

Parsing C/C++ Code without Pre-Processing

Yoann Padioleau

University of Illinois, Urbana Champaign

Abstract. It is difficult to develop style-preserving source-to-source transformation engines for C and C++. The main reason is not the complexity of those languages, but the use of the C pre-processor (`cpp`), especially `ifdefs` and macros. This has for example hindered the development of refactoring tools for C and C++.

In this paper we propose to combine multiple techniques and heuristics to parse C/C++ source files as-is, while still having only a few modifications to the original grammars of C and C++. We rely on the fact that in most C and C++ software, programmers follow a limited number of conventions on the use of `cpp` which makes it possible to disambiguate different situations by just looking at the context, names, or indentation of `cpp` constructs.

We have implemented a parser, Yacfe, based on these techniques and evaluated it on 16 large open source projects. Yacfe can on average parse 96% of those projects correctly. As a side effect, we also found mistakes in code that was not compiled because it was protected by particular `ifdefs`, but that was still analyzed by Yacfe. Using Yacfe on new projects may require adapting some of our techniques. We found that as conventions and idioms are shared by many projects, the adaptation time is on average less than 2 hours for a new project.

1 Introduction

The C pre-processor [19], `cpp`, is heavily used by C programmers. Using clever macros, programmers can overcome some of the limitations of C by introducing new features such as iterators, or can perform some metaprogramming, or can factorize any kind of source code text. This possibility to easily extend C is one of the reasons C is still a popular language even 35 years after its creation. As Stroustrup said “without the C preprocessor, C itself ... would have been stillborn” [23]. In fact, `cpp` is even used in programs written in modern languages such as Haskell [28] or λ Prolog. This freedom has nevertheless its price: it makes it hard to parse C source code as-is, which in turn makes it hard to perform style-preserving source-to-source transformations such as refactorings [9] on C source code.

The combination of C and `cpp` leads to the union of two languages that are easy to parse separately but very hard to parse together as the grammar of such union could be very large, contain many ambiguities, and be very different from the original C grammar. For instance, the sequence `X (Y);` could represent a

simple function call or a macro `X` corresponding to a declaration which happens to be followed by a variable `Y` surrounded by extra parenthesis.

Static analysis tools and compilers avoid those problems by simply first calling `cpp` on the source file and then analyze the pre-processed file that now contains only C constructs. Tools such as CIL [16] (C Intermediate Language) also work on a representation of the code that does not directly match the C language but makes the analysis simpler. This is appropriate when the goal is to find bugs or to generate code that no programmer will have to read or modify. However, for style-preserving source-to-source transformations, using this approach would mean working on a version of the file that is very different from its original form, which would make it very hard among other things to back-propagate the transformation to the original file.

Fortunately, what would be hard to parse and disambiguate for a tool would also be hard to parse and disambiguate for a human. Thus, many programmers follow conventions on the use of `cpp` such as the case sensitivity of macros, their indentation, or the context in which macros can be used (Section 2.3). Programmers can then visually rely on these conventions to easily recognize `cpp` usages. We thus propose to leverage such implicit information to parse C/C++ code, like humans do.

The challenges for this work are:

- Grammar engineering. The ANSI C and C++ grammars are already complex and large. Adding new constructions may lead to numerous conflicts. Resolving these conflicts may require significant changes in the grammar.
- The variety of `cpp` idioms. As macros can be used for many different purposes, we need general techniques to recognize those different usages, as well as extensible techniques that can be easily adapted to new software.

In this paper we make the following contributions:

- We have designed general techniques to recognize `cpp` idioms without adding any ambiguity in the ANSI C and C++ grammars. The main ideas are the notion of *fresh* tokens (transforming Yacc in some sense into a LALR(k) tool), the use of generic *views* to easily specify complex code pattern heuristics, and the use of a configuration file containing macro definitions and heuristic hints that can be used as a last resort.
- We have implemented a parser, Yacfe, that can parse most C/C++ code as is, by extending the C/C++ grammars and by writing heuristics that make use of contextual information, names, and indentation.
- We have evaluated Yacfe on large open source projects and shown that our heuristic-based approach is effective for most C/C++ projects and covers most uses of `cpp`.

The rest of the paper is organized as follows. Section 2 describes background on the parsing problems engendered by `cpp`. Section 3 then presents our extensions to the C/C++ grammars to handle `cpp` constructs and our heuristics that makes it possible to add the previous grammar rules without introducing

any shift/reduce or reduce/reduce conflict. Section 4 describes briefly how to use our framework to perform a basic style-preserving program transformation. Section 5 describes the evaluation of Yacfe on large open source software. We finally discuss related work in Section 6 and conclude in Section 7.

2 Background

In the following, we use the term *ambiguity* when grammar rules authorize the same sequence of tokens to be parsed in more than one way. We use the term *conflict* when the ambiguity is only caused by the limitations of the LALR(1) parsing, as implemented by parser generators such as Yacc [11].

The main constructs and usages of cpp are:

- `#include`, mostly for header file inclusion.
- `#ifdef`, for conditional compilation and header file inclusion guards.
- `#define`, to define macros, with or without parameters; in the latter case the macros are often used to describe symbolic constants.

cpp directives can be used anywhere in a C file. Extending the C grammar to handle all possible usages of cpp directives would require extending rules to handle each possibility. The standard solution is instead to expand cpp directives before parsing.

2.1 `#include`

In practice, `#include` directives are mostly used at the start of a file, at the toplevel, and on very few occasions inside structure definitions. It is thus easy to extend the C grammar and the abstract syntax tree (AST) to handle just those cases. Moreover, as the `#include` token is different from any other C tokens, such extension does not introduce any parsing conflict.

2.2 `#ifdef`

In practice, `ifdefs` are mostly used either as inclusion guards at the very beginning and end of a file, or to support conditional compilation of sequence of statements, or to support function signature alternatives. It is also possible by extending the C grammar to handle this limited set of idioms. The following excerpts show one of these idioms and the grammar extension that supports it:

```
#ifdef MODULE
int init_module(int x)
#else
int init_foo(int x)
#endif
{
```

```

function_def ::= dspec declarator { compound }
               | #ifdef dspec declarator #else dspec declarator #endif { compound }

```

The problematic `ifdefs` are those that are used to support certain kinds of alternatives for expression or statement code as in the following:

```

    x = (1 +
#ifdef FOO
        2)
#else
        3)
#endif
    ;
#ifdef _WIN32
    if (is_unix)
        fd = socket(PF_UNIX, SOCK_STREAM, 0);
    else
#endif
        fd = socket(PF_INET, SOCK_STREAM, 0);

```

In the first case, the `#else` can break the structure of the expression at any place, leading to two branches with only partial expressions. It is thus not sufficient as in the previous extension to add a rule such as:

```

expr ::= ...
       | #ifdef expr #else expr #endif

```

Fortunately, these situations are rare as such code would also be difficult for the programmer to understand. One of our heuristics detects some of those situations by checking if the parenthesis or braces are closed in both branches; if not we currently treat as a comment the `else` branch and suggest to the user that he should perhaps rewrite the code. We found only 116 code sites in the 16 software we analyzed that trigger this heuristic.

In the second case, the `#endif` breaks the condition statement; the first branch is a partial statement with a trailing `else`. In such situations one of our heuristic treats instead the `cpp` directive itself as a comment, which allows to parse correctly the full statement. We currently treat 7.5% of the `ifdef` directives as comments and are working to lower this number by handling more `ifdef` idioms.

2.3 #define and macro uses

Dealing with the definition of macros and their uses is more complex. Figure 1 presents various uses of macros representing common idioms used by programmers. Even if most macro uses look like function calls or variable accesses as in (a), in which case they can be parsed by the original C grammar, this is still not the case for many of them. For (b) only `switch/for/while/if` can have a brace or statement after their closing parenthesis, for (c) and (d) it would require at least a trailing `;` to make the construction look like a regular statement or declaration, for (d) what looks like a function call in fact mixes types and expressions as arguments and is, when expanded, a declaration, for (e) what starts as a function declaration has a multiplication as an argument if we do not have more

<pre> #define MAX(a,b) \ ((a)<(b)?(b):(a)) #define LIMIT 3 int x = MAX(foo(1),10); int y = LIMIT; </pre> <p>(a) mimicking functions or constants</p>	<pre> list_for_each(l,e) { printf("%s", e->name); } </pre> <p>(b) iterator</p>
<pre> BEGIN_LOCK if(x != NULL) printf("%s", x->name); END_LOCK </pre> <p>(c) statement macro</p>	<pre> DECL_LIST(x, int, 0); struct x { ACPI_COMMON_FIELDS int x; } </pre> <p>(d) toplevel and structure macros</p>
<pre> #include <foo.h> int f(UINT * y); int foo() { return z * y; } </pre> <p>(e) #include hiding typedefs</p>	<pre> BZ_EXTERN void __init foo(int x); </pre> <p>(f) attributes and qualifiers</p>
<pre> #define DEBUG(A) do \ { printf("ERROR:", A); } \ while(0) DEBUG(1); </pre> <p>(g) #define partial statement</p>	<pre> __P(int, foo, (int x, int y)) { ... } ASSERTCMP(x, <, b); </pre> <p>(h) miscellaneous macros</p>

Fig. 1. A few cpp idioms

contextual information about ‘UINT’, for (f) there are too many identifiers in the prototype, for (g) it would require first to extend the grammar to recognize macro definitions, and to deal with the \ escape, but the body of this macro is only a partial statement as it lacks a trailing ‘;’, and finally for (h), < is a binary operator that requires two operands and cannot be passed as an argument, and the __P macro does not really look like anything close to a C construction.

Extending the C grammar to handle most of those previous examples would lead to parsing conflicts and ambiguities. For instance, we could try to extend the grammar to deal with the iterator (Figure 1(b)) by adding the grammar rule on line 3:

```

1  statement ::= expr ;
2              | for ( [expr] ; [expr] ; [expr] ) statement
3              | id ( expr ( , expr ) * ) statement
4              | ...
5  expr ::= ...
6  arith_expr ::= ...
7  logical_expr ::= ...
8  primary_expr ::= id
9                  | int
10                 | string
11                 | ...

```

Unfortunately this will generate a LALR(1) shift/reduce conflict. Indeed, having analyzed the identifier, seeing an open parenthesis the algorithm can not determine if it is the start of a function call (which requires to reduce to `primary_expr`), or the beginning of a foreach statement (which requires a shift in the statement rule). To be able to decide requires looking ahead at more tokens; in the previous case to know what token is after the closing parenthesis. One could avoid this conflict by reorganizing the grammar so that the reduce action can be delayed, but as the identifier corresponding to the function call in `primary_expr` (line 8) is deeply nested in the grammar, this would involve a substantial change to the original grammar. Another solution would be to have a more tolerant grammar where invalid constructions would be filtered out in a post-parsing phase. One could then add the rule at line 3 and later check that the leading `expr` can only have the form of a function call:

```

1 statement ::= expr ;
2           | for ( [expr] ; [expr] ; [expr] ) statement
3           | expr statement
4           | ...

```

Unfortunately this will generate ambiguities, as the simple `1+1;` statement could be parsed either as a single statement or as the expression `1` followed by the `+1;` statement, as `+` can be used both as an unary and binary operator.

Similar things happen for the other idioms. For instance, adding a rule to allow single identifiers to be used as statements or declarations leads to numerous ambiguities as the sequence `X (1);` could be either a function call or a macro statement `X` followed later by an expression surrounded by extra parenthesis. Moreover, one grammar extension can also make it harder to add further extensions. For instance, the naive iterator and macro statement extensions each generate numerous conflicts; when combined together they generate a number of conflicts that is superior to the sum of the previous conflicts. There is no guarantee that even if one refactors the grammar to avoid one conflict, this refactoring could be kept as one may have to undo or completely change it to make it possible to support another extension.

Note that for most of the idioms in Figure 1, even if one is not familiar with those idioms, it is quite easy for a human to disambiguate them. Indeed, the name, the presence of visual hints such as newlines or the lack of white-space between tokens, and the context surrounding the construct all contribute to make the meaning clear.

3 The Yacfe Engine

In this section we explain our different techniques to handle cpp, which are (1) our way to extend the C/C++ grammars, (2) a heuristic pre-parsing phase that makes the previous grammar extensions possible without introducing parsing conflicts, and (3) a configuration file that allows users to give additional hints to our heuristics.

3.1 Grammar Extensions and Ambiguities

Tokens such as identifiers can play many different roles. Our solution to this problem is mainly to avoid it by replacing widely used tokens such as identifiers with *fresh* tokens that can not generate any conflict with the existing rules. We re-classify tokens in a phase run between lexing and parsing. For instance, for the iterator extension we re-classify some identifier tokens as “iterator” tokens. Then, we can write grammar rules mentioning iterator tokens instead of identifier tokens. A LALR parser can then even with a look-ahead of 1, seeing an identifier decides which rule to use by inspecting the class of the token: a normal identifier, an iterator identifier, a macro statement identifier, etc. What we essentially do is to mimic what programming language designers do when they extend a language: adding new keywords to avoid ambiguities with previous constructions. Here are examples of some of our extensions to the C and C++ grammars that rely on these fresh tokens:

```
1  statement ::= ...
2             | for ( [expr] ; [expr] ; [expr] ) statement
3             | idMacroIterator ( expr ( , expr)* ) statement
4             | idMacroStatement
5  primary_expr ::= id
6                | int
7                | string
8                | idMacroString string*
9                | ...
10 declaration ::= dspec ;
11              | idMacroDecl ( expr_or_type ( , expr_or_type)* ) ;
12 define_body ::= statement
13             | expr
14             | { Bracelnit initializer ( , initializer)* }
15             | ...
16 init_decl ::= declarator
17            | declarator = initializer
18            | declarator ( ParenConstructorC++ expr ( , expr)* )
19 template_id ::= idTemplate < InfTplt type_or_expr ( , type_or_expr)* > SupTplt
```

Re-classifying some tokens requires recognizing some code patterns, which in turn requires a form of parsing. As we will explain in the next section, our heuristics need only simple forms of parsing. We thus use two layers of parsing where the final sophisticated parser relies on the job done using the simpler parser of the previous layer.

We also had to extend the C grammar to deal with C extensions that are often used by open source software such as the extensions implemented by gcc [22] (embedded assembly, case range, attributes, etc). We currently have added 65 new grammar rules on top of the original 195 C rules, to handle the cpp directives and common macro idioms, as well as 59 rules to handle gcc extensions. Note that we almost didn't modify the rules from the original grammar; we didn't have to reorganize the existing rules while adding the new rules because each extension was local, thanks to the fresh tokens. We only had to slightly refactor the original compound rule because of interactions with `ifdefs` constructs.

3.2 Heuristics and Views

Some heuristics are needed to detect specific tokens that must be re-classified. These interesting tokens are often identifiers, corresponding to different cpp macro idioms, but they can also be specific `ifdef` tokens or even some open parenthesis tokens. Our heuristics look at the context of those tokens, the structure of their names, or their indentation. Some of these heuristics may need to access to a large context of the token such as a large sequence of tokens after (look-ahead) or even before the specific token. Even if some programming languages such as ML provide powerful pattern-matching capabilities over lists and algebraic data-types (ADTs), it is not easy to write some of our heuristics working on such token lists. For instance, for the `foreach` heuristic, we may want informally to look at code patterns like `. *each.* (...)` {, but this can only be incompletely translated in ML (in the OCaml [12] syntax) as:

```
match token_list with
| Id(s) :: TSym("(") :: TSym(")") :: TSym("{") :: _
| Id(s) :: TSym("(") :: _ :: TSym(")") :: TSym("{") :: _
| Id(s) :: TSym("(") :: _ :: _ :: TSym(")") :: TSym("{") :: _
| ...
when s =~ ". *each.*" ->
```

Moreover, even if we could use a form of generalized regular expression over ADTs (as in XDuce or Prolog-III), such regular expressions would not cope with the problem of possible nesting of parenthesized expressions.

To solve this problem we propose the notion of *view* over these tokens that offers an additional layer on top of the list of tokens, to group them into different classes. Views make it possible to use the traditional pattern-matching features of languages such as ML to easily express complex code patterns. This idea was proposed by Wadler in [27] but required originally to extend the programming language. In our case, we do not need to extend ML; we have implemented views using ordinary functions and references (to make it possible to modify elements in the views), looking at views as an idiom instead of a programming language feature. For the iterator example, a parenthesized view adds a tree layer over the list of tokens allowing the previous heuristic to be expressed as:

```
match paren_view token_list with
| Leaf(Id(s) as t1) :: ParenNode(_) :: Leaf(TSym("{")) :: _
```



```

when s =~ ".*each.*" ->
  reclassify t1 TMacroIterator

```

This heuristic looks for a token identifier, followed by a parenthesized term (containing possibly some nested parenthesized terms), followed by an open brace, and re-classify the leading identifier as an iterator identifier if it contains the word “each”. This heuristic will thus reclassify the `list_for_each` identifier in Figure 1(b) but not the `MAX` identifier in Figure 1(a) as the closing parenthesis of the `MAX` parameters is not followed by an open brace.

We have currently implemented 5 views: the parenthesized view, braceized view, ifdef view, line view, and combined line and parenthesized view. These views group tokens in different manners and cover most of our needs. The preceding heuristic is in fact incomplete as it could incorrectly reclassify function names in function definitions such as `int each(int i) { ...`. The current heuristic thus for instance also checks that the identifier is indented, is not at the toplevel, and is not preceded by any other token on its line, using the line view and contextual information from the braceized view. The heuristic also allows for more words than “each” to be part of the identifier (e.g. “loop”).

Here is another heuristic using the combined line and parenthesized view.

```

(* ex: BEGIN_LOCK(X,Y) without trailing ';' *)
match lineparen_view token_list with
| Line([NoL(Id(s) as tok1); ParenthesisedL(_)])
:: Line(tok2::_)
:: _
when indent_of tok1 <= indent_of tok2 &&
  is_upper_case s ->
  reclassify tok1 TMacroStatementParams

```

The code for our heuristics and view constructions currently consists of 2300 lines of OCaml code. An important part of this code is dedicated to the detection of typedefs or type-macro identifiers as in Figure 1(e). An alternative would be to write a dedicated analysis to gather all the typedef declarations from header files in order to build a symbol table for typedef identifiers. Nevertheless, it would require to know where are located those header files (usually specified in makefiles with the `-I` cpp flag) and to have access to those files. For the code of multi-platform applications, this is not always convenient or possible. For instance, one may not have access to the Windows system header files on a Unix machine. We thus opted to try to infer typedefs using heuristics and parse code even with partial information on types.

3.3 Configuration File and Extensibility

Even if the heuristics we have written using the previous views capture many conventions, there are still software or specific programmers using slightly different conventions. For these cases, it would be too demanding to ask to the users of Yacfe to modify the code of Yacfe to add new heuristics. Moreover, those heuristics might be valid only for a set of macros and generate false positives

when applied globally. Therefore, to offer an easy way to partially extend Yacfe, we propose to use an external configuration file where the user can manually specify the *class* of specific but recurring macros that cause the above heuristics to originally fail. We have reused the syntax of cpp for the format of this file but Yacfe recognizes special keywords used in the body of macros as hints. Here is an excerpt of the configuration file for the Linux kernel:

```
#define change_hops YACFE_MACRO_ITERATOR
#define DBG          YACFE_MACRO_STATEMENT
#define KERNEMERG    YACFE_MACRO_STRING
#define DESC_ALIGN   YACFE_MACRO_DECL
```

This file can also be used as a last resort as a way to partially call cpp for difficult macros, such as the one in Figure 1(h), by adding for instance this definition:

```
#define __P(returnt, name, params) returnt name params
```

In this last case, the resulting expansion is marked specially in the AST so that even if a tool using Yacfe can match over such expansion, such tool will be warned if it wants to transform those expansions.

To assist the user in creating those configuration files, Yacfe remembers, while attempting to parse for the first time the different files of a software, the identifiers in the line before and on the same line than a parsing error. Yacfe then returns to the user the 10 most recurring identifiers and displays each time one example of a parsing error containing the identifier. This helps to quickly find and define the recurring difficult macros.

3.4 Other Techniques

To faithfully represent the original program, we also had to keep extra tokens in the AST which are normally abstracted away, such as extra parenthesis as in $(1+((2)))$. We thus have more a concrete syntax tree (CST) than an AST.

We have also implemented an error recovery scheme so that a parsing error in one function does not hinder the parsing of the rest of the file. In case of a parsing error, Yacfe first displays the line where the error occurred and the content of the file around that line. Yacfe then skips the set of tokens before the next function (using heuristics to detect the start of the next function), and returns a special AST error element, indicating the parsing error, containing all the tokens that were skipped (this is useful to compute the statistics of Section 5). Finally, Yacfe re-run the parser for the remaining tokens, for the next function.

Surprisingly, extending Yacfe to handle C++ was not as hard as we imagined. It took us about 2 weeks to parse more than 90% of the source code of Mozilla and MySQL. Parsing C++ is known to be difficult due to the complexity of the language and the numerous ambiguities and LALR(1) conflicts in its official grammar. These ambiguities require contextual information or post-processing semantic analysis to be resolved, as described in the annotated C++ reference manual [5]. Nevertheless, by applying the same techniques we used to disambiguate cpp idioms, we were able by introducing new fresh tokens and their

associated heuristics to parse most C++ code, while using the original C++ grammar almost as-is.

4 Using Yacfe

Parsing is only one component of a program transformation system, but a crucial one. Yacfe offers other services such as type-checking, visitors, AST manipulations, and style-preserving unparsing. This last functionality was relatively easy to implement because of the way we originally parse the source code. The description of those features is outside the scope of this paper and Figure 2 illustrates just a simple example of program transformation using the Yacfe framework (in the OCaml [12] syntax). The transformation consists in replacing every occurrences of ‘0’ by the NULL macro in expressions involving relational comparisons on pointers.

```
(* X == 0 -> X == NULL when X is a pointer *)
open Ast_c
let main =
  let ast = Parse_c.parse_c_and_cpp Sys.argv.(1) in
  Type_annoter_c.annotate_program ast;
  Visitor_c.visit_program {
    Visitor_c.kexpression = (fun k exp ->
      (match Ast_c.unwrap exp with
      | Binary(e1, Logical (Eq), Constant(Int("0")) as e2) ->
        (match Type_c.type_of_expr e1 with
        | Pointer _ ->
          let idzero = Ast_c.get_tokens_ref1 e2 in
          Ast_c.modify_token (fun s -> "NULL") idzero;
        | _ -> ()
        )
      | _ -> k exp
      );
    };
  } ast;
  let tmpfile = "/tmp/modified.c" in
  Unparse_c.unparse ast tmpfile
```

Fig. 2. A simple style-preserving program transformation using Yacfe API in OCaml

Note that as the pointer expression can be an arbitrary expression, including complex computations inside parenthesis, and as the transformation also requires semantic (type) information, it would be hard with lexical tools such as `sed` to correctly perform even this simple transformation. Moreover, as many macros expand to ‘0’, working on the pre-processed code, like most tools do, could lead to many false positives.

C frontends such as CIL [16] also offer visitor services and program transformation abilities, but they output source code that is very different from the original code as they operate after preprocessing. C refactoring tools such as Xref [25] can not perform the transformation shown in Figure 2 as it is not part of their limited set of supported refactorings. Xref supports the renaming of simple macros but performs incorrectly for instance the renaming of an iterator macro.

5 Evaluation

In this section we evaluate the applicability of our techniques by testing if Yacfe can parse the code of popular software. All experiments were made on an Intel Core 2 at 1.86GHz with 2Go of RAM and a 160Go SATA disk. Table 1 presents the parsing results of Yacfe on different types of software (kernel, browser, compiler, game, etc).

Software	Languages	Age (Years)	Type	LOC (kilo)	skipped (%)	correct (%)
Linux	C	17	Kernel	8050k	1.33	98.96
Mozilla	C/C++	14	Browser	5073k	3.05	95.58
Mysql	C/C++	13	Database	1306k	1.82	93.23
Qemu	C	5	Emulator	434k	3.30	97.00
emacs	C/Lisp	31	Editor/OS	395k	4.30	96.14
git	C	3	VCS	94k	0.03	99.91
sparse	C	5	C frontend	26k	0.69	99.41
gcc	C	31	Compiler	1421k	1.45	97.39
Quake III	C	9	Game	311k	2.15	96.09
openssh	C	9	Network	82k	0.69	99.16
pidgin	C	9	Communication	426k	1.48	99.35
kdeedu	C++	7	Education	315k	1.19	95.22
glibc	C	30	Base Library	773k	3.95	92.37
libstdc++	C++	10	Base Library	438k	3.43	55.22
sdl	C	10	Game Library	201k	3.05	95.83
gtk	C	10	GUI Library	737k	0.75	98.20

Table 1. Yacfe parsing results and statistics on 16 large open source projects

Yacfe can parse on average correctly 96% of the code. The percentage is in number of lines. By parsing correctly we mean returning successfully AST elements that are not the AST error element mentioned in Section 3.4. The AST elements may still contain mistakes as we may have bugs in our parser. In fact, we had such bugs in the past and for instance, because of a typo, we generated in the AST the same tree for expressions involving the < and > operators. But, as we have now used extensively the Yacfe framework for more than a year, to

perform program transformations on Linux drivers [17], we found such typos and are now confident that the returned ASTs actually match the source correctly, at least for the C parser (we have not yet tested extensively our C++ parser).

The skipped column represents the percentage of lines that are either (1) completely skipped by Yacfe, for instance for code in branches containing partial statements as explained in Section 2.2 or in `#if 0` branches, or (2) partially skipped, for instance when a line contains a problematic macro mentioned in the configuration file which must be expanded.

For each piece of software, it took us on average less than 2 hours to write the configuration file containing on average 56 hints or definitions of recurring problematic macros. The average time to parse a file is about 0.03s. Analyzing the whole Linux kernel with Yacfe takes 12 minutes, whereas compiling it with gcc (with the maximal number of features and drivers) takes 56 minutes.

Note that even if Yacfe does not parse correctly 100% of the code, it still analyzes in general more code than gcc which processes only certain ifdefs. On an Intel machine with all the features ‘on’, gcc compiles only 54% of the Linux kernel C files, as the code of other architectures is not considered. Moreover, using static analysis tools such as Splint [7] requires also a setup time. For instance, configuring Splint to analyze one Linux kernel driver took us more than 2 hours as Splint can work only after preprocessing which requires among other things to find the appropriate cpp options (more than 40 `-D` and `-I` flags), spread in Makefiles or specific to gcc, to also pass to Splint. This is not needed with Yacfe. Bug detection tools can thus have false positives, for code inside certain ifdefs, as they don’t analyze the whole original source code.

The remaining errors are due to features not yet handled by our parser such as embedded assembly for Windows, Objective C code specific to MacOS, the gcc vector extension, or cpp idioms and macro body definitions not yet handled by our heuristics or grammar or configuration files. Among those errors, we also found true errors in certain ifdef branches which had probably not been compiled for a long time. Some Yacfe parsing errors were also raised because the indentation of macros was not correct, which prevents our heuristics to work (often because tools like `indent` are not aware of cpp constructs and wrongly indented the code). We found 31 such mistakes in the Linux kernel, and have submitted patches that have now been accepted and are part of the latest kernel. We also found 1 or 2 mistakes in Qemu, Openssh, Pidgin, and Mozilla.

Table 1 also shows that the younger the software, the easier it is for Yacfe to parse it. This is probably because the pitfalls of `cpp` are now widely known and thus programmers are more conservative in their use of `cpp`.

Yacfe can still not parse most of the C++ code in the C++ standard GNU library as this code use advanced C++ features and macros not yet handled by our heuristics and grammar. In fact the code is also arguably very hard to disambiguate for a human.

6 Related Work

Ernst et al. [6] presented an extensive study on the usage of `cpp` in 26 open source software, representing a total of 1.4 millions lines of code. They relied on the PcP3 [3] tool to compute various statistics. PcP3 is a framework for pre-processor aware code analysis where the code is still pre-processed but by an embedded `cpp` in which the user can write Perl functions invoked when certain `cpp` constructs are recognized. While this approach using hooks might be enough to statically analyze code, for instance to find `cpp`-related bugs, PcP3 offers no support for program transformation as the code is still in the end preprocessed.

Based on the above study, Brewer et al. [14] proposed a new pre-processor language for C, Astec, which can express many `cpp` idioms while being more amenable to program analysis. They also presented a tool to assist users in migrating code from `cpp` to Astec. They tested their approach on 3 small software and a small subset of Linux.

Past works to parse C code as-is have focused mainly on `ifdefs` [2, 10]. Baxter et al. [2] proposed an extended C grammar, AST, and symbol table dealing with `ifdef` directives, similar to what we presented briefly in Section 2.2. They also impose constraints on where such directives can be placed, mainly at boundaries between statements or declarations. Garrido et al. [10] extended this approach to deal with directives not placed on clean boundaries, for instance between partial expressions as in the example in Section 2.2. They proposed the notion of pseudo-pre-processing where some code preceding `ifdefs` of partial statements or expressions are internally distributed in both branches, to complete them, but marked specially in the AST to make it still possible to back propagate modifications on the original code. They tested their approach on 2 small software. We found, on the code we analyzed, that more than 90% of the `ifdefs` are placed at clean boundaries as it makes the code more readable. Some programmers have also argued that some use of `ifdefs` are considered harmful [20].

Few works have focused on macros, which we found in practice more problematic than `ifdefs` regarding parsing. Livadas et al. [13] proposed another pre-processor, Ghinzu, allowing to track and map code location in the expanded code to the original code. Baxter et al. [4] briefly described the handling of `cpp` in their commercial program transformation system framework DMS. As in PcP3 they implemented their own pre-processor called between the lexing and parsing, but use it only when necessary. They retain some `cpp` macros uses in the AST when possible and fall-down to their embedded pre-processor by expanding some macros in case of parsing errors.

Baxter et al. [4] as well as McPeak et al. [15] have argued for the use of Generalized LR[24] parsing (GLR, or parallel LR) instead of LALR(1) as in Yacc, especially to deal with the C++ language, independently of the pre-processing problem. Using a GLR tool does not reduce the number of conflicts in a grammar, as GLR is still a LR-based technique. But, instead of favoring one choice, for instance shift over reduce, as in Yacc, GLR tries both possibilities and returns a set of parse trees. In some cases many parsing branches eventually reach a dead-end and only one parse tree is returned. The shift/reduce conflict introduced when

adding the iterator construct in the grammar in Section 2.3 is thus irrelevant when using a GLR parser. In other cases, many parsing branches could succeed and GLR thus postpones the disambiguation problem to a post-parsing phase. Using more semantic information the programmer must then decide which of those parse trees are invalid. In this paper we opted instead to disambiguate a priori using views and heuristics, as lexical information such as name or indentation are more relevant than semantic information to disambiguate cpp idioms. Moreover, by using fresh tokens we can have a grammar without almost any LR conflicts whereas using GLR without our fresh tokens would lead for our C grammar to many conflicts and no static guarantees that all ambiguities are resolved by the post-parsing phase. The C++ grammar written by McPeak thus contains 138 conflicts, and does not handle any cpp constructs. Our C and C++ grammars, which also handle cpp constructs, contain respectively 1 and 6 shift/reduce conflicts, including the classic dangling else conflict, and were all resolved by adding precedence directives in the grammar.

To solve some of the C++ conflicts, Willink [29] has heavily rewritten the original C++ grammar, for instance to better deal with templates, which use the `<` and `>` symbols already used for relation comparisons in expressions. Nevertheless, as opposed to the original grammar which provides a useful readable specification of the language, the modified grammar of Willink is hard to read and breaks the conceptual structure of the original grammar. It is also a superset of the language and requires a post-processing analysis to generate the final AST and solve ambiguities.

There are two dedicated refactoring tools for C/C++ listed on the refactoring website [8], including Xref [26], and some IDEs such as Eclipse have some support for C/C++. Nevertheless, they support only a limited set of refactorings, which in turn represent only a small part of the program-transformation spectrum. They do not offer any general mechanism to deal with cpp. Instead, Xref uses a classical C front-end, EDG [1], which like PCP3 implements its own cpp preprocessor. It provides opportunities to track cpp uses, but not to transform them.

Spinnelis [21] focused on the rename-entity refactoring that existing refactoring tools may implement only partially as they can miss some code sites (false negatives). This is because cpp macros can concatenate identifiers with the `#` cpp operator, generating identifiers that may not be visible directly in the original source code. Spinnelis thus proposed techniques to statically track those concatenations. We instead extended the AST and the grammar to accept such concatenation constructs and postpone those analysis to transformation engines working on our extended AST.

No previous work tried to leverage the implicit information programmers use to disambiguate cpp usages, or to represent cpp idioms directly in the AST. They thus all work on program representations that do not directly reflect the original code. Most of the previous work have also been applied only to small software.

7 Conclusion

In this paper we have presented Yacfe, a parser for C/C++ that can represent most cpp constructs directly in the AST. This is made possible by adding new grammar rules, recognizing cpp idioms, without introducing any conflict and ambiguity in the original C and C++ grammars by using fresh tokens. Those fresh tokens are generated by heuristics that leverage the name, context, and indentation of cpp constructs.

We have used Yacfe in the past as part of a project to evolve Linux device drivers [17] in which the correct parsing of most code helped automate most of the work. We have also used it as part of a source code comment study [18] where the maintenance of cpp constructs in the AST was necessary. We hope Yacfe can be useful to other researchers in situations that require manipulating source code as-is, such as for refactoring, when evolving APIs, when offering a migration path from legacy code to modern languages, or to find bugs at the cpp-level for instance on the incorrect use of macros.

Availability

The source code of Yacfe as well as the data used for this paper are available on our web page: <http://opera.cs.uiuc.edu/~pad/yacfe/>.

Acknowledgments

Thanks are due to Julia Lawall and Lin Tan for comments on earlier drafts of this paper, to the anonymous reviewers for suggesting the idea to assist the user in creating the configuration file, and to YuanYuan Zhou to let me spend time on this paper. This work was carried out in part at the EMN. This work was supported in part by the Agence Nationale de la Recherche (France) and by the NSF under grant CNS 06 15372.

References

1. EDG C++ frontend. Edison Design Group www.edg.com.
2. AVERSANO, L., PENTA, M. D., AND BAXTER, I. D. Handling preprocessor-conditioned declarations. In *International Workshop on Source Code Analysis and Manipulation* (2002).
3. BADROS, G. J., AND NOTKIN, D. A framework for preprocessor-aware C source code analyses. *Software, Practice and Experience* (2000).
4. BAXTER, I. D., PIDGEON, C., AND MEHLICH, M. DMS: Program transformations for practical scalable software evolution. In *ICSE* (2004).
5. ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
6. ERNST, M. D., BADROS, G. J., NOTKIN, D., AND MEMBER, S. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering* (2002).
7. EVANS, D. Splint, 2007. <http://www.splint.org/>.

8. FOWLER, M. Refactoring tools. <http://www.refactoring.com/tools.html>.
9. FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
10. GARRIDO, A., AND JOHNSON, R. Analyzing multiple configurations of a C program. In *ICSM* (2005).
11. JOHNSON, S. C. Yacc: Yet another compiler-compiler. Tech. rep., 1979. Unix Programmer's Manual Vol 2b.
12. LEROY, X. Ocaml. <http://caml.inria.fr/ocaml/>.
13. LIVADAS, P. E., AND SMALL, D. T. Understanding code containing preprocessor constructs. In *IEEE Workshop on Program Comprehension* (1994).
14. MCCLOSKEY, B., AND BREWER, E. ASTEC: a new approach to refactoring C. In *FSE* (2005).
15. MCPeAK, S., AND NECULA, G. C. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction* (2004).
16. NECULA, G. C., MCPeAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction* (2002).
17. PADIOLEAU, Y., LAWALL, J. L., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008).
18. PADIOLEAU, Y., TAN, L., AND ZHOU, Y. Listening to programmers: Taxonomies and characteristics of comments in operating system code. In *ICSE* (2009).
19. RITCHIE, D. M., AND KERNIGHAN, B. *The C Programming Language*. Prentice Hall, 1988.
20. SPENCER, H. `#ifdef` considered harmful, or portability experience with C News. In *USENIX Summer* (1992).
21. SPINELLIS, D. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering* (2003).
22. STALLMAN, R. M. *Using GCC*. GNU Press, 2003. GNU C extensions, http://gcc.gnu.org/onlinedocs/gcc/index.html#toc_C-Extensions.
23. STROUSTRUP, B. *The Design and Evolution of C++*. Addison Wesley, 1994.
24. TOMITA, M. An efficient context-free parsing algorithm for natural languages. In *IJCAI* (1985).
25. VITTEK, M. Xrefactory for C/C++. <http://xref-tech.com/xrefactory/main.html>.
26. VITTEK, M. Refactoring browser with preprocessor. In *Conference on Software Maintenance And Reengineering* (2003).
27. WADLER, P. Views: A way for pattern matching to cohabit with data abstraction. In *POPL* (1987).
28. WANSBROUGH, K. Macros and preprocessing in Haskell, 1999. <http://www.cl.cam.ac.uk/~kw217/research/misc/hspp-hw99.ps.gz>.
29. WILLINK, E. D., VYACHESLAV, AND MUCHNICK, B. Fog: A meta-compiler for C++ patterns. Tech. rep., 1998.