Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the Eighth International Workshop on Software Clones (IWSC 2014)

Robust Parsing of Cloned Token Sequences

Ole Jan Lars Riemann and Rainer Koschke

20 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Robust Parsing of Cloned Token Sequences

Ole Jan Lars Riemann[1] **and Rainer Koschke**[2]

[1]ojlr@tzi.de [2]koschke@tzi.de
http://www.informatik.uni-bremen.de/st/
Software Engineering Group
University of Bremen, Germany

Abstract: Token-based clone detection techniques are known for their scalability, high recall, and robustness against syntax errors and incomplete code. They, however, may yield clones that are syntactically incomplete and they know very little about the syntactic structure of their reported clones. Hence, their results cannot immediately be used for automated refactorings or syntactic filters for relevance.

This paper explores techniques of robust parsing to parse code fragments reported by token-based clone detectors to determine whether the clones are syntactically complete and what kind of syntactic elements they contain.

This knowledge can be used to improve the precision of token-based clone detection.

Keywords: software clones, lexical analysis, syntactic analysis, token-based clone detection, syntax-based clone detection

1 Introduction

Token-based clone detection techniques are often based on suffix trees (or a variant thereof, namely, suffix arrays) [Bak92, KKI02, GK09, Kos12] or hashing of token sequences [LHMI07, HJHC10, KRC11]. The underlying algorithms require linear time and linear space to locate clones and scale very well to large programs. They are also known for high recall [BKA+07]. Because their input requires only lexical analysis, they are robust against syntax errors and incomplete code. Their disadvantage, however, is that they may yield clones that are syntactically incomplete. As a consequence, they have lower precision than syntax-based clone detectors [BKA+07] and their results cannot immediately be used to trigger automated refactorings. Filters used to improve precision of token-based techniques cannot leverage syntactic information because that information is not provided by lexical analysis. Sometimes token-based clone detectors use simple heuristics instead, such as counting balanced brackets, but these heuristics may not always work. Determining whether a token sequence forms a complete syntactic unit and what syntactic unit it is, requires a syntax analysis. For this reason, several techniques are based on syntax trees instead. For instance, Baxter et al.'s technique hashes syntax subtrees to detect clones [BYM+98]. Koschke et al. use suffix trees to determine equal sequences in the serialized syntax trees [KFF06, FKF08]. These techniques yield only syntactic clones and they provide a detector with richer syntactic information to filter or rank clones for relevance (for instance, one can rank nested while loops as more relevant than consecutive while loops).

Syntax analysis, however, requires parsing and may not always work when code is incomplete or syntactically incorrect. In particular, for languages with a preprocessor such as C or C++ detecting clones based on the non-preprocessed code is difficult for syntax-based detectors.

In this paper, we explore ways to enrich token-based clone detectors with syntax information. We do that by parsing the output of a token-based clone detector, that is, *after* the clones were detected, while traditional syntax-based techniques run the syntax analysis *before* the clones are detected. All token-based detectors can benefit from this strategy, that is, this strategy is independent of the actual clone detector and can be implemented once for all.

Because we parse only fragments, a full syntax analysis that attempts to derive the root non-terminal of the underlying grammar will hardly ever work. Also, because we do not assume that the fragment is syntactically complete, that is, is derived from any of the syntax rules of the underlying grammar, normal parsing approaches will fail in most cases. We need parsing that is robust against syntactically incomplete code. We also need a way to deal with preprocessor statements in the code to be able to handle C and C++.

Contributions. To this end, this paper explores different strategies of robust parsing to parse code fragments reported by token-based clone detectors to determine whether the clones are syntactically complete and what kind of syntactic elements they contain.

Overview. The remainder of this paper is organized as follows. Section 2 introduces Earley parsing, a very flexible technique that handles all types of grammars including ambiguous ones. Section 3 describes alternative extensions to the classical Earley parser for robustness against syntax errors. Section 4 describes our idea to handle preprocessor statements based on island grammars. Section 5 evaluates the different alternatives for robust Earley parsing. Section 6, finally, concludes.

2 Earley Parsing

Our goal is to parse isolated cloned code fragments written in complex languages such as C and C++ without their context and without semantic analysis. The grammar of C is ambiguous. Hence, we need a parsing technique that copes with ambiguous grammars. Earley's parsing technique is a very flexible technique that handles all types of grammars including ambiguous ones [Ear70]. For this reason, our parsing technique is based on a simplified version of Earley's general parsing technique. We implemented a simpler variant, with the aid of the remarks by Aycock and Horspool [AH02] instead of Earley's original look-ahead mechanism. This section describes the details of this approach.

2.1 Recognizing a Sentence

As most parsing techniques, an Earley parser is based on a context-free grammar defined as $G = \langle N, T, P, \phi \rangle$ where N denotes the set of non-terminal symbols, T the set of terminal symbols (or tokens), and P the set of productions. Adopting the notation of Earley, each production of P has the form $D_p \rightarrow C_{p1} \ldots C_{p\overline{p}}$ with $1 \leq p \leq \overline{P} \wedge D_p \in N \wedge C_{pj} \in (N \cup T)$ where $\overline{P}$ is the number of productions and $\overline{p}$ the number of symbols on the right-hand side of production p. $\phi \in N$ denotes an artificial start symbol that is used to mark the starting point of the grammar. The language

defined by the grammar G is denoted by $L(G)$. The sequence of input symbols is represented by $X_1 \ldots X_{n+1}$ with n as the number of actual input symbols. The artificial last symbol X_{n+1} is used to mark the end of input denoted by $\dashv$.

Given an input in this form, for each input symbol X_i an associated item set S_i is created. Each set contains unique items represented by a triple of a production, a position on the right-hand side of the production indicating how many symbols of that production have been read, and a reference to an earlier item set. Hereafter, this reference shall be known as *start index* and is described shortly. In favor of better readability, an item is written as $\langle u \to \alpha \bullet v\beta, c\rangle$ with $u \in N$, $v \in (N \cup T)$, $\alpha, \beta \in (N \cup T)^*$ and $1 \leq c \leq n+1$, where c denotes the *start index*. They are classified in *final* items, that is, items with the dot ($\bullet$) at the end of the production, and *active* items, that is, items with symbols of the production left to be read.

Initially all sets are empty except the first set S_1, which is initialized with the so-called *start item* $\langle \phi \to \bullet C_{\phi 1} \ldots C_{\phi\overline{\phi}}, 1\rangle$. After that, the items of a set are processed successively in order of their creation. After all items of a set are handled, the algorithm advances to the items of the successive set.

Given the current item set S_i. For each item in S_i, one of the three operations *Predictor*, *Scanner*, or *Completer* is executed.

The **Predictor** operation applies to *active* items of the form $\langle u \to \alpha \bullet v\beta, c\rangle$ whose next symbol is a non-terminal $v \in N$. For each production $p \in P$ such that $v = D_p$, this operation adds the item $\langle D_p \to \bullet C_{p1} \ldots C_{p\overline{p}}, i\rangle$ to S_i. Because S_i is uniquely associated with the input symbol X_i, the *start index* captures the token that triggered the attempt to derive production $D_p \to \bullet C_{p1} \ldots C_{p\overline{p}}$. This information will later be used by the *Completer* operation.

The **Scanner** operation is used if the next symbol of an *active* item $\langle u \to \alpha \bullet v\beta, c\rangle$ is a terminal symbol $v \in T$. It adds the item $\langle u \to \alpha v \bullet \beta, c\rangle$ to the next set S_{i+1}, but only if $v = X_i$.

The **Completer** operation is executed for *final* items $\langle u \to \alpha\bullet, c\rangle$. That means a production was successfully applied. The step adds an item $\langle u' \to \alpha' v' \bullet \beta', c'\rangle$ to the current set S_i for each item $\langle u' \to \alpha' \bullet v'\beta', c'\rangle \in S_c$ such that $v' = u$. While a scanner operation moves the dot $\bullet$ past a terminal, the completer moves it past a non-terminal.

After there are no further unprocessed items the algorithm can decide whether the input sequence $X_1 \ldots X_n$ is a sentence of the language $L(G)$ or not. The input sequence is a sentence of the language iff $\exists I \in S_{n+1} : I = \langle \phi \to \alpha\bullet, 1\rangle$. Such *final* items shall be called *accept items* hereafter. In case of a syntax error, it is possible that some of the sets remain empty. The location of the first unexpected input symbol can be determined by the index of the first empty set. If none of the sets are empty but the last one does not contain an accept item, the input sequence is an incomplete sentence of the language.

Figure 1 shows a complete run of the algorithm for a simple example[1]. The first column denotes the id of each item. It is followed by the production and the *start index*. Each item has attached a letter that indicates, which one of the operations ***P**redictor*, ***S**canner* or ***C**ompleter* it has triggered. **S** in parentheses indicates that the *Scanner* operation was conducted but the next symbol of the item did not match the corresponding input symbol, so that no additional item was created.

[1] At first sight, it resembles the example given by Earley. Actually it yields slightly different sets, since the presented version of the algorithm uses no look-ahead.

$N = \{E, T, F, \phi\}$ $T = \{\text{'+'}, \text{'*'}, \text{'('}, \text{')'}, \text{v}\}$ $\phi = E$

$P = \{E \to E\ \text{'+'}\ T,\ E \to T,\ T \to T\ \text{'*'}\ F,\ T \to F,\ F \to \text{'('}\ E\ \text{')'},\ F \to \text{v}\}$

S_1: v			
1	$\phi \to \bullet\, E$	1	P
2	$E \to \bullet\, E$ '+' T	1	P
3	$E \to \bullet\, T$	1	P
4	$T \to \bullet\, T$ '*' F	1	P
5	$T \to \bullet\, F$	1	P
6	$F \to \bullet$ '(' E ')'	1	(S)
7	$F \to \bullet$ v	1	S

S_2: '+'			
8	$F \to$ v $\bullet$	1	C
9	$T \to$ F $\bullet$	1	C
10	$E \to$ T $\bullet$	1	C
11	$T \to$ T $\bullet$ '*' F	1	(S)
12	$\phi \to$ E $\bullet$	1	
13	$E \to$ E $\bullet$ '+' T	1	S

S_3: v			
14	$E \to E$ '+' $\bullet\, T$	1	P
15	$T \to \bullet\, T$ '*' F	3	P
16	$T \to \bullet\, F$	3	P
17	$F \to \bullet$ '(' E ')'	3	(S)
18	$F \to \bullet$ v	3	S

S_4: '*'			
19	$F \to$ v $\bullet$	3	C
20	$T \to F\ \bullet$	3	C
21	$E \to E$ '+' $T\ \bullet$	1	C
22	$T \to T\ \bullet$ '*' F	3	S
23	$\phi \to E\ \bullet$	1	
24	$E \to E\ \bullet$ '+' T	1	(S)

S_5: v			
25	$T \to T$ '*' $\bullet\, F$	3	P
26	$F \to \bullet$ '(' E ')	5	(S)
27	$F \to \bullet$ v	5	S

S_6: $\dashv$			
28	$F \to$ v $\bullet$	5	C
29	$T \to T$ '*' $F\ \bullet$	3	C
30	$E \to E$ '+' $T\ \bullet$	1	C
31	$T \to T\ \bullet$ '*' F	3	(S)
32	$\phi \to E\ \bullet$	1	
33	$E \to E\ \bullet$ '+' T	1	(S)

Figure 1: Simple example grammar and the sets of items created for the input `1 + 2 * 3`. The symbol *v* for *value* denotes the token type of a literal.

2.2 Constructing Syntax Trees

The described algorithm is only a recognizer, that is, it decides only whether an input sequence is a sentence of the language. To gain syntactic information, however, a syntax tree is required. It is impossible to construct a syntax tree with only the items given. The parser has to know which item participated in the creation of which other item. We follow the proposal of Aycock and Horspool [AH02] and store two different types of links during the recognizing step. If an item I is created by a *Scanner* operation triggered by item I_p, a predecessor link $I \xrightarrow{\Lambda} I_p$ is recorded. Items created by a *Completer* operation trigger the creation of a *predecessor link* $I \xrightarrow{I_c} I_p$ and a *causal link* $I \to I_c$, where I is the item created by the operation and I_p is the item depending on the completion of I_c. The *Predictor* operation records no links because this relation between items is not required for the syntax tree.

After all these links are recorded during the recognition step, they can be used to get the predecessor of a specific item. Starting with a complete item, that is, an item that represents the parsing of a complete production inclusive all its subproductions, the symbol on the left of the dot (•) is checked. If it is a terminal symbol, the associated token can be determined with the help of the set of the item. Items with a non-terminal symbol in front of • require the creation of an entire subtree for this symbol. Which item is used is specified by the *causal link*. The original algorithm as described by Aycock and Horspool can handle only one unique *predecessor link* and one *causal link*, respectively, for an item, so that, in case of ambiguities, only one of all possible syntax trees is created. Because we want a syntax tree containing all valid derivations, we augmented the algorithm with backtracking that follows all links if there are more than one.

3 Robust Earley Parsing

Earley's algorithm described in Section 2 handles ambiguity of identifiers and type names in C successfully. Yet, in its original form, it cannot handle syntax errors and incomplete code.

3.1 Partial Syntax Trees

We first address code fragments containing syntactic structures cut off at the back, that is, code fragments that are a prefix of a sentence of the language. Originally, the algorithm requires at least one *accept item* in the last set to construct a complete syntax tree. Obviously we can search for such an item in prior sets if it is not already present in the last set. This approach is also applicable in the case of an empty set caused by a syntax error. Traversing the sets in reverse order to find an *accept item*, the algorithm can use the first found *accept item* to construct the maximally large partial syntax tree of a particular code fragment.

3.2 Multiple Start Symbols

Another issue are code fragments with syntactic structures cut off at the beginning, that is, code fragments that are a suffix of a sentence of the language. In this case, it is very likely that the fragment is no sentence that can be derived from the start symbol of the grammar. Due to its predictive nature of the algorithm, inherent by the *Predictor* operation, it is not straightforward to let it consider sentences that are derived from other non-terminal symbols of the grammar. As an ad-hoc approach, the parser can be modified easily to allow multiple start symbols. It requires to define additional suitable start symbols explicitly. For the used grammar of C, `block-item-list` was chosen as sole additional start symbol beyond the default root production of `translation-unit`. We selected `block-item-list` because it subsumes all structures contained in a function definition, more precisely, any list of statements and declarations, while `translation-unit` can be used to construct an arbitrary sequence of external declarations, that is, global variables, constants and function declarations and definitions.

3.3 Panic Mode

Both approaches mentioned above could be combined in order to try to recover from syntax errors. Yet, it would require to restart the parser at each occurrence of a syntax error and it is unlikely that this restarted parser succeeds to parse the remaining code after the error without further restarts. A more systematic attempt that tries to correct errors is more reasonable.

Aho and Peterson introduced the concept of *covering grammar* [AP72] to cover expected errors in the grammar. Our own initial tests with short input showed that this heavy-weight approach is too slow for large grammars such as C. The so-called *panic mode*, on the other hand, is described by Grune and Jacobs as "probably the simplest error recovery method that is still somewhat effective" [GJ08]. So, we decided to adopt this simple *panic mode* strategy as a promising alternative.

A parser with integrated panic mode behaves like a normal parser if no syntax error occurs. Only in case of a syntax error, the panic mode becomes active. In this mode, it skips input symbols until one of a predefined set of *synchronizing symbols* or the end of input is reached.

Synchronizing symbols are chosen from the set of *terminal* symbols. If such a symbol was found within the sequence of input symbols, the panic mode puts the parser into a state in which it can process the symbol.

Applied to the Earley Parser, the panic mode becomes active as soon as the parser advances to an empty item set. The index of the item set before this set is marked as *panic index* i_p. The following sets are skipped until the first set with a synchronizing input symbol is found. This item set is referred to as *recovery set* in the following. Since the parsing algorithm can trace multiple alternative derivations in parallel, it is somewhat more complex to restore a valid state than, for instance, in a recursive descent parser. For that type of parser it would suffice to ascend the prediction stack until a production with a matching symbol on the right-hand side is found. The panic mode for Earley's algorithm copies all items of the set S_{i_p} to the *recovery set*. To reduce the amount of uselessly copied items, *FIRST* and *FOLLOW* sets (see [AU77]) can be used to determine if at least the input symbol of the *recovery set* can be read with a specific item. This strategy cannot guarantee that the entire production of the item can be read. Furthermore, the panic mode can distinguish between different types of syntax errors. If the next symbol of an item copied into the *recovery set* equals the input symbol of this set, it indicates the correction of an *insertion error* and the • will not be moved. In contrast, the • is advanced if the symbols do not match, which means that a *deletion* or a *replacement* error is corrected. It is possible that the panic mode cannot find suitable items for the *recovery set*. In this case, the parser has to report an error and terminate. However, it could still be possible to create a partial syntax tree from a potential *accept item* of one of the non-empty sets as described before. When the parser with panic mode fails at some input symbol not at the end, it is restarted from this point. Combined with multiple start symbols it still has the chance to parse some symbols of the remainder.

We chose `;`, `(`, `)`, {, }, and all keywords as the *synchronizing symbols* for C.

3.4 Suffix Parsing

The described panic mode seems to be an acceptable solution for code fragments with simple syntax errors anywhere in the middle of the sequence of symbols. However, it shows weaknesses when it ought to process fragments which are cut off at the beginning. This drawback can be reduced with additional start symbols but only for a limited selection. Since it is likely that cloned fragments are cut off at the beginning, it would be useful to have a parser that deals with this problem specifically. Nederhof and Bertsch [NB96] present a so-called *suffix parser* based on Earley's algorithm. The general version of this parser is primarily used as a tool for the development of a linear suffix parsing technique for *LL(1)* languages but it can also be used to parse a suffix of a sentence of a *general* context-free language, as shown by Grune and Jacobs [GJ08].

To convert a "normal" Earley Parser into a *Suffix Earley Parser*, it is necessary to let the parser consider not only the productions with the same prefix as the input stream but also those whose suffix is a prefix of the input. The set of productions that have possibly the same prefix as the input is created implicitly by the *Predictor* operation when it processes the items of the first set (S_1). So, to extend this set of predicted productions, we have to change the *predictor* operation or create the set explicitly. Then the question is raised which items should be contained in this set to represent the beginning of all possible suffixes of all possible sentences of a given language? The answer is simple. It has to contain all possible items with the same start index. As another

modification, the start index of these items is not allowed to be set to 1 as done before. Because these items represent suffixes of sentences, at this point it is not known how many symbols of the input sequence are cut off, that is, the start position of a later found suffix is unknown and lies somewhere in front of the beginning of the parsed code fragment. This fact is represented by a special placeholder start index Λ for *suffix items*.

After the first set has been initialized with *suffix items*, Earley's algorithm is applied. The operations *Predictor* and *Scanner* remain unchanged, since they ignore the *start index*. Solely the *Completer* operation uses the *start index* and requires an adaptation. For *suffix items* it has to deal with an undefined *start index* Λ. The *Completer* operation identifies items depending on the completion of the currently processed item. The *start index* of the currently processed item is used to determine the set which contains these items. Because all *suffix items* are created in the first set, the *Completer* operation can assume a *start index* of 1 for *suffix items*. However, the operation may only consider *suffix items* of the first set if the currently processed item is a *suffix item*, too. Otherwise, a *suffix* would contribute to the completion of a *prefix* of a sentence.

Beyond the modification of the *Completer* operation, it is required to change the condition that is used to determine if an input is a sentence of a given language. If there is an *accept item* in S_{n+1}, it still means that the input is a sentence of the language. But there is no such item, if the input is no sentence of the language, but only a suffix of a sentence of the language. In this case, the algorithm looks for *final suffix items*. Since *suffix items* are created only in the first set, a *final suffix item* in the last set implies that the input is a complete suffix of a sentence.

The *Suffix Parser* can be seen as an implicit modification of the given grammar. Would this modification be explicit, it would make all symbols of the right-hand side of all productions optional. This can lead easily to an infinitely ambiguous grammar. Such ambiguities cause cycles in the data structures used to create syntax trees after the recognition process. Applied to the *Suffix Parser*, the problematic situation is outlined in Figure 2.

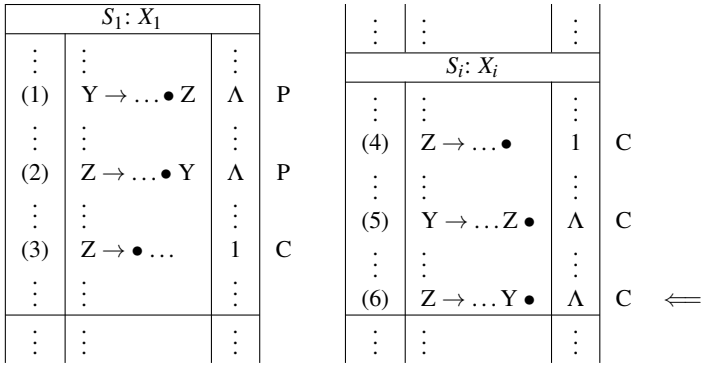


Figure 2: Suffix items causing cycles during the creation of syntax trees

The example assumes that the grammar contains the productions $Y \to \ldots Z$ and $Z \to \ldots Y$. The dots (…) represent a non-empty sequence of further symbols. At the beginning of the algorithm, the items (1) and (2) are created. Both items trigger the *Predictor* operation. It is sufficient to focus on the operation of item (1). It has to create at least the item (3), which causes eventually the creation of the *final* item (4) as element of some later set S_i. Item (4) triggers a *Completer*

operation, which searches for *active* items whose next symbol is Z. In S_1 it finds item (1) and copies it as item (5) to S_i. Item (5) is again *final* and causes a *Completer* operation which creates item (6). At this point the cycle occurs. The *Completer* operation required by item (6) would create an item which would be exactly the same as item (5). The set properties of the set prevent the reinsertion of that item, but for the later creation of syntax trees, a link between item (5) and item (6) is added, which forms a cycle together with the link from item (6) to item (5) created by the *Completer* operation of item (5). The algorithm used to create syntax trees (see Section 2.2) cannot detect such cycles and would not terminate in this case.

In order to still be able to gather results with the given syntax tree creation, a simplified version of the suffix parsing algorithm was implemented. Instead of modifying the *Completer* operation, *final suffix items* are just skipped. This leads to the disadvantage that only suffixes of a production of the grammar are recognized and not general suffixes of the language. An exemplary run of the simplified suffix parsing algorithm is given in Figure 3. It uses the same notation as Figure 1 except for the additional dashed line that separates the initially created *suffix items* from the regular items of the first set. Furthermore Λ is used to mark *suffix items*.

$N = \{E, \phi\}$ $T = \{\text{'+'}, \text{'*'}, \text{v}\}$ $\phi = E$ $P = \{E \to E\ \text{'+'}\ E,\ E \to E\ \text{'*'}\ E,\ E \to \text{v}\}$

S_1: '+'			
1	$\phi \to \bullet E$	Λ	P
2	$E \to \bullet E$ '+' E	Λ	P
3	$E \to E \bullet$ '+' E	Λ	S
4	$E \to E$ '+' $\bullet E$	Λ	P
5	$E \to \bullet E$ '*' E	Λ	P
6	$E \to E \bullet$ '*' E	Λ	(S)
7	$E \to E$ '*' $\bullet E$	Λ	P
8	$E \to \bullet$ v	Λ	(S)
9	$\phi \to \bullet E$	1	P
10	$E \to \bullet E$ '+' E	1	P
11	$E \to \bullet E$ '*' E	1	P
12	$E \to \bullet$ v	1	P

S_2: v			
13	$E \to E$ '+' $\bullet E$	Λ	P
14	$E \to \bullet E$ '+' E	2	P
15	$E \to \bullet E$ '*' E	2	P
16	$E \to \bullet$ v	2	S

S_3: '*'			
17	$E \to$ v $\bullet$	2	C
18	$E \to E$ '+' $E \bullet$	Λ	
19	$E \to E \bullet$ '+' E	2	(S)
20	$E \to E \bullet$ '*' E	2	S

S_4: v			
21	$E \to E$ '*' $\bullet E$	2	P
22	$E \to \bullet E$ '+' E	4	P
23	$E \to \bullet E$ '*' E	4	P
24	$E \to \bullet$ v	4	S

S_5: $\dashv$			
25	$E \to$ v $\bullet$	4	C
26	$E \to E$ '*' $E \bullet$	2	C
27	$E \to E \bullet$ '+' E	4	(S)
28	$E \to E \bullet$ '*' E	4	(S)
29	$E \to E$ '*' $E \bullet$	Λ	
30	$E \to E \bullet$ '+' E	2	(S)
31	$E \to E \bullet$ '*' E	2	(S)

Figure 3: Simple example grammar and the sets of items created for input `+ 2 * 3`

3.5 Error Correcting Suffix Parsing

With the suffix parsing algorithm, syntax errors cannot be handled as selectively as with the panic mode. However, if a syntax error occurs, the parser can be restarted at the point of its occurrence in the hope that the remaining part of the input is a suffix of a sentence of the language. In the following an implementation of a streamlined variant of the error correcting suffix parser presented by Nederhof and Bertsch [NB96] is described. First it tries to detect a sentence of a suffix of the language as explained above. When the end of input or an empty set is reached, that is, a syntax error is detected, the recognizing step stops and searches for an *accept item* or a *final suffix item* in order to create a complete or partial syntax tree tree as depicted in Figure 4.

The index of the set after the set of the item picked for the syntax tree construction (S_j and S_{j+1}) is stored as starting point for the restarted suffix parser. This assures that the algorithm tries to reparse the already read part of the input that cannot be covered with a complete syntax tree. Before the parser is restarted, all sets have to be reset, that is, all their items have to be deleted. Additionally, the new starting set S_{j+1} has to be initialized with the initial *suffix items*. Since the computation of these items is independent of the current input symbol, a template set is prepared before the first parse and copied when needed. It is expected that this restarting strategy works quite well with some syntax errors, while it has to restart several times with other syntax errors. This seems not to be a big drawback, because in this case probably only a few input symbols are skipped until another suffix is found for which a syntax tree can be created. As mentioned, the selection of *accept items* for the syntax tree construction is very similar to the procedure described in Section 3.1, but now we are interested in suffixes as well. So we have to choose between regular *accept items* and *final suffix items*. Since the syntax tree of a complete sentence, represented by an *accept item*, appear more valuable than one of a suffix of a sentence of the same length, regular *accept items* are preferred over *final suffix items* within the same set. Only if none of those items are found, a prior set is considered.

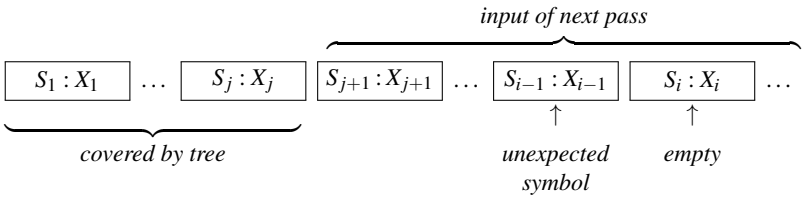


Figure 4: Restart of error-correcting suffix parser

4 Preprocessor

As already mentioned, one objective is to preserve the robustness of token-based clone detectors, in particular also with regard to unprocessed preprocessor statements. Usually, token-based clone detectors operate on non-preprocessed source code in order to report clones in the original code as viewed by the developer. So the parser for the clone fragments has to deal with preprocessor statements. The problem is that the preprocessor in C is not part of the actual grammar of C. As Baxter and Mehlich [BM01] state: "... *C preprocessor directives can occur between any pair of tokens.*" Baxter and Mehlich, hence, modify the basic grammar of C to allow preprocessor statements at places where they occur frequently in software systems. This solution may work well for systems with an orderly used preprocessor but will probably fail to process code fragments without those restrictions. Furthermore, it requires informed manual modifications of the grammar.

Two other approaches by Gazzillo and Grimm [GG12] and Kästner et al. [KGR+11, KGO11] respectively try to consider almost every possible evaluation of the preprocessor statements and the resulting variants of generated C code. Although these exhaustive strategies do not require a modification of the grammar based on an heuristic, they still require complete source code.

Because only the code fragments yielded by a clone detector are given without their embedding syntactic context in our case, this requirement cannot be satisfied.

The output of the preprocessor depends upon macro definitions within the code and passed to the preprocessor by way of command-line arguments. The latter are not known without knowledge about the build process. The former are neither known if they are set in header files not accessible to the parser. Only for macro definitions within the code visible to the parser, there would be a possibility of evaluation. It would require us to integrate a preprocessor to handle these visible directives. The benefit of that would be marginal, however, as we parse only fragments without further context and chances are very high that these do not contain sufficient preprocessor information to leverage an integrated preprocessor. That is why we decided to follow a different path.

For this worst-case scenario we instead use an island parser to identify the pieces of source code to be passed to the C parser. A so-called island grammar distinguishes between *islands*, that is, the part that is covered by the grammar, and *water*, that is, parts of the input the grammar for which no detailed productions exist [Moo01].

In order to apply island grammars on code fragments with preprocessor directives, it is required to decide which language is treated as *island* and which is treated as *water*. The first thought is to skip the problematic preprocessor part and define it as *water* but this would discard a lot of information we could gather. If a preprocessor statement is contained completely in a fragment, it is straightforward to construct a syntax tree that represents the structure of the fragment from the perspective of the preprocessor. Following this strategy, the parts between the preprocessor tokens are *water* and that way not examined by the island parser. Actually, they have not to be processed by the island parser because, we have an error correcting parser that is able to parse fragments of C code and each skipped fragment can be passed to an instance of this C parser. In fact, this makes the job of the C parser a little easier, because, with the exception of macro calls, the smaller fragments contain no preprocessor code. The similarity of macro and function calls makes it impossible for the island parser to distinguish between them. Therefore, they are treated as function calls.

For each of these subfragments, the error correcting parser constructs one complete or multiple partial suffix trees, if possible. These trees are inserted into the syntax tree created by the island parser afterwards. To get the right positions for those insertions, the island parser uses sentinel nodes which are replaced by the syntax trees of the subfragments, as depicted in Figure 5.

5 Evaluation

This section evaluates the different strategies for robust parsing of cloned token sequences, namely, *suffix*, *panic*, *rpanic*, and a baseline approach *none*. The baseline *none* stops at the first syntax error but still tries to create a syntax tree for the part it understood so far. Strategy *panic* skips tokens until a synchronizing token is found and restarts if the panic mode fails to restore a valid parser state and creates syntax trees afterwards (cf. Section 3.3). Strategy *suffix* implements the lightweight suffix parser described in Section 3.4 equipped with the error recovery described in Section 3.5. Strategy *rpanic* is a variant panic mode similar to *panic*. Both modes try to reach a state where parsing can be continued. If they fail, *panic* just restarts at the point of the failure while *rpanic* adopts the restart strategy of *suffix*, that is, depending on the

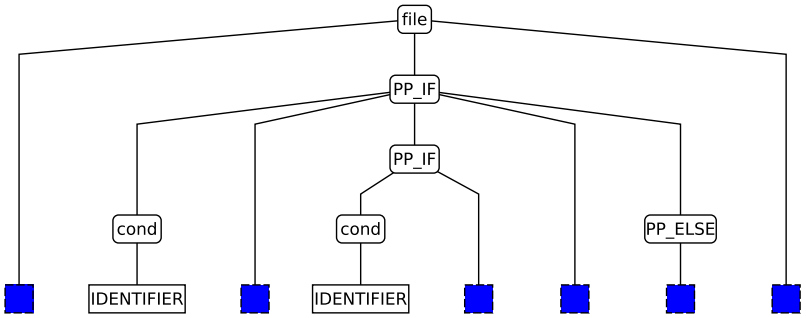


Figure 5: Syntax tree of preprocessor code with sentinel nodes

coverage of the syntax trees as described in Section 3.5. Thus, *rpanic* restarts at the first token that is not part of the partial syntax tree. All parsers are called by the island preprocessor parser.

The research questions addressed are:

RQ1 How much syntactic information can be gathered from the cloned token sequences using the alternative approaches for robust parsing?

RQ2 What are the recall and precision of the alternative approaches?

RQ3 What are the runtime costs of the alternative approaches?

5.1 Clone Detector

Our approach parses the token sequences gathered by any type of token-based clone detector. That is, the approach is technically independent of a particular detector. In this study, we used our own token-based clone detector *iClones*[2] [GK09] because we are familiar with it. We speculate that it is representative for token-based clone detectors in general, at least, for those based on suffix trees because they all share the same detection algorithm and differ only in their pre- and postprocessors. Whether the results obtained are actually comparable to those we would obtain with a different token-based clone detector is future work.

From the point of view of parsing, type-1 and type-2 clones are not different because the parser does not depend upon token values (e.g., `foo`) but only its token type (e.g., identifier). Furthermore we wanted to make sure that the same fragment is not entered twice in the evaluation. One simple solution to achieve this is to detect only type-1 clone classes and to select only one representative from each class as input for the robust parser. By definition, type-1 clone classes are true partitions of clone sequences. If we had mixed type-1 and type-2 clone classes, we could have had a fragment that is both in a type-1 and a different type-2 clone class. We excluded type-3 clones because the fragments in a type-3 clone class are all different and the results would have depended upon the choice of one of the fragments, which is arbitrary.

[2] http://www.softwareclones.org

system	version	SLOC	# clone classes	∅ fragment size
Handbrake	0.9.9	70.617	67	158.22
Apache HTTPD	2.4.4	146.715	80	152.68
GNU Wget	1.14	68.059	149	798.52

Table 1: Subject Systems; # denotes the number and ∅ the average size of fragments

Consequently, we used *iClones* with the following settings. Tokens were not transformed in any way; we gathered only type-1 clones, the input language was C; and clones had to have at least 100 tokens to be reported.

5.2 Subject Systems

We gathered our results for the systems listed in Table 1. The table gives their size in terms of source lines of code (SLOC) – counting only those lines that have at least one token as measured by David A. Wheeler's tool *SLOCCount* –, their number of type-1 clone classes, and their average number of tokens of cloned fragments. *Handbrake*[3] is a tool to convert video formats with a graphical user interface. *Apache HTTPD*[4] is a web server. *GNU Wget*[5] is a command-line tool to download files from the Internet. All systems are written in C and were used in previous studies by other authors, too.

5.3 Research Question RQ1

To assess RQ1, we gather the following measures for a token sequence S:

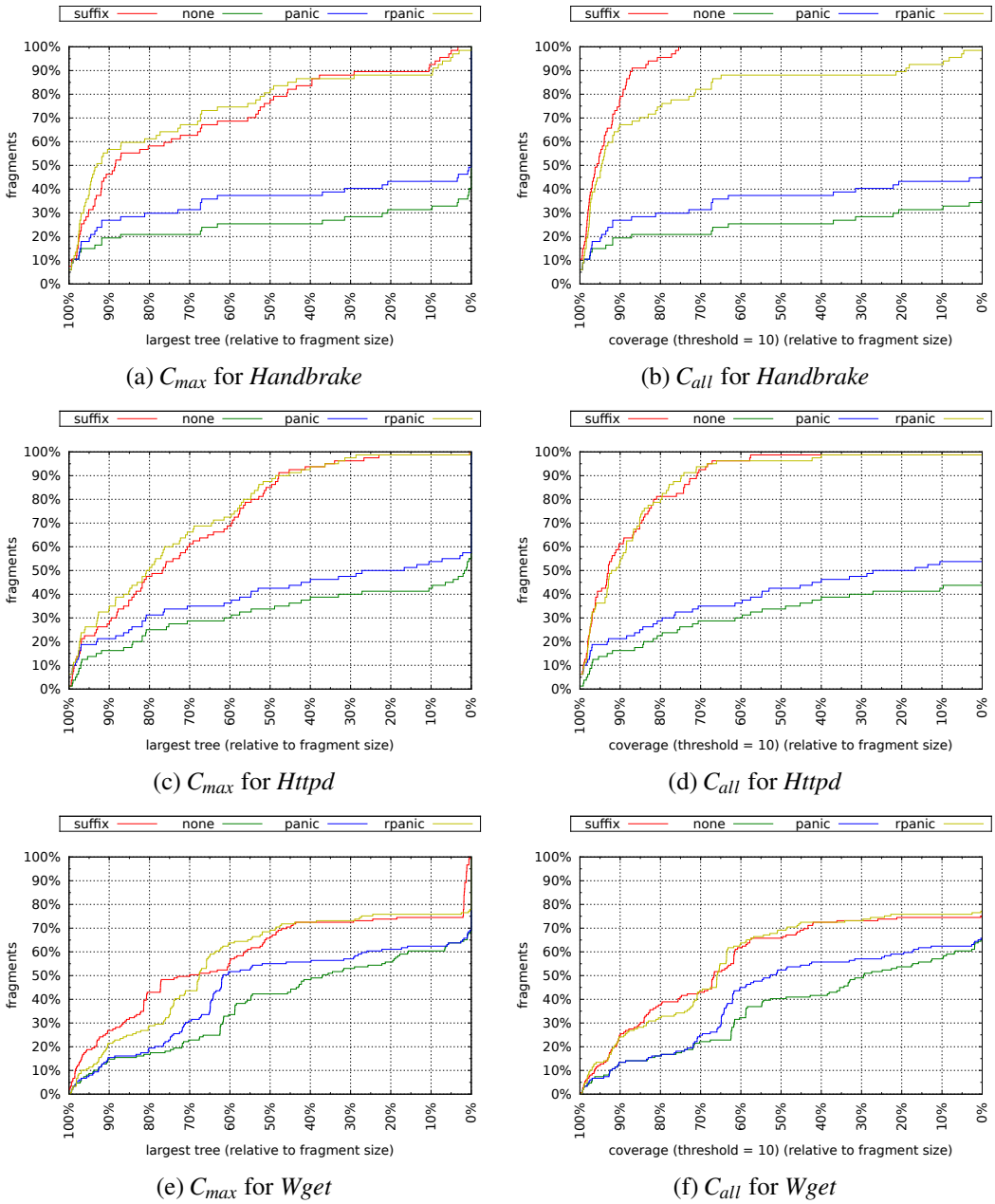
- coverage C_{max} of the maximally large syntax tree that can be derived from S
- coverage C_{all} of all syntax trees that can be derived from S

The sequence of tokens s_t covered by a syntax tree t is the sequence of leaves of t read from left to right. The coverage of a syntax tree t derived from a token sequence s is defined as $C_{max}(s) = |s_t|/|s|$. If $C_{max}(s) = 1$, the token sequence c was fully parsed. The coverage s_T of a set of syntax trees, T, derived from a token sequence s is as follows: $|\bigcup t \in T : s_t|/|s|$. This measure captures cases where the token sequence is not a complete syntactic structure but rather contains many smaller complete syntactic structures. It tells how well the combination of these smaller syntactic units explains the token sequence.

To ignore tiny, usually meaningless and ambiguous syntax trees, we used a threshold of 10 tokens for C_{all}. For instance, the relatively trivial token sequence `10,20` could be an initializer or parameter list. The threshold of 10 tokens was determined by manual inspection.

The results are shown in Figure 6. The x axis in the charts on the left show C_{max} and on the right they show C_{all}. The y axis in all charts show the relative cumulative number of fragments of the respective coverage measure.

[3] http://handbrake.fr/downloads.php
[4] http://archive.apache.org/dist/httpd
[5] http://ftp.gnu.org/gnu/wget



(a) C_{max} for *Handbrake*

(b) C_{all} for *Handbrake*

(c) C_{max} for *Httpd*

(d) C_{all} for *Httpd*

(e) C_{max} for *Wget*

(f) C_{all} for *Wget*

Figure 6: Coverage of the alternative approaches

We can observe that the strategies *none* and *panic* are always worse than the other two strategies, where *panic* is able to gather about 10 % larger trees than *none*. Strategy *rpanic* implements a more sophisticated restart and as the charts show is capable to cover more tokens than the simpler *panic* mode as a consequence.

The strategies *suffix* and *rpanic* are close to each other. Sometimes *rpanic* is slightly better than *suffix* in terms of C_{max}, sometimes it is slightly worse. Strategy *rpanic* skips tokens in case of syntax errors and is still able to construct a tree, while *suffix* will always restart. As a consequence, the largest trees of *suffix* tend to be smaller. The best results are obtained for *Httpd* where both strategies are able to fully parse a fragment in more than 20 % of the cloned token sequences.

The values for C_{all} capture not only the largest tree but also smaller trees after a restart of the parser. Consequently, one should expect that the curve of C_{all} should always be above the corresponding curve for C_{max} for each strategy and system. Yet, that is not the case for *none* and *panic* for *Handbrake* and for *none* for *Httpd* as well as for all strategies for *Wget* (generally in the range of 0–10 % on the x axis). The reason for that is that we ignore trees smaller than the threshold of 10 tokens only for C_{all} and not C_{max} and these strategies created many such small trees.

Wget is different from the other systems also for the strategies *rpanic* and *suffix*. It contains an automatically generated parser, which consists of many repetitive comma-separated literals used to initialize parser tables and also several lists of case labels. All of them were reported by the clone detector as syntactically incomplete sequences. The following code snippet illustrates the problem: S = `,10,20,1,20,15,0,0,0`. The *panic* modes will skip all tokens in S until it reaches a synchronizing symbol (see Sections 3.3 and 3.4). Because there is no synchronizing symbol in S, both panic modes will skip the complete sequence and fail to parse it. Interestingly, the *suffix* strategy fails as well. The simplified C grammar rule for such initializer lists is as follows: $L \rightarrow L',' E|E$. With respect to this rule, S is syntactically incomplete from the left. Suffix parsers are designed for such situations (see Section 3.4). Hence, a full suffix parser should be able to pick up these cases successfully. The suffix parser we used, however, is a lightweight variant and fails in such situations, too (see Section 3.4). As a result, the *suffix* strategy yields syntax trees that consist of only two tokens, which are then ignored. About 75 % of the 118,979 cloned tokens of *Wget* parsed in our study stem from this generated parser code. These reasons explain why the coverage for *Wget* is worse. We did not exclude this type of automatically generated code because the code could have been written manually, too, (lists of literals are not uncommon in handwritten C code). This case hints at a general problem with incomplete repetitive structures.

5.4 Research Question RQ2

If a cloned token sequences is syntactically correct, all strategies should behave alike as there is no syntax error they would have to cope with. If the token sequence does not form a complete syntactic unit, the strategies deal with syntax error in different ways and may produce different syntax trees. We want a strategy to capture as much syntactic information as possible from a syntactically incorrect token sequence. RQ1 addressed this question from a more quantitative perspective in terms of the fraction of tokens derived from the syntax tree, but did not assess

whether the syntax tree itself is a correct derivation from the given sequence. One may argue that every such syntax tree derived from an Earley parser is correct by construction as the Earley parser will correctly apply the syntax rules. Yet, this tree might not be the tree expected by a human. A strategy may skip tokens after syntax errors (panic modes) or even at the beginning (suffix mode). They are heuristics to find a recovery point after which normal parsing is restarted. The starting point, however, decides what kind of tree will be recovered from then on. A suboptimal starting point may trigger suboptimal syntax analysis.

RQ2 addresses the issue of correctness and completeness of the strategies in terms of precision and recall. Precision and recall are measured against an Oracle O of expected syntax trees for given token sequences. Let E be the set of syntax trees extracted by applying a certain strategy S. Then precision of S is defined as the fraction of derived syntax trees derived by applying S that are correct: $|O \cap E|/|E|$. Recall is the fraction of expected syntax trees actually extracted: $|O \cap E|/|O|$.

Recall and precision require an oracle. In case the compilation unit is syntactically correct, we could create a syntax tree for the whole unit and compare the results of the partial syntax analysis to subtrees of the whole-unit syntax tree. That will not work in many cases, however, if the code contains preprocessor statements or is syntactically wrong. If there were an algorithm to create an oracle for syntactically incorrect code or code with preprocessor directives, that is, compute correct syntax trees for arbitrary token sequences, we would not need our strategies in the first place. Furthermore, we do not even know an operational definition of correct syntax trees for such arbitrary token sequences. For this reason, we use a human oracle. One of the authors, namely, Riemann determined the syntactic categories for a sample of the cloned token sequences manually. The syntactic category of a subsequence of tokens is the grammar non-terminal from which the subsequence is derived. In terms of syntax trees, it is the root node of a subtree. Specifically, we considered struct, union, and enum specifiers as well as labeled, compound, selection, iteration, and jump statements plus function definitions from the C grammar as relevant and meaningful syntactic constructs for such roots.

An extracted syntax tree is considered matched with a syntax tree from the oracle when both have the same syntactic category and the positions of their respective left-most and right-most tokens do not differ by more than 5 characters. The positions are not required to be identical but only close enough to mitigate possible errors the human oracle has made in deciding the positions manually. These types of errors may have been caused by imprecisely picking source positions with the mouse cursor.

Riemann validated all 67 cloned fragments of *Handbrake* and all 80 fragments of *Httpd*, but only 65 out of 149 for *Wget*. Sampling was required for *Wget* because we were unable to look at all the many cloned token sequences. If the population from which samples are to be drawn consists of subpopulations with varying characteristics, it is advantageous to sample each subpopulation (stratum) independently. Because our clones differ in size, we used proportional stratified sampling to draw the clones to be validated. Stratified sampling divides members of the population into homogeneous subgroups before sampling. That is, we sort all fragments by their size and partition this sequence in steps of 10 %, resulting in ten partitions. Per partition we randomly draw a number of fragments proportional to the relative number of fragments of that partition. That is, if there is a partition P that has double the number of fragments than a different partition P', the number of fragments drawn from P is double the number of fragments

drawn from P'.

Recall and precision gathered for the sample are shown in Figure 7 and Table 2. As we can see, precision is fairly high for all strategies, but the more elaborate strategies *suffix* and *rpanic* provide much better recall than *panic* and *none*, where *panic* is only slightly better than *none*. When we compare *suffix* and *rpanic*, we find that both are almost equally good for *Httpd*, while *suffix* has better precision than *rpanic* for the other two system but also worse recall.

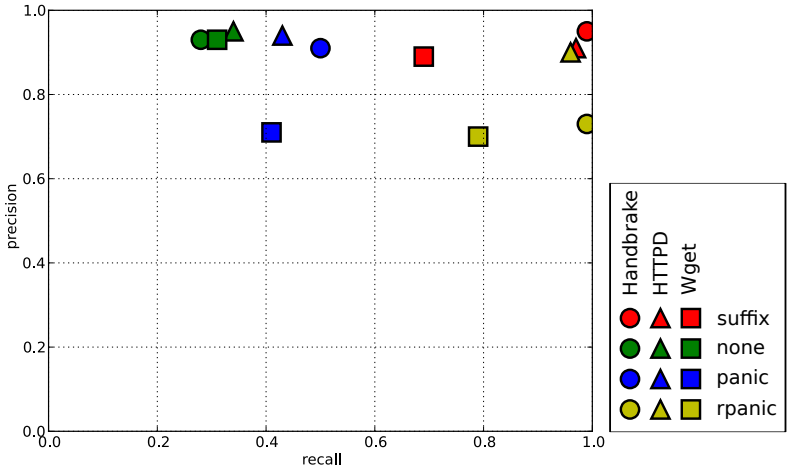


Figure 7: Recall and precision

	Handbrake		*Apache HTTPD*		*GNU Wget*	
	recall	precision	recall	precision	recall	precision
`suffix`	0.99	0.95	0.97	0.91	0.69	0.89
`none`	0.28	0.93	0.34	0.95	0.31	0.93
`panic`	0.50	0.91	0.43	0.94	0.41	0.71
`rpanic`	0.99	0.73	0.96	0.90	0.79	0.70

Table 2: *Recall and precision*

5.5 Research Question RQ3

Research question RQ3 investigates the costs for each strategy in terms of CPU time and memory consumption. We measure memory consumption by the number of Earley items built for each token sequence. The numbers were gathered on a Thinkpad X201 with an Intel Core i5 M560 at 2.67 GHz, 4 GB 1333 MHz DDR3 RAM, and an OCZ VERTEX2 SSD running Linux Mint 13.

Table 3 shows the results. As we can see for the relative numbers (normalized by the sequence length) all strategies require less time for *Wget* than for the other two systems. We need to point out that the numbers are relative to the total length of the cloned token sequence and not only to those actually processed by the strategies. In particular, strategy *none* skips many tokens in case of early errors in the input. This characteristics hit *Wget* particularly for the mentioned reason of

repetitive initializer lists.

It is remarkable that the actual parsing step requires more time than the creation of the syntax tree for *Handbrake* and *HTTPD*. In case of *Wget* it is opposite, except for *panic* and *rpanic*. The larger fragments in *Wget* can be made responsible for this behavior as the worst-case of syntax tree construction is exponential. This argument is backed up by the fact that this behavior cannot be observed for *panic* and *rpanic*, which both failed to parse the three largest fragments. Those three largest fragments had so many syntax errors that the program ran out of memory for the two *panic* modes. The size of these fragments were about 77,000, 1,750, and 1,000 tokens, respectively. All of them were found in a file that was automatically generated and, hence, they may be considered atypical. Nevertheless they hint at deficiencies for the two *panic* modes. Strategies *suffix* and *none* did not have any problem with these large clones.

Overall we find that the more elaborate strategies *suffix* and *rpanic* require generally also more time.

	Ørec	Øcst	Øitems
`suffix`	44.76	21.86	12341.87
`none`	14.24	4.45	4608.51
`panic`	16.30	5.07	6075.52
`rpanic`	69.25	9.44	12852.40
	rec	cst	items
`suffix`	0.28	0.14	77.32
`none`	0.09	0.03	28.87
`panic`	0.10	0.03	38.06
`rpanic`	0.43	0.06	80.52

(a) *Handbrake*

	Ørec	Øcst	Øitems
`suffix`	44.42	29.74	12335.57
`none`	10.26	3.92	3681.13
`panic`	12.53	4.13	5083.95
`rpanic`	61.53	5.99	10025.54
	rec	cst	items
`suffix`	0.34	0.23	95.45
`none`	0.08	0.03	28.48
`panic`	0.10	0.03	39.34
`rpanic`	0.48	0.05	77.57

(b) *Apache HTTPD*

	Ørec	Øcst	Øitems
`suffix`	40.30	42.01	13821.46
`none`	30.71	143.15	8651.69
`panic`	4.99	4.24	1774.17
`rpanic`	32.74	4.59	2670.64
	rec	cst	items
`suffix`	0.009	0.011	3.231
`none`	0.007	0.020	2.023
`panic`	0.001	0.001	0.414
`rpanic`	0.008	0.002	0.623

(c) *GNU Wget*

Table 3: *Resource consumption*
Ørec is the average time for parsing a token sequence and *Øcst* the average time for constructing the syntax tree in milliseconds; *Øitems* is the average number of Earley items. The numbers for these in the lower part of the tables are the same characteristics normalized by the respective length of a fragment in terms of number of tokens.

6 Conclusion

In this paper, we have explored ways to gather syntactic information from cloned token sequences. We used an Earley parser because it can cope with all context free grammars, even with ambiguous ones, which is a useful feature for parsing code snippets without their context. Because the fragments to be parsed stem from a token-based clone detector, we needed to equip the Earley parser with a robust fault-tolerance strategy. We compared three such strategies to a baseline that does not attempt to recover from a syntax error. The study shows that the more elaborate strategies of fault tolerance are capable to cover more tokens and to gather richer syntactic information than a simpler strategy and the baseline. The better information comes also at higher costs, but the costs for the more elaborate strategies are still low enough for practical

applications in general. Yet, we also experienced a few cases where the exponential worst-case nature of the algorithm hit the limits of available memory. These were cases of automatically generated sequences of repetitive code structures (e.g., lists of case labels or literals) that were syntactically incomplete. Such cases should already be filtered by a clone detector able to recognize repetitions [Kos12].

Given these promising results of principal feasibility, we want to use robust parsing in future research as an aide for syntactic filters to improve the precision of token-based detectors or to rank their results. It is also worthwhile to extend our study to larger code bases in order to encounter other language constructs that make C/C++ a difficult language to parse. Other researchers analyzing C/C++ code may benefit from this approach. For instance, Merlo et al. [MLPB13] had to preprocess a large telecommunication software to remove all preprocessor directives and then run an island parser to gather their metrics for clone detection.

Bibliography

[AH02] J. Aycock, N. R. Horspool. Practical Earley Parsing. *The Computer Journal* 45(6):620–630, Nov. 2002.

[AP72] A. Aho, T. Peterson. Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM Journal on Computing* 1(4):305–312, 1972.

[AU77] A. V. Aho, J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley Pub. Co., Reading, Massachusetts, 1977.

[Bak92] B. S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*. Pp. 49–57. Mar. 1992.

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Computer Society Transactions on Software Engineering* 33(9):577–591, Sept. 2007.

[BM01] I. D. Baxter, M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. Pp. 281–290. 2001.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. Clone Detection Using Abstract Syntax Trees. In Koshgoftaar and Bennett (eds.), *International Conference on Software Maintenance*. Pp. 368–378. IEEE Computer Society Press, 1998.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2):94–102, Feb. 1970.

[FKF08] R. Falke, R. Koschke, P. Frenzel. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Software Engineering* 13(6):601–643, 2008.

[GG12] P. Gazzillo, R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. *SIGPLAN Not.* 47(6):323–334, June 2012.

[GJ08] D. Grune, C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, New York, second edition, 2008.

[GK09] N. Göde, R. Koschke. Incremental Clone Detection. In *European Conference on Software Maintenance and Reengineering*. Pp. 219–228. IEEE Computer Society Press, 2009.

[HJHC10] B. Hummel, E. Juergens, L. Heinemann, M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *International Conference on Software Maintenance*. Pp. 1–9. IEEE Computer Society Press, 2010.

[KFF06] R. Koschke, R. Falke, P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Working Conference on Reverse Engineering*. Pp. 253–262. IEEE Computer Society Press, 2006.

[KGO11] C. Kästner, P. G. Giarrusso, K. Ostermann. Partial preprocessing C code for variability analysis. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. Pp. 127–136. ACM Press, 2011.

[KGR+11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Pp. 805–824. ACM Press, 2011.

[KKI02] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Computer Society Transactions on Software Engineering* 28(7):654–670, 2002.

[Kos12] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *European Conference on Software Maintenance and Reengineering*. Pp. 309–318. IEEE Computer Society Press, 2012.

[KRC11] I. Keivanloo, J. Rilling, P. Charland. SeClone – A Hybrid Approach to Internet-Scale Real-Time Code Clone Search. In *International Conference on Program Comprehension*. Pp. 223–224. IEEE Computer Society Press, 2011.

[LHMI07] S. Livieri, Y. Higo, M. Matushita, K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source. In *International Conference on Software Engineering*. Pp. 106–115. ACM Press, 2007.

[MLPB13] E. Merlo, T. Lavoie, P. Potvin, P. Busnel. Large scale multi-language clone analysis in a telecommunication industrial setting. In *International Workshop on Software Clones*. Pp. 69–75. IEEE Computer Society Press, 2013.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In *Working Conference on Reverse Engineering*. Pp. 13–22. 2001.

[NB96] M.-J. Nederhof, E. Bertsch. Linear-time suffix parsing for deterministic languages. *Journal of the ACM* 43(3):524–554, May 1996.