 10 sec. even for the small #found case (48 for qA) and the large #found case (187,653 for qC). Thus, we would say that the execution time is not heavily affected by the result size (#found) but mainly affected by the target size (#line). Targets Ant in Java and PgSQL in C have similar sizes around 140-180 Klines, and the execution times are also similar around 1-1.5 sec. This would show that the execution time is not strongly affected by the target language.

For comparison to `grep` we have employed a query qE, that is almost equivalent to qA except qE does not match the targets with more than one line.

qE(`grep`**):**

```
([a-zA-Z_][a-zA-Z_0-9]*)\s*<
([a-zA-Z_][a-zA-Z_0-9]*)\s*\?\s*
\1\s*:\s*\2
```

This query is complex and hard to create for inexperienced `grep` users. It has been executed by `grep` 3 to 9 times faster than `ccgrep`, but it missed some expected matches of the code snippets with two or more lines.

As the conclusion of RQ3, although the speed of `ccgrep` is slower than `grep`, it is sufficiently fast and acceptable as a search tool even for large targets such as 3 million LOC Linux kernel.

6 Related Works

There are numerous publications on code clone detection methods and their tools[18, 20]. Most of those tools focus on finding all of the code clone pairs in the target file collection. They report all code clones or similar code snippets with similarities higher than a certain threshold. Precisely controlling the matches with meta-symbols like ours cannot be accomplished by those approaches.

There are several tools specialized for finding code snippets. CBCD has been designed for finding related code snippets from a buggy code snippet, by using matching of Program Dependence Graph (PDG)[16]. It can be used to find type 1, 2, and 3 clones; however, the matching generally requires a long pre-processing time to construct PDG, and so this approach would not fit the nimble clone finding that we are interested in. NCDSearch has been designed to find similar code snippets in the pile of source code files for the analysis of code reuse and evolution[12]. The approach would be unique and interesting, but the speed is slower than ours. Micro-clones are recently getting focus due to their importance[4, 13]. Our tool is one of the convenience tools for finding micro clones.

Siamese has been developed for finding code clone pairs for a query method or file using multiple representations of n-gram token sequences with inverted index[17]. It requires a long indexing time (e.g., about 10 minutes indexing time for 10,000 method target). Thus its application and usage would be different from ours.

Variants of `grep` such as context grep `cgrep`, approximate grep `agrep`, and many others had been proposed and implemented to meet various requirements[1]. However, there is no one for clone-based matching like ours. Semantic-based matching tool `sgrep`[5], data-structure-based matching tool `coccigrep`[15], and the logic-based query pattern capturing language[22] were proposed, where the specific notations for the queries are provided without using the notion of clones like ours.

7 Conclusions

We have presented `ccgrep` that effectively finds code snippets in the target files with the notion of code clone and meta-pattern. It is a practical and effective pattern matching tool, easy-to-use to many software engineers.

As a future direction, we are interested in further performance improvement by using more efficient pattern matching algorithms. Also, we are trying to spread the use of `ccgrep` to industry collaborators who are trying to detect similar bug patterns in their legacy systems.

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Number 18H04094, and Osaka University Program for Promoting International Joint Research. We are grateful for important comments from T. Kamiya, N. Yoshida, Raula Gaikovina Kula, E. Choi, K. Takenouchi, T. Kanda, and M. Matsushita.

Table 3. Token-Level Matching

token(s) in query	matched token(s) in target	simple example of match	
		query	target
reserved word†	exact reserved word	`while`	`while`
delimiter	exact delimiter	`(`	`(`
identifier	any identifier‡	`myname`	`abc`
literal	any literal‡	`1`	`100`
$identifier	exact identifier	`$myname`	`myname`
$literal	exact literal	`$1`	`1`
$.	any single token	`$.`	`if`
$# X	any shortest token sequence ending with X	`$# +`	`while(f(a+`
$$ X	any shortest token sequence ending with X, excluding X inside well-balanced bracket {...}, [...], or (...)	`$$ +`	`while(f(a+1))+`
X $— Y	either X or Y	`+ $| -`	`-`
X $*	repeated sequence of X zero or more times	`( $*`	`(((`
X $+	repeated sequence of X one or more times	`( $+`	`((`
X $?	X or none	`( $?`	`(`
$(X1 X2 ... $)	X1, X2, ... (group for further regular expression operations)	`$( a++ $| ++a $)`	`a++`

†Type names are treated as identifiers.
‡Identifier and literal may match only the exact one by an option.
- Tokens starting with $ are meta-tokens and others are regular tokens.
- Wildcard meta-tokens $# and $$ match in reluctant way, and $*, $+, and $? match in possessive way[10].
- X, Y, X1, X2, ... are any regular token or a group with $(... $).

Appendix: Formulating Matching

Here we formulate the matching made by `ccgrep`. The input of the matching is the query q, the target T of source code files in a programming language L, and matching option o. The output is a list of matched code snippet t in T. We refer to reserved words, delimiters (operators, brackets, ; ...), identifiers, and literals in L as *regular tokens*. Other tokens starting with meta symbol $ are called *meta-tokens*. q is a sequence of regular tokens and the meta-tokes, and each matched result t is a sequence of the regular tokens. These token sequences do not contain comments, white spaces, or line breaks. We always consider the matching on the token sequence level, not on the character level.

In Table 3, we define a token-level matching for various kinds of tokens with simple examples. The basic ideas of these matches are as follows.

– A language-defined token such as reserved words or delimiters matches the exact token.
– A user-defined token such as an identifier or literal can match the same kind of token with a possibly different name or value. To pin down them to a specific identifier name or literal value, $ is used before the token. For example, `$count` would match only the token `count`.
– Wildcard tokens $., $#, and $$ are introduced for the matches to any single token, any token sequence, or any token sequence discarding paired brackets, respectively.
– Popular regular expression operators for choice, repetition, and grouping are introduced to enhance the expressiveness.

Consider that query q is a token sequence $q_1, ..., q_m$ $(1 \leq m)$, and a target t is a token sequence $t_1, ..., t_n$ $(0 \leq n)$. From q_1 to q_m, if each token in the query matches tokens in the target from t_1 to t_n as defined in Table 3 without overlapping or orphan tokens, then we say q matches t.

For the query token sequence $q_1, ..., q_m$ and the target token sequence $t_1, ..., t_n$, if $n = m$ and $norm(q_i) = norm(t_i)$ for each i, then q matches t as *type 2 matching*. Here $norm$ is a normalization function to flat the distinction of identifiers (or literals), defined below.

$$norm(x) \equiv \begin{cases} \#id & \text{if } x \text{ is an identifier} \\ \#li & \text{if } x \text{ is an literal} \\ x & \text{otherwise} \end{cases}$$

In type 2 matching, an identifier in the query can match any identifier in the target, and also a literal in the query can match any literal in the target.

q1: `a = 0; b = 10;`
t1: `x = 10; y = 200;`

q1 matches t2, because the sequences of the normalized tokens are both $[\#id, =, \#li, ;, \#id, =, \#li, ;]$.

A special case of type 2 matching, with a constraint such that for any identifier or literal q_i if $q_i = q_j$, then $t_i = t_j$, is *Parameterized matching* or *P-matching*. This is sometimes referred to consistent or aligned matching, meaning the same identifiers (or literals) in the query are mapped into the same ones in the target. P-matching is formally defined with a specialized normalization function $norm_p()$, as follows.

$$norm_p(x) \equiv \begin{cases} \#id_{pos(x)} & \text{if } x \text{ is an identifier} \\ \#li_{pos(x)} & \text{if } x \text{ is a literal} \\ x & \text{otherwise} \end{cases}$$

Here, $pos(x)$ is a function returning position i such that identifier (or literal) x is the i-th identifier (literal) newly appeared in the token sequence. Note that any meta-token starting with $ in the query and their matched tokens in the target are out of consideration of $pos()$.

q2: `a = 0; a = a + b;`
t2: `y = 0; y = y + c;`

For q2, $pos(a) = 1$ and $pos(b) = 2$, and for t2, $pos(y) = 1$ and $pos(c) = 2$. q2 matches t2 as P-matching, because the P-normalized sequences are both $[\#id_1, =, \#li_1, ;, \#id_1, =, \#id_1, +, \#id_2, ;]$. The following case is type 2 matching but not P-matching.

q3: `a = 0; a = a + b;`
t3: `y = 0; y = z + c;` (type 2 matching but not P-matching)

At t3, `z` cannot be matched by `a` because $norm_p(a) = \#id_1$ is not equal to $norm_p(z) = \#id_2$. As a default of CC matching, P-matching is assumed but it can be changed by the tool's option.

References

1. Abou-Assaleh, T., Ai, W.: Survey of global regular expression print (grep) tools. In: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.3326 (2004)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. ACM Press, Addison-Wesley, New York (1999)
3. Baker, B.S.: A program for identifying duplicated code. Proc. of Computing Science and Statistics: 24th Symposium on the Interface 24 pp. 49–57 (1992)
4. Beller, M., Zaidman, A., Karpov, A., Zwaan, R.A.: The last line effect explained. Empirical Softw. Engg. **22**(3), 1508–1536 (2017). https://doi.org/10.1007/s10664-016-9489-6
5. Bull, R.I., Trevors, A., Malton, A.J., Godfrey, M.W.: Semantic grep: Regular expressions + relational abstraction. In: Ninth Working Conference on Reverse Engineering, 2002. Proceedings. pp. 267–276 (Nov 2002). https://doi.org/10.1109/WCRE.2002.1173084
6. Carter, S., Frank, R., Tansley, D.: Clone detection in telecommunications software systems: A neural net approach. In: Proc. Int. Workshop on Application of Neural Networks to Telecommunications. pp. 273–287 (1993)
7. Cordy, J.R., Roy, C.K.: The nicad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 219–220 (June 2011). https://doi.org/10.1109/ICPC.2011.26
8. FreeSoftwareFoundation: Gnu grep 3.3 manual (2018), https://www.gnu.org/software/grep/manual/grep.html
9. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press, New York, NY (1997)
10. Habibi, M.: Java Regular Expressions: Taming the Java.util.regex Engine. Apress (2004). https://doi.org/10.1007/978-1-4302-0709-2
11. Inoue, K., Miyamoto, Y., German, D.M., Ishio, T.: Code clone matching: A practical and effective approach to find code snippets. arXiv **CS.SE**(2003:05615v1), 1–11 (2020)
12. Ishio, T., Maeda, N., Shibuya, K., Inoue, K.: Cloned buggy code detection in practice using normalized compression distance. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018. pp. 591–594 (2018)
13. Islam, J., Mondal, M., Roy, C., Schneider, K.: Comparing bug replication in regular and micro code clones. In: 27th International Conference on Program Comprehension (ICPC19). pp. 81–92 (05 2019)
14. Kernighan, B., Pike, B.: The Practice of Programming. Addison-Wesley (1999)
15. Leblond, E.: Coccigrep introduction, http://home.regit.org/software/coccigrep/
16. Li, J., Ernst, M.D.: Cbcd: Cloned buggy code detector. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 310–320 (June 2012). https://doi.org/10.1109/ICSE.2012.6227183
17. Ragkhitwetsagul, C., Krinke, J.: Siamese: Scalable and incremental code clone search via multiple code representations. Empirical Software Engineering **24**(4), 2236–2284 (2019)
18. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165–1199 (2013)
19. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: Proceedings of the 34th International Conference on Software Engineering. pp. 255–265. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012), http://dl.acm.org/citation.cfm?id=2337223.2337254

20. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming **74**(7), 470 – 495 (2009)
21. Singer, J., Lethbridge, T.C.: Whatś so great about 'grep'? implications for program comprehension tools. In: Tech. Rep., National Research Council, Canada (1997)
22. Sivaraman, A., Zhang, T., Van den Broeck, G., Kim, M.: Active inductive logic programming for code search. In: Proceedings of the 41st International Conference on Software Engineering. pp. 292–303. IEEE Press (2019)
23. Soetens, Q.D., Demeyer, S.: Studying the effect of refactorings: A complexity metrics perspective. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology. pp. 313–318 (Sep 2010). https://doi.org/10.1109/QUATIC.2010.58
24. Svajlenko, J., Roy, C.K.: Evaluating clone detection tools with bigclonebench. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 131–140. IEEE (2015)