2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

A Pattern Search Method for Unpreprocessed C Programs based on Tokenized Syntax Trees

Atsushi Yoshida and Yoshinari Hachisu
Faculty of Science and Engineering, Nanzan University, Seto, Aichi, Japan
Email: {atsu, hachisu}@nanzan-u.ac.jp

***Abstract*—Pattern search of programs is a fundamental function for supporting programming. In this paper, we propose a search method for unpreprocessed programs, which are difficult to parse. Our parser directly parses them by rewriting token sequences, and allows minor errors in syntax trees. The search tool takes queries that are the same as the format of program fragments. By using the same parser for both queries and target programs, programmers have no need to describe the detail structures of syntax trees in queries. To support accurate search, we also show an alignment tool for branch directives, which converts undisciplined directives to discipline ones, and a reverse macro expansion tool, which integrates the use of macro calls. Finally, we present some experiments in which we have applied the tools to an open source application, and discuss how to improve our tools.**

***Keywords*-pattern search, unpreprocessed C program, rewrite rule, parser**

I. Introduction

Text pattern search is a fundamental function of tools for editing text documents. It is useful for checking how the words are used and finding the places of words to be changed. Program files are also text documents, and we often use this function to find elements. It is, however, not useful for finding program patterns, such as candidates of bad smells and typical uses of functions, and we need a method to search structural patterns in abstract syntax trees. In the case of the C language, we need to consider the existence of preprocessor directives, and pattern search for unpreprocessed programs is more difficult because the parsers need to accept alternatives of code introduced by branch directives, such as `#ifdef`, and syntactical errors, such as macro calls with statements in their arguments. Furthermore, describing search queries is exhausting because we need to understand the details of syntax trees including preprocessor directives.

In this paper, we propose a pattern search method for unpreprocessed programs. In our method, users describe patterns as code fragments, which may include preprocessor directives. The search tool parses both patterns and target programs by the same parser and generates syntax trees represented by token sequences. The syntax trees of the patterns are converted to the rewrite rules on token sequences that insert markers before and after the matched regions in token sequences representing syntax trees of target programs. After the rewrite rules are applied, the tool outputs code fragments generated from matched regions on token sequences of target programs (Figure 1). Figure 2 is an example pattern, which matches the pairs of continuous branch directives with same condition that have a statement or declaration between them. For improving readability, these pairs may be merged into one. Figure 3 is a candidate of merging that we found in FreeBSD 10.0 [1] by the pattern. The symbols "{<" and ">}" represent the markers of the matched region. Conditions of branch directives often represent crosscutting concerns, and the search tool is also useful for finding them. Figure 3 can be applied to find the common pairs of pre and post processes for specific function calls, which represent crosscutting concerns.

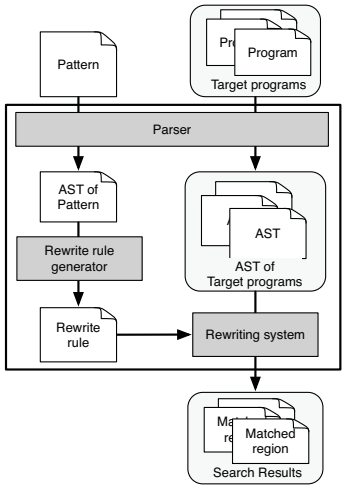

Fig. 1: The process of the pattern search tool.

```
#ifdef ${x:ID_MC}
$[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif

$[: ${:STMT} $| ${:DECL} $]

#ifdef ${x}
$[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif
```

Fig. 2: A pattern for continuous branch directives.

Our parser can directly parse unpreprocessed programs, but the syntax trees that it builds may contain incorrect subtrees. This is unavoidable for parsing real programs without normalizing them or applying preprocess for a specific configuration. Even if the syntax trees are incorrect, pattern search is able to succeed because patterns and target programs are parsed by the same parser, and the parser builds identical sub-

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.16

```
### src/sys/i386/include/intr_machdep.h, 152 ###
{<#ifdef SMP
void    intr_add_cpu(u_int cpu);
#endif
int     intr_add_handler(const char *name, int vector,
 driver_filter_t filter,
    driver_intr_t handler, void *arg, enum intr_type flags,
 void **cookiep);
#ifdef SMP
int     intr_bind(u_int vector, u_char cpu);
#endif>}
```

Fig. 3: A search result of the pattern Figure 2.

trees for both patterns and the matched regions in programs. We have developed the parser based on rewrite rules on token sequences. Rewrite rules gradually inserts syntactical information to sequences and preserve syntactical errors as is. The basic concept has been proposed as the island grammar [2], and an implementation for ANSI C has been proposed as Iterative Lexical Analysis (ILA) [3]. Our parser is specialized for unpreprocessed C programs. Though we may be able to solve some syntactical errors by analyzing how macros are expanded, we have not adopted it because we assume the situations that not all definitions are always available, such as being under development.

Our method has two problems. One is the existence of preprocessor directives. Users need to consider all possible directives while describing patterns, but it is difficult to do actually. The other is unintegrated macro use. If programs contain the expressions that can be replaced with macro calls, users need to consider both cases about the use of macro calls. Since these problems are difficult to solve systematically, we propose two support tools for normalizing target programs. One is an alignment tool of undisciplined directives [4], which do not exist on the border of statements, declarations and function definitions. The alignment tool moves undisciplined directives to the borders while preserving the behaviors. By aligning directives, users have no need to consider the directives inside expressions. The other is a reverse macro expansion tool, which searches the regions matched with the replacement of macro definitions and replaces them with macro calls. This tool can also contribute to the improvement of maintainability.

In the following sections, we show the design of the parser in Section II, and the design of the pattern search tool in Section III. We also explain the alignment tool of branch directives and the reverse macro expansion tool in Section IV. In Section V, we present the evaluation of the parser and tools and discuss problems of the tools. Our tool set, which we call TEBA, is built on Perl, and we use the terms and the symbols of Perl in this paper. The tool set is available at `http://tebasaki.jp/src`.

II. Tokenized Syntax Tree and the Parser

For representing syntax trees containing syntactical errors, we use token sequences. In a token sequence, a non-terminal element is represented by pairs of virtual tokens, which mean the beginning and the end of elements. This style is the same

TABLE I: The types of virtual tokens.

UNIT_BEGIN, UNIT_END	the unit of the result
B_DIRE, E_DIRE	directives
B_MCB, E_MCB	the replacements of macro definitions
B_FUNC, E_FUNC	function definitions
B_DE, E_DE	declarations
B_ST, E_ST	statements
B_TD, E_TD	typedef declarations
B_SCT, E_SCT	struct definitions
B_EN, E_EN	enum definitions
B_UN, E_UN	union definitions
B_CP, E_CP	non-statement blocks { ... }
B_FD, E_FD	function declarators
B_FR, E_FR	function references
B_CAST, E_CAST	cast operators
B_P, E_P	expressions
B_LB, E_LB	labels

with the XML format, but we have not adopted it because the rewrite rules are not limited to XML operations. Each token has a type and a text. Type means the class of tokens, such as identifier, literals, and operators. We defined 51 types as terminal elements and 16 pairs of virtual tokens as non-terminal ones. Table I is the list of the virtual tokens. Text is the string representation of a token, and a virtual token has an empty string. White spaces and comments are also tokens. By concatenating the texts of all the tokens, we get the original text of the parsed program. Figure 5 is an example token sequence generated from Figure 4. To save space in this paper, all the tokens of white spaces are deleted in the figure.

Token sequences themselves are not comprehensive, and we present a mixing style of a program and tokens in the right of Figure 4. The marks surrounding brackets, such as "[B_ST]" and "[E_ST]," are virtual tokens, which represent non-terminal elements. The prefixes "B_" and "E_" of type names mean the beginning and the end of elements. For improving readability, the virtual tokens "B_P" and "E_P," which mean expression level elements, are represented by curly braces: "{" and "}." The blue marks following identifiers, such as ":ID_TP," mean their classes: type, variable/function, tag, member, and macro.

The parser parses a token sequence by applying rewrite rules. Rewrite rules add virtual tokens for identified elements. They also rename types of tokens to distinguish semantically different tokens whose texts are the same, such as colons, which are used for labels, ternary operators, and declarators in structs. For capturing non-terminal elements, the pairs of braces and virtual tokens have the same unique pair ids. In Figure 5, the symbols prefixed with "#" represent pair ids.

A parser based on rewrite rules has been proposed as Iterative Lexical Analysis (ILA) [3], which supports ANSI C syntax. Because the basic concept of our parser is the same as ILA, we mainly explain how our parser treats unpreprocessed programs. The merits of rewrite rule-based parsers are: (1) it is possible to parse parts of programs, such as the replacements in macro definitions, and (2) it is easy to complement tokens in an ad hoc way. The basic strategy for parsing unpreprocessed

```
#ifdef ABS_DOUBLE
double abs(double v)
#else
int abs(int v)
#endif
{
  if (v > 0) {
    return v;
  } else {
    return -v;
  }
}
```

```
[B_DIRE]#ifdef {ABS_DOUBLE:ID_MC}[E_DIRE]
[B_FUNC][B_FD]double:ID_TP {[B_FR]{abs:ID_VF}(double:ID_TP {v:ID_VF})[E_FR]}
[B_DIRE]#else[E_DIRE]
int:ID_TP {[B_FR]{abs:ID_VF}(int:ID_TP {v:ID_VF})[E_FR]}[E_FD]
[B_DIRE]#endif[E_DIRE]
[B_ST]{
  [B_ST]if {({v:ID_VF} > {0})} [B_ST]{
    [B_ST]return {v:ID_VF};[E_ST]
  }[E_ST] else [B_ST]{
    [B_ST]return {-{v:ID_VF}};[E_ST]
  }[E_ST][E_ST]
}[E_ST][E_FUNC]
```

Fig. 4: A sample program (left) and a mixing style of the program and tokens (right).

```
UNIT_BEGIN <>
B_DIRE #E0001 <>
PRE_TOP <#>
PRE_DIR <ifdef>
B_P #E0002 <>
ID_MC <ABS_DOUBLE>
E_P #E0002 <>
E_DIRE #E0001 <>
B_FUNC #E0003 <>
B_FD #E0004 <>
ID_TP <double>
B_P #E0005 <>
B_FR #E0006 <>
B_P #E0007 <>
ID_VF <abs>
E_P #E0007 <>
P_L #B0001 <(>
ID_TP <double>
B_P #E0008 <>
ID_VF <v>
E_P #E0008 <>
P_R #B0001 <)>
E_FR #E0006 <>
E_P #E0005 <>
B_DIRE #E0009 <>
PRE_TOP <#>
PRE_DIR <else>
E_DIRE #E0009 <>
ID_TP <int>
B_P #E0010 <>
B_FR #E0011 <>
B_P #E0012 <>
ID_VF <abs>
E_P #E0012 <>
P_L #B0002 <(>
ID_TP <int>
B_P #E0013 <>
ID_VF <v>
E_P #E0013 <>
P_R #B0002 <)>
E_FR #E0011 <>
E_P #E0010 <>
E_FD #E0004 <>
B_DIRE #E0014 <>
PRE_TOP <#>
PRE_DIR <endif>
E_DIRE #E0014 <>
B_ST #E0015 <>
C_L #B0003 <{>
B_ST #E0016 <>
CT_IF <if>
B_P #E0017 <>
P_L #B0004 <(>
B_P #E0018 <>
B_P #E0019 <>
ID_VF <v>
E_P #E0019 <>
OP <>>
B_P #E0020 <>
LIN <0>
E_P #E0020 <>
E_P #E0018 <>
P_R #B0004 <)>
E_P #E0017 <>
B_ST #E0021 <>
C_L #B0005 <{>
B_ST #E0022 <>
RE_JP <return>
B_P #E0023 <>
ID_VF <v>
E_P #E0023 <>
SC <;>
E_ST #E0022 <>
C_R #B0005 <}>
E_ST #E0021 <>
CT_EL <else>
B_ST #E0024 <>
C_L #B0006 <{>
B_ST #E0025 <>
RE_JP <return>
B_P #E0026 <>
OP_U <->
B_P #E0027 <>
ID_VF <v>
E_P #E0027 <>
E_P #E0026 <>
SC <;>
E_ST #E0025 <>
C_R #B0006 <}>
E_ST #E0024 <>
E_ST #E0016 <>
C_R #B0003 <}>
E_ST #E0015 <>
E_FUNC #E0003 <>
UNIT_END <>
```

Format of tokens: *type_name pair_id?* '<' *text* '>'

Fig. 5: A token sequence generated from Figure 4.

TABLE II: The obstacles for parsing real programs.

(A1) **Obstacle:** Unbalanced brackets:
```
#define LOOP_BEGIN while (0) {
```
Solution: Add virtual brackets for completing pairs.

(A2) **Obstacle:** Statements not followed by semicolons:
```
INITIALIZE(a)
INITIALIZE(b)
```
Solution: Add virtual semicolons at the ends of statements.

(A3) **Obstacle:** Natural language sentences that forcefully cause errors:
```
#ifndef HAS_FUNC1
This system requires FUNC1.
#endif
```
Solution: Add virtual semicolons at the ends, and concatenate tokens in the sentences.

(A4) **Obstacle:** Types and attributes masked by macro calls:
```
TYPE(x) func(ARG) { ... }
```
Solution: Weaken the conditions for capturing function definitions.

(A5) **Obstacle:** Multiple else nodes:
```
      if (c) ...
#ifdef COND1
      else { ... }
#elif COND2
      else { ... }
```
Solution: Allow independent else statements.

(A6) **Obstacle:** Parsing the inside of macro definitions:
```
#define VALID_TEST(x) if (!cond(x)) prt_err(x)
/* Macros are not always expressions. */
```
Solution: Add virtual tokens for semicolons and statements.

(A7) **Obstacle:** Lack of definitions of identifiers:
```
a = (X)(y); /* cast or function call? */
```
Solution: Guess the class of identifiers, such as type, variable, tag and member, from their contexts.

(A8) **Obstacle:** Statements in arguments of macro calls:
```
REPEAT(NUM, ++i; a[i] = f(i); b[i] = g(i); );
```
Solution: Find function calls which the parser failed to parse, and add virtual tokens for statements.

programs is to ignore directives. The parser parses directives at first and, in the following processes, treats them as white spaces while preserving them. In general, directives are placed on the borders of statements and declarations, and this strategy often works well. There are, however, some obstacles in practice. Table II shows the obstacles we found in Coreutils 8.22, the examples, and our solutions. Our solutions depend on Coreutils 8.22, and we need to analyze other programs for making them more general. Coreutils, however, covers typical styles of unpreprocessed programs, because it includes various kinds of programs: commands, system libraries, and test programs.

The difficulties of parsing real unpreprocessed programs are categorized to two problems. One is syntactical error and the other is a lack of definitions of identifiers. The obstacles except (A7) are typical syntactical errors, and we have relaxed the parsing rules and added the rules for complementing lacking tokens, such as semicolons and brackets, based on heuristics. The complemented tokens have types and empty texts. Obstacle (A7) affects the structures of syntax trees. Especially, distinguishing type name and variable/function name is important. Since it is difficult to make symbol tables exactly for unpreprocessed programs, we assumed same names are used for the same class. The parser checks the contexts for each occurrence of identifiers and guesses the classes for each same name identifier by using heuristics. This assumption is valid for most programs, though there are rare exceptions [1].

The parser consists of eight filters. Figure 6 shows the filters and the obstacles that they solve. The filters except P1 and P3 are mainly described as rewrite rules and small auxiliary filters. The auxiliary filters are introduced because

[1]For example, in contrib/oid2name/oid2name.c of Postgresql 9.3.4 [5], the formal argument `eary` is declared with type `eary` too

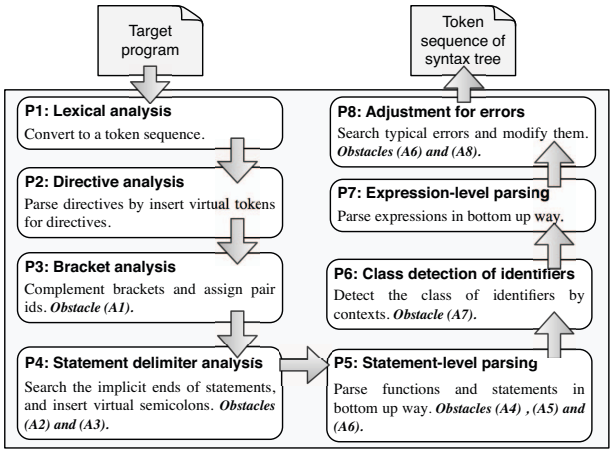

Fig. 6: The parsing process of the parser.

```
@OP03 => "OP\s+<[\*\/%]>\n"

# '{': _B_X, '}': _E_X, '(': B_P, ')': E_P, '<': _B_OP03, '>': _E_OP03
# {a} * {b} * {c} => {a} <* (b)> * (c) => {a} <* (b)> <* (c)>
{ $op:OP03 $sp:SP $bx#1:_B_X $any:ANYEXPR $ex#1:_E_X }
=>> { ''#1:_B_OP03 $op $sp $bx:B_P $any $ex:E_P ''#1:_E_OP03 }

# {a} <* (b)> <* (c)> => {(a) * (b)} <* (c)> => {((a) * (b)) * (c)}
{ $bx1#1:_B_X '(?>':X $any1:ANYEXPR $ex1#1:_E_X ')':X $sp1:SP
  $#2:_B_OP03 $op:OP03 $sp2:SP $any2:ANYEXPR $#2:_E_OP03 }
=>> { ''#1:_B_X $bx1:B_P $any1 $ex1:E_P $sp1 $op $sp2 $any2 ''#1:_E_X }
```

Fig. 7: The rewrite rules for parsing expressions combined with the operators, "/," "*," and "%" in the filter P7.

their features are impossible to be implemented by rewrite rules. We have also replaced some rewrite rules with the auxiliary filters due to insufficient performance. The rewriting system of token sequences, which we have built, translates the rules to Perl scripts of string substitution using extended regular expressions and rewrites token sequences as a text.

A rewrite rule searches a token subsequence, and then inserts virtual tokens or changes types of tokens. Figure 7 is an example of the rules in the filter P7. This rule set inserts virtual tokens _B_X and _E_X for the expressions combined by operators "/," "*," and "%," and changes the tokens of _B_X and _E_X to B_P and E_P. The virtual tokens _B_X and _E_X are temporary tokens, which are temporarily used as markers and removed by the end of parsing. In this case, the virtual tokens _B_X and _E_X represent the last identified expressions. In the filter P7, expressions are identified in the order of precedence of operators from the bottom-up. In Figure 7, the lines beginning with # are comments, and the line beginning with @ defines a special type as a regular expression in string for tokens.

A rule is of the form: "{ *pattern* } => { *replacement* }." The token subsequence matched with the *pattern* is replaced with the *replacement*. We can use the operator =>> for recursive rewriting, instead of =>. The tokens prefixed with "$"s are variables for matched tokens. *Pattern* consists of variables followed by type names, which match any tokens of the types. The texts of tokens can be specified too. The

alternations: $[: *alternative1* $| *alternative2* $]
Ex. $[: return ${:EXPR}; $| exit(${:EXPR}); $]

repeats: $[: *elements* $]*quantifier*
(*quantifier* is one of *, +, ?, *?, or +?.)
Ex. $[: ${:STMT} $]*

concatenation: *token1* $## *token2*
(The concatenation of *token1* and *tokens2* match an identifier)
Ex. prefix_ $## ${x:ID_VAR}

stringification: $# *token*
(*token* matches the contents of a string literal)
Ex. $# ${name:ID_VAR}

predefined pattern: ${ *name* : *type* }
(*Name* can be ommited. *Type* is all types of tokens and predefined type: DECL, STMT, DIRE, FUNCDEF, EXPR, ARGLIST.)
Ex. ${:DECL}, ${x:STMT}

pattern reference: ${ *name* }
(The reference matches a token that predefined pattern of the same *name* matches.)
Ex. ${x}

region specifier: ${%begin}, ${%end}
(The parser parses whole of a pattern, but generates token patterns between the specifiers.)
Ex. ${:DECL}

Fig. 8: The meta tokens for queries.

ids, such as "#1" and "#2," following the variable names means that the tokens have the same pair ids. By specifying ids, *pattern* matches the correct pairs of virtual tokens, which represent non-terminal elements. *Pattern* can include named groups and repeats as with regular expression. In *replacement*, variables are referred to by name. The variables followed by type names change the types of tokens. New tokens can be inserted by describing the tokens with texts and type names, such as "'':_B_X." For the optimization of performance, *pattern* can include the special pairs of tokens "'(?>':X" and "')':X," which suppress backtracking while matching. They are the same with independent subexpression of Perl.

III. Pattern search on tokenized syntax trees

A typical technique of pattern search on syntax trees is to describe patterns of structures of trees, such as using XQuery. Programmers, however, need to know exact structures of trees for describing queries, and sometimes need extra efforts, such as parsing small sample programs, to understand them. In the case of unpreprocessed programs, the structures are more complicated to understand due to the existence directives and macro uses. Our approach is to describe a code fragment as a query, as shown in Figure 2, and to generate patterns for syntax trees from it automatically. The format of the queries is basically the same with the C, and is extended with meta tokens in Figure 8. In Figure 2, each branch directive contains a sub-pattern matching a list of statements, declarations, and directives. A sub-pattern between two branch directives matches a statement or declaration.

Patterns are parsed by the parser in Section II, which is extended for the meta tokens. The meta tokens are defined as lexical tokens, and predefined patterns are treated as syntactical elements. The symbols for alternations and repeats are preserved, but ignored as white spaces in the parsing process

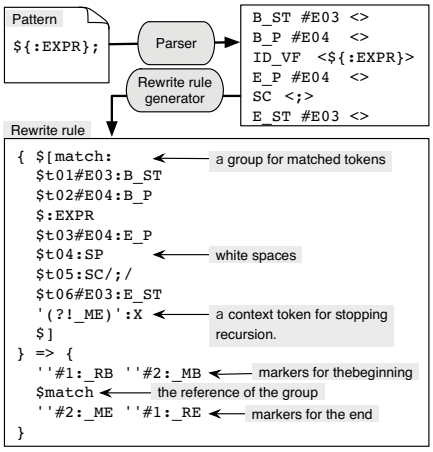

Fig. 9: An example of rule generation.

A or B:
`preg.pl -ta` *A file.c* `|` `preg.pl -T` *B*
(Searches regions matched with patterns A or B in *file.c*)

A and B:
`preg.pl -t` *A file.c* `|` `preg.pl -uT` *B*
(Searches pattern *B* in the regions matchend with pattern *A*. *B* should be specialized one of *A*.)

A and not B:
`preg.pl -t` *A file.c* `|` `preg.pl -rT` *B*
(Removes the regions matched with pattern *B* from the regions matched with pattern *A*. *B* should be specialized one of *A*.)

A in B
`preg.pl -t` *B file.c* `|` `preg.pl -ucT` *A*
(Searches pattern *A* inside the regions matched with pattern *B*.)

Options:
`-t`: outputs token sequence instead of program text.
`-T`: reads an input as token sequence instead of program text.
`-a`: outputs the whole of target programs, including matched regions.
`-u`: removes markers from the input.
`-r`: removes matched regions from the input.
`-c`: searches between markers.

Fig. 10: Fundamental combinations of queries.

as with preprocessor directives. After parsing patterns, the tool generates rewrite rules internally. Figure 9 shows an example of parsed tokens and generated rewrite rules. The *pattern* of the rewrite rule contains the variables corresponding to the parsed tokens and three kinds of symbols: group "`match`" for capturing the whole of tokens, variables for arbitrary white spaces, and a context token for stopping recursive matching. The *replacement* contains the group reference and two pairs of marker tokens representing matched token subsequences. One of the pairs is to be used to represent the context regions for output, mentioned later. After applying generated rewrite rules to the target programs repeatedly until marking all matched regions, the tool outputs the marked regions.

The expressive power of our patterns is limited. If we introduce more expressive meta tokens, we must face the same difficulties of parsing unpreprocessed programs. Briefly, two different syntaxes for the C and meta tokens are mixed in patterns. We can, however, combine queries without introducing special notations. Our tool is implemented as a command-line tool, and we can combine queries by connecting tool invocations with pipes. Figure 10 shows fundamental combinations and the options of the command. In the figure, `preg.pl` is the command name. The patterns can be specified directly as an argument, or given by files with option "`-f`." In the latter case, we can specify multiple patterns in a file. For example, when we want to count branch directives used inside functions as a metric, we can gather the directives by Figure 11. The pattern file *ifdef.pt* is Figure12. This file contains three patterns separated by "`%%---`", and the tool finds all types of branch directives.

```
preg.pl -ta '${:FUNCDEF}' sample.c | preg.pl -Tuc -f ifdef.pt
```

Fig. 11: A command for searching directives inside functions.

```
#ifdef ${:EXPR}
$[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif

%%---

#ifndef ${:EXPR}
$[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif

%%---

#if ${:EXPR}
$[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif
```

Fig. 12: *ifdef.pt*: a pattern for branch directives.

Programmers often want to see the contexts including the matched regions. The search tool has three options for context regions: (1) number of lines, (2) number of parent blocks, and (3) number of borders. Type (1) is the same way with grep, which is a traditional line-based search tools. Type (2) is to specify how many elements it goes up to the root of the tree. Type (3) is similar to (1), but it counts the number of borders of statements, declarations, function definitions, and directives while moving on the token sequence in the both directions. For example, Figure 13 is a pattern to find function bodies that consist of a single branch directive. This pattern is useful for finding the refactoring candidates when we select a coding style in which functions are defined separately for each compile condition. Figure 14 is a search result in FreeBSD 10.0, and the context option is to display a parent block, which is the whole function in this case. In Figure 13, function `f()` is defined, but the tool searches the function bodies and not function `f()` itself. Though the tool parses the whole of the pattern, it generates a rewrite rule using only tokens between `${%begin}` and `${%end}`. The reason for introducing this style is the difficulty of defining a general pattern of function definitions. A function definition may have macro uses in its return type, such as "`__inline`" in Figure 14, or a branch directive in its head, as shown in Figure 4. Figure 13 can match only function bodies and not other compound statements because a virtual token for the end of functions, `E_FUNC`, exists before `${%end}`. This technique, which is to describe

```
int f()
${%begin}
{
#ifdef ${:EXPR}
 $[: ${:STMT} $| ${:DECL} $| ${:DIRE} $]+
#endif
}
${%end}
```

Fig. 13: A pattern for a function body covered by a branch directive.

```
### src/sys/i386/include/cpufunc.h, 132 ###
static __inline void
enable_intr(void)
{<{
#ifdef XEN
        xen_sti();
#else
        __asm __volatile("sti");
#endif
}>}
```

Fig. 14: A search result of Figure 13 with a parent block.

concrete codes as contexts, is limited to apply, but convenient.

IV. Support Tools for Accurate Search

When we describe patterns for unpreprocessed programs, we always consider the existences of directives. Especially undisciplined directives are problematic. If they exist, we need to consider the directives inside expressions, and the possible positions they appear are too many to describe patterns. Undisciplined directives may also introduce structural errors, such as unpaired brackets and multiple else blocks in a if statement. To remove undisciplined directives, we have developed an alignment tool for directives. The tool moves the branch directives to the borders of four types of elements: statements, declarations, function definitions, and directives. These elements are units of lists, and the branch directives that have them as alternatives do not break the structures of syntax trees.

Macro calls are also problematic for search. The queries become complex and inaccurate when a macro is used and there still exist the parts that can be replaced with the macro calls in a program. We need to describe queries while considering both used and unused cases. This situation is, however, a bad smell for refactoring. We should integrate macro uses before search. To support the refactoring, we propose a reverse macro expansion tool, which replaces the parts matched with the replacement of a macro definition with the macro calls.

Macro calls have another problem that they make the parser fail to parse correctly. For example, they may hide parts of control statements, and the parser fails to parse them as statements. It is difficult to improve this problem systematically, and we need to refactor the programs for the parser-friendly style. The reverse macro expansion tool can support it; after extracting problematic macro calls, we can apply the reverse macro expansion tool to the extracted parts with the new macro definition that does not break the syntax. The forward macro expansion is almost the same with the reverse one. We have implemented a tool for the forward expansion too.

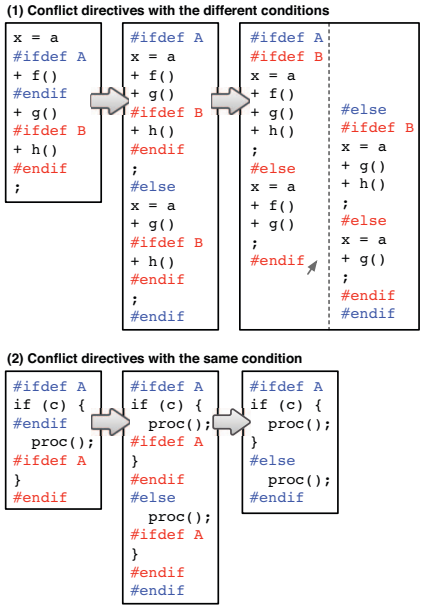

Fig. 15: The alignment processes for conflicting directives.

A. The alignment tool for branch directives

The alignment tool moves branch directives to the borders by copying texts between the directives and the borders into the each branch. The target borders are determined by the smallest elements or lists of them that contain the directives. In this section, we use a term "branch" as the parts between two paired directives of `#ifdef`, `#else`, `#endif`. For simplifying the explanation, we use these three kinds of directives in the following, though our implementation supports all kinds.

If two directives conflict, they need to be moved in order. The conflict means that one directive exists in a branch of the other directive or that a common text part exists that is to be moved into both branches. The tool moves the conflicting directive on the former line, and then moves the other one. Figure 15 shows two typical cases of conflicts and how they are aligned. If the conditions are contradicted, such as the combination of `defined(A)` and `!defined(A)`, the tool deletes the branches. In the case (2), the tool deletes two pairs of directives. If the contradicted conditions remain, the unbalanced brackets may occur like the center one in the case (2), and the parser fails to parse correctly.

The correctness of the borders is important for this tool. Though our parser allows incorrectness of syntax trees, the border positions are almost correct. The borders can be identified by the tokens that are delimiters of statements and declarations. Only the implicit delimiters of statements and declarations, which are not followed by semicolons like (A2) in Table II, may be incorrect.

The alignment tool has an advantage that it makes the syntax tree structurally simple, and the aligned programs can be analyzed by existing analyzers that accept only disciplined directives. We show the aligned program of Figure 16 in Figure 17,

```
#ifdef ABS_T
ABS_T abs_##ABS_T(ABS_T v) {
#else
int abs_int(int v) {
#endif
#ifdef ABS_T
  if (ABS_T_CMP(v))
#else
  if (v >= 0)
#endif
    return v;
  else
    return
#ifdef ABS_T
      ABS_T_REV(v);
#else
      -v;
#endif
}
```

Fig. 16: A program including undisciplined directives.

```
#ifdef ABS_T
ABS_T abs_##ABS_T(ABS_T v) {
  if (ABS_T_CMP(v))
    return v;
  else
    return
      ABS_T_REV(v);
}
#else
int abs_int(int v) {
  if (v >= 0)
    return v;
  else
    return
      -v;
}
#endif
```

Fig. 17: An aligned program of Figure 16.

in which two functions are defined separately. The drawbacks are that aligned programs may become exponentially large and that users need to understand the correspondences between the original program and the aligned one. In our implementation, we prepared options for selecting target directives: (1) all, (2) directives with else branches, and (3) marked ones by users. Option (2) and (3) are for reducing the size of the aligned programs. We have prepared option (2) because the directives without else branch do not break syntax even if the parser ignores directives in the parsing process. The marks for option (3) are the specific comments at the end of directive lines.

A typical use of option (3) is to align directives in only blocks that include candidates of search results. We can find candidate blocks by searching key elements in patterns with an option for displaying contexts. The alignment tool has an option to add target marks to the directives in context regions that the search tool outputs. After searching key elements and aligning blocks, we can search patterns without considering the existence of directives.

B. Reverse macro expansion

The reverse macro expansion tool searches the parts matched with the replacement of a macro definition, and replaces them with the macro calls. Ideally, by applying all macro definitions in programs, we can integrate macro uses. In practice, the macro definitions of constant values may be applied to unexpected places, and we need to manually select the definitions that we want to apply.

(Q1) The query for searching macro definitions with arguments:

```
#define ${name:ID_MC}(${args:ARGLIST}) ${body:ANY}
```

(Q2) The query for searching macro definitions without arguments:

```
#define ${name:ID_MC} ${body:ANY}
```

(Q3) The pattern for removing do-while(0) in macro definitions without arguments:

```
%before
#define ${name:ID_MC} do ${body:STMT} while(0)
%after
#define ${name} ${body}
%end
```

Fig. 18: The queries used in the reverse macro expansion tool.

Definition: `#define ADD(a, b) ((a) + (b))`

Reverse Pattern:	**Forward Pattern:**
`%ex`	`%ex`
`%before`	`%before`
`${a:EXPR} + ${b:EXPR}`	`ADD(${a:EXPR}, ${b:EXPR})`
`%after`	`%after`
`ADD(${a}, ${b})`	`((${a}) + (${b}))`
`%end`	`%end`

Fig. 19: Macro expansion patterns generated in the tool.

The expansion tool is an extension of the search tool. It searches the macro definitions by the queries, Q1 and Q2 in Figure 18, and generates transformation patterns, which are extensions of the search queries. While the search tool inserts the markers around the matched regions by using rewrite rules, the expansion tool replaces the matched regions with the tokens of macro calls. Before converting from macro definitions to transformation patterns, the tool analyzes macro arguments and replaces them with predefined patterns. Figure 19 shows the examples of internal transformation patterns that the tool generates from a macro definition. In the "Reverse Pattern," the query following `%before` is generated from the replacement of macro definition. The format is the same with the search tool, except the predefined patterns have names. The expression following `%after` is a macro call template, which contains the references of the named predefined patterns. The references are to be replaced with the tokens matched by the predefined patterns. `%ex` means that the query is an expression and not a statement.

To avoid unexpected results of expansions, macro definitions often contain extra parentheses and control statements, "`do ... while(0)`," surrounding the macro replacements. These extra descriptions are not used at the places to be replaced. The expansion tool normalizes macro definitions by the transformation patterns that remove the extra descriptions, like Q3 in Figure 18. In the "Reverse Pattern" of Figure 19, the extra parentheses of the arguments are removed.

When we improve the correctness of a syntax tree by refactoring macro definitions, we need to expand macro calls before the reverse expansion. The forward expansion of macro calls is easy to be implemented, because the mechanism is the same with the reverse replacement. Figure 19 also shows an internal pattern of the forward expansion. In the pattern, the extra parentheses remain because whether they can be removed

depends on the contexts of macro calls.

V. Evaluation and discussion

A. Evaluation of the parser

Our search tool parses both target programs and queries using the same parser. To evaluate the parser, we have applied it to all *.c and *.h files in Coreutils 8.22, 1169 files in total. We have examined the results from three points: (1) consistency, (2) uncombined expressions, and (3) type identification. Our parser allows incorrect syntax trees, and it is difficult to check the correctness for all programs. We had tested the parser with small samples and several files in Coreutils, and we have found these points can be applied for systematic evaluations.

(1) The evaluation of consistency means to check existence of unpaired brackets, unpaired virtual tokens, and temporary tokens. The existence of unpaired brackets and unpaired virtual tokens means that the parser fails to build a syntax tree. Unpaired brackets occur by failing to insert virtual brackets in the filter P3 in Figure 6. Virtual tokens are not always inserted by pairs, especially in the P5 filters. Temporary tokens are virtual tokens that are introduced as temporary markers in rule sets. For distinguishing the temporary tokens, we use the names beginning with "_" for them, such as `_B_X` in Figure 7. The existence of temporary tokens means the rules are not applied as expected. We have processed all 1169 files in Coreutils, and found all results are consistent.

(2) An uncombined expression means two successive expressions that are not combined with an operator. This situation occurs by incorrect parsing rules. For example, if an occurrence of a binary operator is identified as a unary operator, this situation occurs. Our parser tries to find statements not followed by semicolons. If it misses to find, uncombined expressions may occur. There are, however, cases that two successive expressions are valid, and we need to check them manually. For example, if a declaration has two macro calls for types and attributes, these two macro calls are not combined by an operator and look like an uncombined expression. We have found 1536 uncombined expressions in 196 files (16.8%); the ones caused by type identification errors are 25 in 12 files. The type identification error means the error that identifiers representing type are analyzed as other kinds of identifiers, such as variables. The ones caused by string concatenations, where all strings are hidden by macro calls, are 207 in five files. These 232 errors are difficult to fix because the parser does not use the information of macro expansion. The others are caused by macro calls representing types and attributes in declarations. Though they are correct results, they may cause other errors such as incorrect identification of types and declarations.

(3) We have checked the types are identified correctly, because the result (2) suggests the syntax trees may contain incorrect identification of types. Since it is difficult to know all right classes of identifiers, we have checked the identifiers whose name ends by "`_t`", which is a typical naming convention for type name. We distinguished same name identifiers in different files. We have found 1614 identifiers and confirmed that 1428 (88.5%) identifiers are identified as type. Most of the types that are not identified appear only in cast operators and no declaration using them exists.

file:	`lib/getopt.in.h`
directive:	`# define __GETOPT_CONCAT(x, y) x ## y`
applied to:	499 files (117701 places)
file:	`lib/stdarg.in.h`
directive:	`# define va_copy(a,b) ((a) = (b))`
applied to:	414 files (18662 places)
file:	`src/system.h`
directive:	`#define _(msgid) gettext (msgid)`
applied to:	6 files (55 places)
file:	`lib/filename.h`
directive:	`# define ISSLASH(C) ((C) == '/')`
applied to:	33 files (65 places)

Fig. 20: Macro definitions whose reverse expansion is applied.

B. Evaluation of the support tools

For preparing test queries for the search tool, we have generated queries for reverse macro expansion from 1851 macro definitions extracted from src and lib in Coreutils 8.22. These definitions do not include duplications and the ones that define constant values. The reverse macro expansion tool is an extension of the search tool, and the difference is the way to rewrite the matching regions. The evaluation of the expansion tool can be understood as the one for the search tool.

For confirming that the queries can find program parts, we have applied the queries to 521 files, which are all *.c files in src and lib directories. As a result, 374 queries have been applied to 504 files, 6.86 million places in total. This result does not include the self-applications of macro definitions, which means the replacement of a macro definition is replaced with the macro call of itself. There are many generic macros, such as `__GETOPT_CONCAT` and `va_copy(a,b)` in Figure 20, and reverse expansion of these macros is not useful. We have found examples of unintegrated macro uses. The typical examples are macro `_()` and `ISSLASH()` in Figure 20. These macros are used in many files, but several files contain the parts that can be replaced. We have also found another interesting example; a macro `POS_AFTER_TAB` is defined in the file `src/pr.c`, but an expression that can be replaced still exists in the same file (Figure 21).

We have applied reverse expansion for each macro definition to the file where the macro is defined. In this case, we have set the options of disabling the normalization of queries and of allowing the self-application of macro definitions. We expected all the replacements in macro definitions are replaced with the macro calls of themselves. The result is that 1885 of 2045 (90.7%) macros are replaced. We have confirmed that complicated definitions are replaced, such as `DEFINE_SORT_FUNCTIONS` in src/ls.c, which includes function definitions. The macros that are not replaced are 162. The reason of the failure is that the classes of identifiers are different. Because the macro definitions have not enough information to guess the class of identifiers, the classes of identifiers are different from the ones in the target programs.

A target macro definition in src/pr.c:

```
/* The horizontal position we'll be at after printing a tab character
   of width c_ from the position h_. */
#define POS_AFTER_TAB(c_, h_) ((h_) + TAB_WIDTH (c_, h_))
```

A result of reverse macro expansion:

```
--- coreutils-8.22/src/pr.c 2013-12-04 09:48:30.000000000 -0500
+++ pr.x 2014-03-20 10:30:36.000000000 -0400
@@ -1275,8 +1275,7 @@

   /* Estimate chars_per_text without any margin and keep it constant. */
   if (number_separator == '\t')
-    number_width = (chars_per_number
-            + TAB_WIDTH (chars_per_default_tab, chars_per_number));
+    number_width = (POS_AFTER_TAB(chars_per_default_tab,chars_per_number));
   else
     number_width = chars_per_number + 1;
```

Fig. 21: A result of reverse macro expansion.

We discuss how to solve this problem in Section V-D.

C. Evaluation of the alignment tool

For confirming that the tool aligns the directives without breaking the semantics, we have applied the alignment tool to all files in Coreutils 8.22. The transformation may exponentially increase the sizes of files, and the tool has failed for five files[2] due to lack of memory. For checking the correctness of the modified programs, we have compiled them and run the test programs. We have gotten a compile error for src/factor.c because a branch directive and a macro definition referred to in the branch condition are swapped in position. This is caused by an empty macro that appears without the following semicolon. The parser combines this macro call to the following declarations, and the directives between them are treated as undisciplined. This error is unavoidable without knowing that the macro is empty. After replacing the file with the original one, we have confirmed the compilation succeeded. The result of the test has become the same with the original one. This means the tool transforms programs without breaking the semantics, except src/factor.c.

The tool has modified 147 (12.6%) files in 1164 files, not including the five failed files. The directives to be moved comprise 654 (8.9%) directives of all 7232 directives. The total size of the modified files is increased to 202.9%. We have applied the tool again with the option of restricting the branch directives that have else parts, which may make the syntax trees incorrect. In this case, the tool has worked for all files, including the five files failed in the former experiment. The modified files are 32 (2.7%) in 1169 files, and the directives that are to be moved are 98 (4.6%) in 2119 directives. The size of the modified files is increased to 167.4%. The number of undisciplined directives is not high, but it is hard to ignore them. For maintainability, keeping the sizes smaller is better. Our tool uses a brute-force algorithm, and we need to consider sophisticated algorithms [4].

D. Discussion

These results show that the parser accepts real programs and generates syntax trees. Though it is difficult to measure the accuracy of the parser exactly, we can guess that the errors are less than 20% conservatively from the experiments on two successive expressions and type identifications. This is a little higher than we expected, but it does not affect the accuracy of the search tool directly because the tool requires the same results of parsing for both queries and target programs.

The result of the experiments on the reverse macro expansion tool shows that the search tool can find the complex parts, such as control statements and functions. Though we have evaluated the self-application of macros, it is not enough to evaluate that the tool can find all the parts. By testing small examples, we have found that the reverse expansion tool may not replace a macro whose arguments are referred to more than twice. For example, if the macro is "`#define POW(a) ((a)*(a))`," the tool needs to find same two expressions for the argument `a`. If the expression to be replaced is "`(x+1)*(x+1)`," the tool need to compare the two expressions "`(x+1)`" for equality. Our rewriting system, however, has no function for comparing sub-trees, and the tool cannot match the expression. The system can compare the equality of two tokens, such as variables, and there are cases that tool can find the expression, such as Figure 21, in which the argument `h_` appears twice. We are going to improve the rewriting system for supporting equality of two sub-trees.

The experiment on the alignment tool shows that the borders of elements are correctly identified by the parser because the tool is never trapped in an infinite loop and does not break the semantics. Though the tool has failed for five files, the reason is not identification of the borders. These files contain many directives in a statement or a declaration. For example, the declaration of variable `mode_info` in src/stty.c contains 34 directives. These directives have no else parts, and they never break the syntax and the readability. It is a practical solution to align only the directives containing else parts.

A big problem with the search tool is the identification of the classes of identifiers. Especially, the ability to distinguish types and variables/functions has a great effect on the accuracy of the search. We have three options for improving this ability: (a) improving the rules of the parser, (b) using the symbol information of target programs, and (c) ignoring differences between types and variables. The option (a) is a straightforward way for improving parser. The rules for distinguishing types and variables depend on heuristics, and there may be room to improve them. We should, however, avoid introducing complicated heuristics. The option (b) is to use extra information. If the queries are parsed with the information of the identifier classes extracted from the target programs, the correctness of the queries increases. There are cases that this approach does not work. If a type is described as a predefined pattern in a query, and the identifiers corresponding to the type in the target programs are identified incorrectly as variables, the query never matches. The option (c) is to degrade the correctness of syntax trees for both the query and the target programs. By treating all the user-defined types as variables, the effect of incorrect identification can be eliminated. This approach can decrease the false negatives, but

[2] lib/vasprintf.c, src/{sty.c, sig2str.c}, gnulib-tests/test-{signal, fcntl}-h.c

increase the false positives because the matching conditions become weak. We are going to improve the parser while considering these options.

VI. Related work

Ohloh Code [6] provides a code search service. Ohloh adds the tags of elements, such as class and method, in programs, and the tags can be used in the queries. SrcML [7] parses unpreprocessed programs and generates them in XML format. For searching elements, we can use the standard XML methods, such as XQuery and XSLT. These tools do not support structures of expression levels. SrcML does not support the classes of identifiers, and implicit termination of statements. LSME [8] is a token-based fact extractor, whose approach is similar in terms of program pattern matching based on token sequences. LSME does not treat syntactical elements, and the power of the pattern is lower than ours.

There are tools for supporting analysis or refactoring of unpreprocessed programs [7], [9]–[12]. Though these tools accept unpreprocessed programs with disciplined directives or require specific conditions of preprocessing, they provide dependency analysis functions or the base information for them, which our tool set does not support. Our tool set can support them as a preprocessing filter for converting undisciplined directives to discipline ones. CRefactor [13] and SuperC [14] support undisciplined directives, but they align directives in parsing. Our parser preserves undisciplined ones in parsing, and the alignment tool can align directives selectively after parsing.

Rascal [15] supports search patterns as code fragments extended with typed pattern variables, which is the same style with our patterns. The differences are that our patterns support contexts for concrete patterns and that back references of pattern variables can be used in patterns. The former is introduced for avoiding complex patterns matching to unpredictable macro uses, and Rascal may not need this technique because the syntax of a target language is strictly defined.

VII. Conclusion

We have proposed a pattern search method for unpreprocessed programs. Our parser converts the programs to token sequences, and parses them by applying rewrite rules. By using rewrite rules, the parser parses the insides of directives and accepts syntactical errors. Our search tool accepts the queries described as code fragments extended with meta tokens. The tool can search the parts containing the syntactical incorrect elements because the queries and the target programs are parsed using the same parser and the results of parsing are identical. We have also shown the two support tools for accurate search: an alignment tool for branch directives and a reverse expansion tool for macro definitions. Though our tool set does not support the complicated queries using dependencies, it provides us a convenient and lightweight environment for search. They also enable other analyzers to analyze the undisciplined programs by converting them to disciplined ones.

Future work is to improve the class analysis of identifiers. Our experiment shows that the quality of the class analysis strongly affects the one of the search functions. The implementation and the experiments are based on Coreutils 8.22. Though we have applied our tools to FreeBSD, as shown in examples, we have not tested enough on it. There is a threat that the result depends on Coreutils. We need to apply other open source applications and improve rewrite rules of the parser and the search algorithms. We also need to extend our tools for the other programming languages whose programs are mixtures of multiple languages, such as JavaScript and HTML.

Acknowledgment

We would like to thank Prof. James R. Cordy at Queen's University for his useful advice and support. This work was partially supported by JSPS KAKENHI Grant Number 26350344.

References

[1] T. F. Project, "FreeBSD," https://www.freebsd.org.

[2] L. Moonen, "Generating robust parsers using island grammars," in *Proc. 8th Working Conf. on Rev. Eng.* IEEE Computer Society Press, Oct. 2001, pp. 13–22. [Online]. Available: http://www.cwi.nl/~leon/papers/wcre2001/

[3] A. Cox and C. Clarke, "Syntactic approximation using iterative lexical analysis," in *11th Inter. Work. on Program Comprehen., 2003.*, 2003, pp. 154–163.

[4] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 191–202. [Online]. Available: http://doi.acm.org/10.1145/1960275.1960299

[5] T. P. G. D. Group, "PostgreSQL:the world's most advanced open source database," http://www.postgresql.org.

[6] B. D. S. Inc., "Ohloh code search," https://code.ohloh.net.

[7] M. L. Collard and J. I. Maletic, "Document-oriented source code transformatin using xml," in *Proc. of 1st Inter. Work. on Software Evolution and Transformation*, 2004, pp. 11–14.

[8] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 262–292, July 1996. [Online]. Available: http://doi.acm.org/10.1145/234426.234441

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *Proc. 26th Inter. Conf. on Soft. Eng.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 625–634. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999466

[10] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *Proc. 18th Inter. Conf. on Compiler Construction 2009*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 109–125.

[11] D. Waddington and B. Yao, "High-fidelity C/C++ code transformation," *Sci. Comput. Program.*, vol. 68, pp. 64–78, September 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287845.1288023

[12] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," *SIGPLAN Not.*, vol. 46, no. 10, pp. 805–824, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2076021.2048128

[13] A. Garrido, "Program refactoring in the presence of preprocessor directives," Ph.D. dissertation, Champaign, IL, USA, 2005, aAI3199001.

[14] P. Gazzillo and R. Grimm, "SuperC: parsing all of C by taming the preprocessor," *SIGPLAN Not.*, vol. 47, no. 6, pp. 323–334, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254103

[15] P. Klint, T. van der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE Inter. Working Conf. on*, Sept 2009, pp. 168–177.