Parsing All of C by Taming the Preprocessor

NYU CS Technical Report TR2011-939

Paul Gazzillo Robert Grimm
New York University
{gazzillo,rgrimm}@cs.nyu.edu

Abstract

Given the continuing popularity of C for building large-scale programs, such as Linux, Apache, and Bind, it is critical to provide effective tool support, including, for example, code browsing, bug finding, and automated refactoring. Common to all such tools is a need to parse C. But C programs contain not only the C language proper but also preprocessor invocations for file inclusion (`#include`), conditional compilation (`#if`, `#ifdef`, and so on), and macro definition/expansion (`#define`). Worse, the preprocessor is a textual substitution system, which is oblivious to C constructs and operates on individual tokens. At the same time, the preprocessor is indispensable for improving C's expressivity, abstracting over software/hardware dependencies, and deriving variations from the same code base. The x86 version of the Linux kernel, for example, depends on about 7,600 header files for file inclusion, 7,000 configuration variables for conditional compilation, and 520,000 macros for code expansion.

In this paper, we present a new tool for parsing all of C, including arbitrary preprocessor use. Our tool, which is called SuperC, is based on a systematic analysis of all interactions between lexing, preprocessing, and parsing to ensure completeness. It first lexes and preprocesses source code while preserving conditionals. It then parses the result using a novel variant of LR parsing, which automatically forks parsers when encountering a conditional and merges them again when reaching the same input in the same state. The result is a well-formed AST, containing static choice nodes for conditionals. While the parsing algorithm and engine are new, neither grammar nor LR parser table generator need to change. We discuss the results of our problem analysis, the parsing algorithm itself, the pragmatics of building a real-world tool, and a demonstration on the x86 version of the Linux kernel.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Languages, Algorithms

Keywords C, preprocessor, LR parsing, fork-merge LR parsing, SuperC

1. Introduction

C code continues to be essential to all aspects of computing, touching on government, business, and leisure. Large and complex programs, such as Linux, Apache, and Bind, run on a wide range of devices, from embedded systems (e.g., smartphones) to servers (e.g., cloud services). Consequently, effective tool support for developing and maintaining C code is of critical importance. For example, source code browsers can help write new C code, bug finders help improve existing C code, and automated refactorings help evolve large and complex codebases.

But all these tools are considerably complicated by the fact that C code mixes *two* languages: the C language proper and the preprocessor, which supports file inclusion (`#include`), conditional compilation (`#if`, `#ifdef`, and so on), and macro definition/expansion (`#define`). Preprocessor usage is pervasive [11, 32]. For example, the x86 version of the latest stable Linux kernel (2.6.39.2) depends on about 7,600 header files for file inclusion, 7,000 configuration variables for conditional compilation, and 520,000 macros for code expansion. Preprocessor usage isn't limited to the kernel either: The familiar `stdio.h` program header in Ubuntu 10.0 depends on 44 more files with a total of 1,293 conditionals, which are nested up to 35 levels deep, and 966 macros, which are defined up to five times each (through conditionals). To make matters worse, the preprocessor is a textual substitution system; it is oblivious to C constructs and operates on individual tokens. For example, `stdio.h` contains three conditionals and five macros with incomplete syntactic units (expressions, statements, and so on).

Existing tools for developing and maintaining C punt on the full complexity of processing the two languages at the same time. They assume that preprocessor usage is well-formed, i.e., that directives surround C's syntactic units, and/or they process a single configuration at a time. Most research focused on parsing the two languages does not fare any better [1, 3–5, 12, 16, 25, 27, 31, 35]. Only the recent TypeChef project seeks to completely solve the problem of parsing C with arbitrary preprocessor directives [21, 22]. It comprises three basic ingredients: (1) preserve conditionals during preprocessing—in TypeChef, by annotating each token with its *presence condition*, i.e., the conjunction of nested conditional tests; (2) fork parser states on diverging presence conditions and then merge them again on convergence; and (3) embed all resulting alternatives in the abstract syntax tree (AST) through static choice nodes. But TypeChef also suffers from three limitations. First, it does not support several, more subtle features of the preprocessor and C, requiring careful manual configuration to parse the x86 version of Linux. Second, it is fairly complex, combining lexing and preprocessing into a new "variability-aware lexer" and reengineering the parser through a new LL parser combinator library. Third, its Scala-based implementation is relatively slow. In our tests on x86 Linux, it has a median per-file latency of 17.9 seconds, 90th percentile latency of 26.6 seconds, and worst-case latency of 652.2 seconds.

This paper presents SuperC, a new tool for parsing C with arbitrary preprocessor directives, which uses the same basic approach (developed independently) while improving on TypeChef in all aspects. First, SuperC is based on a systematic analysis to identify all interactions between lexing, preprocessing, and parsing and to ensure that our tool is complete. Second, SuperC is carefully de-

signed to keep individual concerns separate and to only innovate where necessary. Crucially, it is based on a novel variant of LR parsing, which handles preprocessor conditionals by automatically forking and merging parser states and eliminates the need to rewrite grammars as well as parser table generators. These *fork-merge LR parsers* are comparable to GLR parsers, such as those generated by Elkhound [26] or SDF2 [8, 34], in that they tolerate ambiguity. But whereas GLR parsers allow for several nonterminals to match the same input, fork-merge LR parsers allow for several inputs (conditional branches) to match the same nonterminal. Third, by building on an LR foundation, SuperC is faster than TypeChef. For the same x86 Linux kernel, it has a median per-file latency of 5.8 seconds, 90th percentile latency of 6.6 seconds, and worst-case latency of 14.2 seconds. Since our implementation assumes little in advanced language support (unlike TypeChef's Scala-based combinator library), SuperC also is more easily portable to lower level languages (such as C) that allow for further, fine-grained performance tuning.

This paper makes the following contributions:

- A systematic analysis of the challenges involved in parsing C with arbitrary preprocessor usage and an empirical quantification for the x86 Linux kernel.
- A novel variant of LR parsing, which recognizes preprocessor conditionals without the need for rewriting grammars or parser table generators.
- A comprehensive description of SuperC, a tool for parsing all of C, and its demonstration on the x86 version of the Linux kernel.

We have released SuperC as open source at `http://cs.nyu.edu/rgrimm/xtc/`. Retargeting it to other languages is relatively straight-forward: change that language's preprocessor to preserve conditionals and connect it with our Fork-Merge LR parsing engine. No changes are necessary to the language's LR grammar (if available), the parser table generator, or the parsing engine. Finally, we leave type checking across conditionals as future work (just like TypeChef).

2. Problem and Solution Approach

The main challenge in processing C is that a program, as written, is fundamentally different from its semantics, as expressed through C constructs proper, due to wide-spread preprocessor usage. File inclusion means that individual files are only fragments of compilation units. Macros mean that fragments of program text can be completely different from the actual program. And conditionals mean that only some fragments of program text are valid. So, even when processing only one configuration at a time, any C tool needs to go through three steps to build an AST. First, it lexes each source file to convert sequences of characters into tokens. Second, it preprocesses the tokens, resolving file includes, macro definitions and expansions, as well as conditionals. Third, it parses the resulting tokens, which represent only C constructs proper. The preprocessor and parser can exchange tokens, instead of printing and lexing again, since they share key lexical syntax.

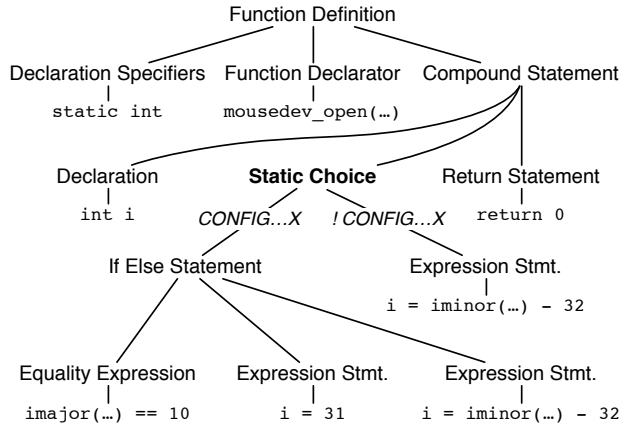
This pipeline of lexer, preprocessor, and parser does not change when processing all of C, with two crucial differences: Preprocessing now needs to preserve conditional branches and parsing now needs to explore all branches while also integrating them into the AST as static choice nodes. Furthermore, since conditionals may appear around arbitrary tokens, the parser may only combine branches after recognizing complete C constructs. This approach to parsing all of C was first suggested by Garrido and Johnson for CRefactory [16], though their parser recognizes only some conditional usage. It is also taken by TypeChef and our own SuperC. McCloskey and Brewer have explored normalizing (some) macro

```
1  #include "major.h" // Defines MISC_MAJOR to be 10
2
3  #define MOUSEDEV_MIX 31
4  #define MOUSEDEV_MINOR_BASE 32
5
6  static int mousedev_open(struct inode *inode, struct file *file)
7  {
8    int i;
9
10 #ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
11   if (imajor(inode) == MISC_MAJOR)
12     i = MOUSEDEV_MIX;
13   else
14 #endif
15     i = iminor(inode) - MOUSEDEV_MINOR_BASE;
16
17   return 0;
18 }
```

(a) The unpreprocessed source.

```
1  static int mousedev_open(struct inode *inode, struct file *file)
2  {
3    int i;
4
5  #ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
6    if (imajor(inode) == 10)
7      i = 31;
8    else
9  #endif
10     i = iminor(inode) - 32;
11
12   return 0;
13 }
```

(b) The preprocessed source preserving all configurations.



(c) Sketch of the AST containing all configurations.

Figure 1. From source code to preprocessed code to AST. The example is edited down for simplicity from drivers/input/mousedev.c.

invocations in Macroscope [25]. But due to the many and often subtle interactions between preprocessor features (see below), we are skeptical whether a general solution for macro normalization is, in fact, possible. Figure 1 illustrates the three steps—from source code to preprocessed code to abstract syntax tree—on a program snippet, which, like all examples in this paper, has been taken from the Linux kernel.

Two basic design decisions immediately arise from this approach to parsing all of C: (1) How to represent the different con-

```
1 #ifdef CONFIG_SCHED_DEBUG
2 # define const_debug __read_mostly
3 #else
4 # define const_debug static const
5 #endif
6
7 const_debug unsigned int sysctl_sched_nr_migrate = 32;
```

Figure 2. A multiply defined macro from kernel/sched.c.

figurations after preprocessing and (2) what parsing technology to build on. First, SuperC preserves conditional directives as *compound tokens* and otherwise relies on regular tokens, whose *presence conditions* are implicit in the boolean conjunction of nested conditional tests. This representation is close to the original source and facilitates pretty printing code after preprocessing for debugging purposes, as with gcc's `-E` option. In contrast, TypeChef directly annotates each regular token with its presence condition, which trades representational uniformity against preprocessor complexity. Second, SuperC builds on LR parsing, which makes the parser stack explicit and thus makes it trivial to fork parsing state for conditional branches. It also is key to SuperC directly reusing existing grammars and parser table generators. In contrast, TypeChef relies on an LL parser combinatory library, which trades ease of experimentation against advanced language support and relatively slow parser performance.

2.1 The Gory Details

With the basic approach in place, we now turn to a systematic analysis of the interactions between lexing, preprocessing, and parsing C code. Most complications stem from three properties of the preprocessor. First, the preprocessor operates on tokens and is oblivious to C constructs. Directives and macro invocations alike may appear between arbitrary tokens and also produce arbitrary tokens. Second, preprocessor directives cannot be nested within each other. They must start with the beginning of a source line and must end with the end of that same line. Yet, macro invocations and entire conditionals may span several source lines. Third, preprocessor invocations may be nested within each other. Notably, macro invocations may produce further macro invocations. They may also span conditionals, with only part of the invocation being contained in a conditional.

Table 1 summarizes all interactions between lexer, preprocessor, and parser. Rows denote features and are grouped by the three steps. The first column names the feature, the second column identifies language and source granularity, and the third column describes the general processing strategy. The remaining columns capture complications due to feature interactions. If an interaction applies, the corresponding table entry explains how to overcome the complication. Gray table entries highlight interactions not yet supported by TypeChef. In contrast, SuperC does address all interactions—with exception of annotating tokens with layout as well as error, warning, pragma and line directives, which is mostly implemented but still too buggy to use.

Layout. The first step is lexing, which converts raw program text into tokens while also stripping layout, i.e., whitespace and comments. Lexing is performed before both preprocessing and parsing and does not interact with the two latter steps. At the same time, automated refactorings, by definition, restructure source code and need to output source code *as originally written*, modulo any intended changes. Consequently, they need to preserve layout, and a complete solution for parsing C needs to (optionally) annotate tokens with surrounding layout—plus, keep sufficient information about preprocessor invocations to restore them as well.

```
1  // In include/linux/byteorder/little\_endian.h:
2  #define __cpu_to_le32(x) ((__force __le32)(__u32)(x))
3
4  #ifdef __KERNEL__
5  // Included from include/linux/byteorder/generic.h:
6  #define cpu_to_le32 __cpu_to_le32
7  #endif
8
9  // In drivers/pci/proc.c:
10 _put_user(cpu_to_le32(val), (__le32 __user *) buf);
```

Figure 3. A macro conditionally expanding to another macro name.

Macro (un)definitions. The second step is preprocessing. It collects macro definitions (`#define`) and undefinitions (`#undefine`) into a macro symbol table—with definitions being either object-like

`#define` *name body*

or function-like

`#define` *name*(*parameters*) *body*

Both definitions and undefinitions for the same name may appear in different conditional branches. As a result, the macro symbol table needs to store multiple versions of such a macro, annotated by their presence conditions. Furthermore, since macros exist in a single, global scope, the macro symbol table needs to replace versions, when a macro is re- or undefined under the same presence conditions. When a macro's definition depends on a configuration, as in lines 1–5 of Figure 2, the macro propagates an *implicit conditional* to wherever it is used. The definition of `sysctl_sched_nr_migrate` on line 7 depends on the macro `CONFIG_SCHED_DEBUG`, even though there is no explicit conditional. Garrido and Johnson first observed that a configuration-preserving preprocessor can expand such a multiply-defined macro to an explicit conditional, so as not to lose any configuration information [16].

Macro invocations. As already discussed, the preprocessor expands macro invocations to all definitions of the corresponding macros. Since macro invocations may be nested in macro definitions, the preprocessor needs to revisit the result of every macro expansion. Furthermore, since C compilers have several built-in object-like macros, such as `__STDC_VERSION__` to indicate the C Standard's version number, the preprocessor needs to be configured with the ground truth for the targeted compiler.

Beyond these fairly simple issues, a configuration-preserving preprocessor needs to handle two, more subtle interactions. First, some versions of a multiply-defined macro may be invalid at a particular invocation site, notably because the invocation is surrounded by conditionals creating an implicit presence condition. Consequently, the preprocessor should expand only valid definitions of a macro. SuperC uses a binary decision diagram [9] (BDD) library for this purpose. It facilitates the tracking of presence conditions and testing whether such conditions are equivalent or infeasible.

Second, function-like macro invocations interact with conditionals. Both macro name and arguments may contain conditionals, either explicitly in the source or implicitly through multiply-defined macros. These conditionals can alter the invoked macro, i.e., its name, and the provided arguments, including their number and values. For conditionals that modify the macro invocation (e.g., by changing the number of arguments), a configuration-preserving preprocessor needs to hoist the conditionals from within the invocation to the outside. Figure 3 illustrates this complication. If `__KERNEL__` is defined, line 6 defines macro `cpu_to_le32` to name another macro, `__cpu_to_le32` as defined on line 2. Other-

	Language & Granularity	Strategy	Complications				
			Surrounded by Conditionals	Contains Conditionals	Spans Conditionals	Contains Multiply-Defined Macros	Other
Lexer							
Layout	CPP Arbitrary	Annotate tokens					
Preprocessor							
Macro (Un)Definition	CPP One line	Use conditional macro table	Multiple entries in macro table			Expansion delayed until invocation	Remove redefinitions
Macro Invocation	CPP Many lines	Expand to all definitions	Expand to valid definitions only	Hoist conditionals around invocation	Hoist conditionals around invocation	Expand nested macros	Get ground truth for built-ins from compiler
Token Pasting & Stringification	CPP One line	Apply pasting & stringification		Hoist conditional around operations		Hoist conditional around operations	
File Inclusion	CPP One line	Include & preprocess	Preprocess under presence condition			Hoist conditional around inclusion	Reinclude when guard macro is not false
Conditionals	CPP Many lines	Preprocess tests & all branches	AND nested conditions	AND nested conditions		Hoist conditionals around expression	Preserve order for non-boolean expressions
Error Directives	CPP One line	Annotate tokens	Indicates infeasible branch				
Line, Warning, & Pragma Directives	CPP One line	Annotate tokens					
Parser							
C Constructs	C Arbitrary	Fork-merge LR parsing		Fork parsers			
Typedef Names	C One token	Use conditional symbol table					Fork parsers on ambiguous names

Table 1. The interactions between lexing, preprocessing, and parsing.

```
1 #define __gcc_header(x) #x
2 #define _gcc_header(x) __gcc_header(linux/compiler-gcc##x.h)
3 #define gcc_header(x) _gcc_header(x)
4 #include gcc_header(__GNUC__)
```

Figure 4. A computed include from include/linux/compiler-gcc.h.

wise, the former macro is undefined. Consequently, the invocation on line 10 may either be an invocation of macro `__cpu_to_le32` or of function `cpu_to_le32`.

Token pasting and stringification. Macros may contain two special operators to modify tokens: The infix token pasting operator (`##`) concatenates two tokens, and the prefix stringification operator (`#`) converts a sequence of tokens into a string literal. The preprocessor simply applies these operators, with one complication: Similar to macro invocations, the operators' arguments may contain conditionals, either explicitly in the source or implicitly through multiply-defined macros. Consequently, a configuration-preserving preprocessor needs to hoist the conditionals outside operator invocations. Figure 4 illustrates stringification on line 1 and token pasting on line 2. Figure 5 illustrates hoisting a multiply-defined macro out of the token pasting operator. When expanding the macro `DEFAULT_FETCH_TYPE`, the preprocessor ends up with `u##` being applied to a conditional, for the multiply-defined macro `BITS_PER_LONG`. It then hoists the conditional around the token pasting operator, yielding the conditional shown in part (b). Finally, it performs the token pasting, yielding a conditional with `u64` when `CONFIG_64BIT` is defined and `u32` otherwise.

File inclusion. To arrive at a complete compilation unit, the preprocessor resolves file inclusion directives (`#include`) and re-

```
1  // In include/asm-generic/bitsperlong.h
2  #ifdef CONFIG_64BIT
3  #define BITS_PER_LONG 64
4  #else
5  #define BITS_PER_LONG 32
6  #endif
7
8  // In kernel/trace/trace_kprobe.c
9  #define __DEFAULT_FETCH_TYPE(t) u##t
10 #define _DEFAULT_FETCH_TYPE(t) __DEFAULT_FETCH_TYPE(t)
11 #define DEFAULT_FETCH_TYPE _DEFAULT_FETCH_TYPE(BITS_PER_LONG)
```

(a) The macro definitions.

```
1 #ifdef CONFIG_64BIT
2 u##64
3 #else
4 u##32
5 #endif
```

(b) Hoisting `CONFIG_64BIT` outside the token pasting operator.

Figure 5. An example of hoisting a multiply-defined macro when expanding `DEFAULT_FETCH_TYPE`.

curses over header files. As for macro invocations, the preprocessor needs to track implicit presence conditions due to surrounding conditionals. Furthermore, it needs to correctly handle *guard macros*. By convention, header files protect against multiple inclusion with the incantation:

```
#ifndef FILENAME_H
#define FILENAME_H
```

If the guard macro `FILENAME_H` is undefined, the preprocessor needs to process the included header—even if the header has been included before and the guard macro has since been undefined. More interestingly, the include directive itself may invoke macros. As for macro invocations in general, token pasting, and stringification, a configuration-preserving preprocessor needs to hoist any conditional implicit in the macro outside the directive. Figure 4 illustrates such a *computed include* with the `gcc_header` macro. It relies on several other macros performing token pasting and stringification to arrive at a header file name containing the compiler's version number, as provided by the built-in `__GNUC__` object-like macro.

Conditionals. A configuration-preserving preprocessor needs to process all branches of a conditional and their respective tests, instead of just processing the first branch whose test resolves to true. If multiply-defined macros appear in a test, the configuration-preserving preprocessor needs to hoist the corresponding full conditional outside the directive. Finally, it needs to track the current presence condition, by performing a conjunction of nested conditions. This is straight-forward for boolean expressions by using BDDs (as done by SuperC) or even a more heavy-weight SAT solver (as done by TypeChef). However, conditional tests may contain arbitrary integral arithmetic and comparisons, such as `BITS_PER_LONG == 32`. Since there is no known efficient algorithm for comparing arbitrary polynomials [20], the preprocessor needs to treat such conditions as opaque and their branches akin to barriers in multi-threaded programming. It must never omit or combine them, it must preserve their source code ordering, and it must not move other branches across them.

Other preprocessor directives. The C preprocessor supports four additional directives, to issue errors (`#error`) and warnings (`#warning`), to configure compilers (`#pragma`), and to overwrite line numbers (`#line`). A configuration-preserving preprocessor simply reports errors and warnings appearing outside conditionals, and also terminates for such errors. More importantly, it treats conditional branches containing error directives as infeasible and disables their parsing. Otherwise, it optionally preserves such directives as token annotations to support automated refactorings.

C constructs, incl. typedef names. The third and final step is parsing. After configuration-preserving preprocessing, tokens may contain conditionals. Consequently, the parser needs to fork its internal state when reaching a conditional and merge it after the conditional again—but only when considering the same derivation of nonterminals for the same token. This *merge discipline* ensures that the resulting AST is well-formed, i.e., contains only complete syntactic units. Figure 1(b) illustrates this constraint. The conditional on lines 5–9 of the preprocessed source contains an incomplete syntactic unit due to the trailing `else`. As a result, the two subparsers recognizing the conditional and the implicit (and empty) `#else` branch can only merge after each processing line 10 and thus each recognizing a complete statement, preceded by a declaration (on lines 3) and within a function definition (on lines 1–13).

A final complication results from the fact that C syntax is context-sensitive [30]. Depending on context, names can either be typedef names, i.e., type aliases, or they can be object, function, and `enum` constant names. Furthermore, the same code snippet can have fundamentally different semantics, depending on names. For example,

```
T * p;
```

is either a *declaration* of `p` as a pointer to type `T` or an *expression statement* computing the product of `T` and `p`, depending on whether `T` is a typedef name. In the presence of conditionals, however, a name may be both. Consequently, the parser's symbol table for disambiguating such names now needs to store multiple versions and their presence conditions, just like the macro symbol table. Furthermore, when encountering an ambiguously declared name, the parser needs to implicitly fork its internal state.

3. Fork-Merge LR Parsing

Having systematically analyzed the problem and sketched the key ingredients of the solution in Section 2, we now turn to parsing C with conditionals between arbitrary tokens. We do not present configuration-preserving preprocessing in further detail, since its implementation follows directly from the description in the previous section and, while tedious to engineer completely and correctly, does not require algorithmic innovation. In contrast, parsing C with conditionals does require algorithmic innovation. To this end, we turn to LR parsing, more specifically LALR(1) parsing.

LR parsers are bottom-up parsers [2, 24]. To recognize their input, LR parsers maintain an explicit parser stack, which contains terminals, i.e., tokens, and nonterminals. On each step, LR parsers perform one of four actions: (1) *shift* to move the next token in the input onto the stack, (2) *reduce* to replace one or more topmost stack elements with a nonterminal, (3) *accept* to successfully complete parsing, and (4) *reject* to terminate parsing with an error. The choice of action depends on both input and parser stack. To ensure efficient operation, LR parsers encode their status in a single state, that is, combine a deterministic finite automaton (DFA) with a stack to form a push down automaton (PDA). Furthermore, for both actions and state transitions, they only consider the next k tokens in the input, which is called the *lookahead*. In practice, most such LR(k) parsers are LR(1) and, more specifically, LALR(1) [10]. The latter give up some expressivity of full LR(1), but also have more compact action and transition tables. Either way, the PDA implementation is the same, e.g., considers only the next token in the input.

Compared to top-down parsing techniques, such as LL [29] and PEG [6, 13], LR parsers are attractive for our purposes for four reasons. First, LR parsers make the parsing state explicit, in the form of PDA state and stack. Consequently, it is easy to fork the parser state on a conditional, e.g., by representing the stack as a singly-linked list and adding several stack frames that point to the shared remainder. Second, LR parsers are easier to implement efficiently, since they just operate PDAs. Third, LR parsers support arbitrary left-recursion in addition to right-recursion, which is helpful for writing programming language grammars. Finally, most of the complexity of LR parsing is isolated to table generation, which we do not modify at all. At the same time, table generation also tends to be somewhat brittle, with many a developer dreading obscure shift-reduce or reduce-reduce conflicts.

3.1 The Algorithm

Algorithm 1 shows the pseudo-code for the Fork-Merge LR (FMLR) parsing algorithm. Unlike regular LR parsers, which rely on a single PDA, FMLR parsers operate a set of *subparsers*, denoted by P. Each subparser p runs its own PDA, with $p.a$ denoting the next token for that subparser. The algorithm starts with a single subparser, initialized with the first token a_0. It then performs regular LR actions on subparsers with regular tokens (lines 6–16), while also forking on conditionals (lines 17–20) and merging whenever possible (lines 25–26). It ends when all subparsers have either accepted or rejected the input (lines 22–24). The set of subparsers P is *ordered* by their tokens' positions in the input. That way, the FMLR algorithm always advances the subparsers with the earliest token (lines 4–21) and thus ensures that no subparser moves beyond a position that would allow it to merge again. We do not show PDA states in the algorithm; they are updated in the regular LR manner by the SHIFT and REDUCE functions.

Algorithm 1 The Fork-Merge LR Parsing Algorithm

```
   a is a token.
   p is a subparser, with p.a denoting its next token.
   P is the set of subparsers ordered by input position.
 1: procedure PARSE(a_0)
 2:     P ← { initial subparser for a_0 }
 3:     loop
 4:         for all subparsers p in P on the earliest p.a do
 5:             if p.a is a regular token then
 6:                 ▷ Regular LR
 7:                 if action(p) is shift then
 8:                     SHIFT(p)
 9:                     p.a ← next token after p.a
10:                 else if action(p) is reduce then
11:                     REDUCE(p)
12:                 else if action(p) is accept then
13:                     Remove p from P
14:                 else ▷ Found a parsing error
15:                     Remove p from P
16:                 end if
17:             else if p.a is a compound token then
18:                 ▷ Fork subparsers on the conditional
19:                 P ← FORK(p) ∪ (P \ p)
20:             end if
21:         end for
22:         if P = ∅ then ▷ Done
23:             return
24:         end if
25:         ▷ Merge subparsers again
26:         MERGE(P)
27:     end loop
28: end procedure
```

Parsing regular tokens. For regular tokens, FMLR behaves just like regular LR (lines 6–16; highlighted in gray). The subparser p operates its PDA like a regular LR PDA by shifting, reducing, accepting, and rejecting. The only difference is that the implementation of the accept and reject actions only removes the subparser p from the set of all subparsers P instead of immediately stopping the entire parser. Still, if there are no conditionals in the input, FMLR performs LR parsing. As a direct result, an FMLR parser can reuse LR parser tables without modification. In turn, this enables us to reuse an existing LR parser table generator and an existing C grammar for SuperC.

Forking on conditionals. For conditionals, FMLR needs to fork the subparser p and replace it with the resulting parsers in P (lines 17–20). The forking itself is performed by simply creating several new subparsers p_i that have the same PDA state and stack as p, but have different new tokens $p_i.a$. As a result, all subparsers in P form a directed acyclic graph (DAG) of PDA stack fragments, comparable to Elkhound's GLR parser [26]. This is an efficient encoding of FMLR PDAs. It also is correct because, up to a FORK operation, all subsequently forked parsers have processed the same input in the same way, i.e., have the same PDA execution history.

The challenge is determining *which* new subparsers to create. Intuitively, FORK should create a new subparser for each branch of the conditional, which starts with the compound token $p.a$. However, conditionals may have empty branches, which, like the `#else` branch in Figure 1, may even be implicit. Furthermore, conditionals may be directly nested within conditional branches and they may directly follow other conditionals. To put it differently, the input is ambiguous and FORK needs to create a new subparser for each non-ambiguous path through the input—by determining each path's next regular token.

To this end, FMLR's FORK function computes FOLLOW$(p.a)$, the set of all *regular* tokens immediately following the compound token $p.a$ across all paths through the input. It then forks a parser for each distinct $a \in$ FOLLOW$(p.a)$. For completeness, the computation of FOLLOW is shown in Appendix A. It is comparable to the computation of follow sets for grammars [2], with two differences. First, instead of considering all alternatives of productions, it considers all branches of conditionals. Second, since there is no equivalent to nonterminals, it does not compute a fixed-point. Rather, it simply scans the input (without backtracking) until reaching a state where all possible paths through all conditionals have reached regular tokens.

Merging subparsers. After stepping and forking subparsers as well as checking for parser completion, FMLR attempts to merge subparsers (lines 25–26). Since FMLR's main loop on lines 3–27 only steps and forks subparsers at the earliest input position and then always invokes the MERGE function, the algorithm ensures that subparsers are, in fact, merged as soon as possible. The MERGE function itself must follow the *merge discipline* introduced at the end of Section 2: It may only merge subparsers that have the same derivation of nonterminals for the same position in the input. Finding candidate subparsers that are at the same position is trivial, since the set of subparsers P is already ordered by input position. Ensuring that candidate subparsers have the same derivation of nonterminals means comparing their PDAs, including their stacks. In practice, the comparison of PDA stacks never traverses the complete stacks. Since PDA stacks form a DAG, the stack comparison either reaches differing stack frames (and thus does not merge) or identical stack frames (and thus does merge).

Finding the next token. A final detail of the FMLR algorithm is how to move to the next token. As written, the algorithm assumes that a compound token on line 17 also starts a conditional (`#if` and `#ifdef`), even though compound tokens may also start conditional branches (`#elif` and `#else`) or end conditionals (`#end`). The FORK function already ensures that each forked subparser has a regular token as its next token, and thus does not violate the algorithm's compound token invariant. Getting the next token on line 9, however, must take compound tokens into account. If the next token in the input is a compound token starting a conditional branch or ending a conditional, then it must skip ahead to the token right after the conditional. Furthermore, due to nested conditionals, it needs to repeat this process—until the next token is either a regular token or a compound token starting a conditional.

3.2 Discussion

A naive implementation of the FMLR algorithm has performance exponential in the size of the input, even when it is neither necessary nor desired. For instance, consider the code in Figure 6, which conditionally initializes an array. It effectively encodes a binary number and a naive implementation of FMLR would fork an exponential number of subparsers, with each recognizing one such binary number. To avoid this blow-up for common coding idioms, such as conditional `struct` and `union` member declarations as well as array initializers, a performant implementation of FMLR needs to do three things. First, it needs to avoid recomputing FOLLOW for all but the first conditional in the chain of conditionals.

```
1  static int (*check_part[])(struct parsed_partitions *)={
2  #ifdef CONFIG_ACORN_PARTITION_ICS
3    adfspart_check_ICS,
4  #endif
5  #ifdef CONFIG_ACORN_PARTITION_POWERTEC
6    adfspart_check_POWERTEC,
7  #endif
8    // 16 more similar initializers
9    NULL
10 };
```

Figure 6. An example of a conditional encoding a binary number from fs/partitions/check.c.

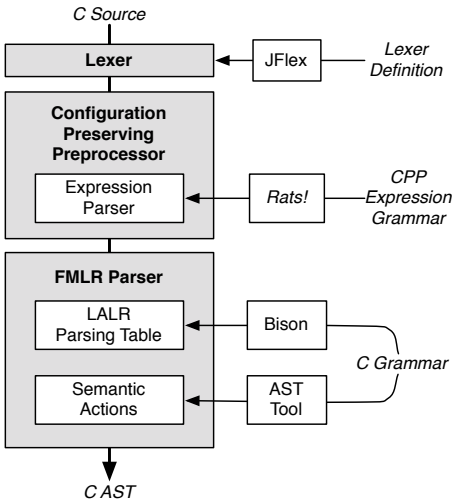


Figure 7. The SuperC architecture. The actual tool is shown on the left, while supporting tools and their inputs are shown on the right.

Second, it needs to delay the actual forking, until required by an LR operation. Third, it needs to recognize each conditional as a complete syntactic unit and merge subparsers after each conditional again. SuperC meets all three requirements and, in the above example, produces an AST that preserves the binary number through a list of 18 static choice nodes.

Since meeting the third requirement depends on the input, it is still possible to construct pathological inputs that cause exponential performance. This is an inherent limitation of *any* algorithm that completes syntactic units without memoizing all intermediate results. For instance, both SuperC and TypeChef need to parse line 10 of the preprocessed code in Figure 1(b) twice, once to complete the `if` statement starting on line 6 and once as a stand-alone statement. Additional, though comparatively minor sources of nonlinear performance for FMLR are the ordered set P and the MERGE function. Notably, a straight-forward implementation of an ordered set has $O(|P|)$ insertion time and $O(1)$ extraction time; though an implementation with $O(\log\log|P|)$ insertion time is possible [33]. Furthermore, the pairwise comparison of candidate subparsers for merging is $O(n^2)$ in the number n of candidates, with each comparison walking PDA stacks down to a single common frame in the worst case.

4. Pragmatics

Having presented the FMLR algorithm, we now turn to the SuperC tool and explore the pragmatics of building a real-world system. Figure 7 illustrates SuperC's architecture. Our tool implements all three steps for parsing all of C—lexing, preprocessing, and parsing itself—in Java. The lexer produces the initial stream of tokens; the preprocessor, in turn, produces a stream of tokens and compound tokens; and the parser produces a program's AST. The lexer is automatically generated by JFlex [23] from an existing lexer definition included with Roskind's C grammar [30], which we modified to support common gcc extensions, such as `__attribute__`. We engineered the configuration-preserving preprocessor from scratch, using the solution strategies identified in Section 2 and summarized in Table 1. Since the preprocessor needs to parse expressions contained in conditional directives, we reused a C expression grammar distributed with the *Rats!* parser generator [18]. This has enabled us to more quickly bootstrap SuperC. The parsing engine is our implementation of the FMLR algorithm. It accepts LALR parser tables for any language in the format produced by Bison [14]. Since Bison generates C headers, we rely on a very short C program to convert its tables to Java. Because of this, we can directly reuse Roskind's C grammar (modified to support gcc extensions). In addition to parser tables, SuperC's parsing engine also accepts semantic actions for building ASTs. They are automatically generated from grammar annotations by a small tool we developed.

In implementing FMLR, we faced three main engineering challenges, all of which are caused by the need to recognize an input while also generating an abstract syntax tree. (1) The parser needs to create static choice nodes when merging, and do so efficiently. (2) Downstream tools need an AST that is amenable to further processing. (3) The parser needs to support context-sensitive languages like C. To help address these challenges, our implementation of the FMLR parsing engine provides three extra components for each subparser besides the next token and PDA stack. First, it maintains the *presence condition* for each subparser, automatically updating it when entering and leaving conditionals. Second, it stores a *semantic value* on each stack frame, which is used for building the AST. Third, it provides each subparser with its own *parsing context*, which, in combination with semantic actions, enables the parsing engine to recognize context-sensitive languages. The remainder of this section discusses the three challenges in further detail, one subsection for each.

4.1 Merging Subparsers

While merging subparsers, SuperC's FMLR engine also creates static choice nodes. Each subparser's semantic value and presence condition becomes a child of the newly created static choice node, which, in turn, becomes the semantic value of the merged subparsers. In contrast to conditional tests in the (preprocessed) source, the presence conditions of static choice nodes are self-contained. As a result, downstream tools using SuperC's ASTs can determine a subtree's presence condition by simply finding the lowest enclosing static choice node, instead of walking from/to the top of the tree. Because semantic values contain only terminals or tree nodes resulting from the reduction of nonterminals, the generated AST contains only complete syntactic units—even when conditionals in the original source code do not. By default, SuperC treats all nonterminals in the grammar as complete syntactic units, which may be surrounded by static choice nodes. Since downstream tools may not be prepared to handle static choice nodes for each and every kind of AST node, SuperC also provides a simple grammar annotation facility that overrides this default; it is described in the next subsection.

Merging can be an expensive operation. The merge discipline requires that the parsing engine fulfills two merge criteria: (1) subparsers are at the same input position and (2) they are in the same parser state. A naive implementation would perform a pair-wise comparison of *all* subparsers to identify appropriate candidates, which is clearly wasteful. Instead, SuperC tries to merge only sub-

Name	Description
`layout`	Omit terminal from the AST.
`passthrough`	Use the only child's value.
`list`	Build a linear list for a recursive production.
`complete`	Treat as a complete syntactic unit.
`action`	Execute a developer-specified semantic action.

Table 2. Grammar annotations.

parsers on the earliest next token. After all, they are the only ones being stepped during the next iteration through the FMLR algorithm. Keeping subparsers in a priority queue facilitates efficient selection of such merge candidates. More specifically, SuperC implements the priority queue as a linked list, ordered by earliest next token. As a result, insertion performance is $O(n)$ in the length n of the queue, while access to subparsers at the earliest next token is constant. This is an appropriate trade-off: Insertion happens only when forking subparsers on conditionals, while access to the earliest subparsers happens during each iteration of the FMLR algorithm.

Even with fast access to the earliest subparsers, SuperC needs to enforce the second merge criterion by comparing parser states, including stacks. To further reduce the number of such pairwise comparisons, SuperC maintains a mapping from current PDA state number to subparsers. Since the number of states is reasonably small (about 900 for the C parser), the mapping is implemented as an array keyed by state number. Finally, even when performing a pairwise comparison, SuperC never walks complete stacks. As already described in Section 3, all subparsers are forked from a single, original subparser and their state stacks form a DAG.

4.2 Building Abstract Syntax Trees

Many parser generators and libraries, including Bison and TypeChef, require semantic actions for generating abstract syntax trees. To simplify AST construction, SuperC includes an annotation facility that eliminates the need for explicit semantic actions in most cases. Developers simply add a special comment of the form

`/**` *annotation*`,` *annotation*`,...` `**/`

next to the definition of a token or nonterminal. SuperC's AST tool then extracts these comments and instructs the parsing engine accordingly. Without annotations, the parsing engine builds a parse tree, containing all tokens as leaves and all nonterminals as inner nodes. Tokens form a statically typed class hierarchy, with the common superclass `Token` defining its interface. Nodes are instances of a single generic node class, which stores a name and list of children. Though, internally, that class utilizes further classes to optimize memory utilization for nodes with a small, fixed number of children.

Currently supported annotations are summarized in Table 2. The first three, `layout`, `passthrough`, and `list`, control how SuperC creates nodes for regular productions. First, `layout` instructs the parsing engine to treat tokens, such as those representing punctuation (`,`, `;`, and so on) and grouping (`{`, `}`, and so on), just like layout, i.e., omit them from the AST. Second, `passthrough` instructs the parsing engine to reuse a child's semantic value, if it is the only child in an alternative. Though other alternatives in the same production, which have more than one child, still produce AST nodes. This annotation is particularly useful for C expressions, whose productions are nested 17 levels deep to encode precedence. Third, `list` instructs the parsing engine to encode the semantic values of a recursive production as a linear list, instead of creating a recursive chain of tree nodes. It is helpful because LR grammars typically represent repetitions as left-recursive productions. SuperC implements such lists as pairs, comparable to functional languages. When forked subparsers add elements to the same list (started before forking), SuperC currently annotates individual list elements with their presence conditions. We are exploring alternatives that avoid this inelegant sharing of lists between subparsers.

The fourth annotation, `complete`, instructs the parsing engine to only treat productions with that annotation as complete syntactic units. This lets grammar writers limit the kinds of nonterminal values directly appearing in static choice nodes. The selection of complete syntactic units should be made carefully. Both the engine's default behavior (treating all nonterminals as complete) and `complete` annotations on too many productions result in ASTs with static choice nodes around too many kinds of constructs for downstream tools to handle effectively. Having too few complete syntactic units can result in an exponential state explosion even for common idioms. Section 3.2 illustrated this issue on the conditional array initializer shown in Figure 6. SuperC's C grammar tries to strike a balance by treating not only declarations, definitions, statements, and expressions as complete syntactic units, but also members in commonly configured lists, including `struct` and `union` member declarations, function parameters, and `struct` and array initializers.

As a final annotation, `action` instructs the parsing engine to execute an arbitrary semantic action. When reducing a nonterminal with an `action` annotation, the parsing engine calls out to a plug-in interface that allows external Java code to execute. Developers implement their own plug-in class (which implement the interface) and then configure the parsing engine with their code. That code has access to the corresponding subparser and can perform arbitrary operations, including building an AST node and updating the parsing context for context-sensitive languages.

4.3 Managing Parser Context

To support the recognition of C, SuperC's parsing engine associates each subparser with a context object. It also provides a conditional symbol table to determine whether names are typedef names or not and whether they are ambiguously defined. The symbol table is shared between the contexts of all subparsers. At the same time, each subparser's context separately tracks the current C language scope, which may differ across conditional branches. SuperC's C parser relies on semantic actions to update the symbol table when encountering C declarations and when entering and leaving the scope of functions, loops, etc. When forking, the engine makes a shallow copy of the context, leaving the symbol table untouched. When merging, the engine simply throws one of the contexts away, since it also makes sure to only merge subparsers in the same C language scope. Finally, when recognizing names that are ambiguously defined, it forks subparsers, thus ensuring that SuperC's C parser correctly handles this final complication introduced in Section 2.

We plan to refactor the current implementation, creating a plug-in interface for context management comparable to SuperC's existing plug-in interface for semantic actions. Developers, again, will implement their own plug-in class and then configure the parsing engine with their code. As in the current implementation, each subparser will reference the context, which is available to semantic actions. Furthermore, to support forking and merging of subparsers, the plug-in interface will contain four callbacks: (1) *fork context* to fork the current parser context, (2) *merge contexts* to merge contexts again, (3) *may merge* to determine whether two contexts allow merging, and (4) *should fork* to determine whether the current context requires forking, even if the subparser is not at the beginning of a conditional. These callbacks will then be automatically invoked by SuperC's parsing engine, thus providing a general solution for recognizing context-sensitive languages.

	Total	C Files	Headers
SLoC	7,916,995	85%	15%
All Directives	747,856	37%	63%
`#define`	520,054	19%	81%
`#if`, `#ifdef`, `#ifndef`	47,205	64%	36%
`#include`	129,291	88%	12%

Table 3. Preprocessor directives in the x86 Linux kernel source as compared to source lines of code.

Header Name	# C Files
include/linux/module.h	5,429
include/linux/slab.h	4,973
include/linux/init.h	4,044
include/linux/kernel.h	3,806
include/linux/delay.h	2,396
include/linux/interrupt.h	1,788
include/linux/errno.h	1,637
include/linux/types.h	1,584
include/linux/string.h	1,343
include/linux/platform_device.h	1,197

Table 4. The ten most commonly included x86 Linux header files.

5. Evaluation

This work's primary goal is to develop a general solution for parsing C with arbitrary preprocessor usage. To evaluate how well our work meets this goal, we consider two main factors. First, we empirically characterize the extent of preprocessor usage in the x86 Linux kernel. The objective is to provide quantitative evidence for preprocessor usage being extensive and complex—in addition to the anecdotal evidence already provided in previous sections. Second, we empirically evaluate SuperC's handling of the x86 Linux kernel. The objective is to provide experimental validation of SuperC's functionality. Notably, this entails quantifying SuperC's performance, while also comparing it to TypeChef's.

The focus of our study is the x86 Linux kernel because the kernel (a) is large and complex, (b) has many developers with different coding styles and skills, and (c) needs to meet considerable flexibility and performance requirements. Consequently, we expect Linux to provide us with a cornucopia of use cases for preprocessor and language interactions. In evaluating the x86 Linux kernel, we first present a brief *static* analysis of the source code as written, which reflects a developer's view of preprocessor usage. We follow with a more detailed *dynamic* analysis of preprocessor behavior, which reflects a tool's view of preprocessor usage. To this end, we utilize an instrumented version of SuperC and the latest stable version of the x86 Linux kernel (2.6.39.2). In evaluating SuperC, we focus on characterizing performance, since it implicitly validates its functionality. To ensure a fair comparison with TypeChef, this part of our evaluation relies on a slightly older version of the x86 Linux kernel (2.6.33.3); it is included with TypeChef's source distribution and has been manually prepared for processing by that tool.

In summary, our experimental evaluation demonstrates that (1) the x86 Linux kernel indeed makes extensive use of the preprocessor, with many of its features interacting with each other, and (2) SuperC shows reasonable performance, while TypeChef is not only slower but also has worse scalability.

5.1 Preprocessor Use in the x86 Linux Kernel

The Linux kernel can be configured to run across a wide range of operating conditions with conflicting requirements, ranging from embedded devices such as smartphones to massive server farms for cloud computing. The preprocessor plays a crucial role in enabling this flexibility. It provides concision, adds expressiveness to C, abstracts over hardware and compilation environment, and enables optimizations even when the compiler does not (e.g., inlining).

To capture the developers' view of preprocessor usage, we counted preprocessor directives in x86 Linux source files. Table 3 shows the results of this *static* analysis. It reports the total source lines of code (SLoC), the total number of preprocessor directives, and the number of macro definitions, conditionals, and includes, respectively. For each category, the table shows the absolute number and the fraction found in C files (`.c`) as well as header files (`.h`). Nearly 10% of all x86 Linux SLoC are preprocessor directives, a fairly substantial portion of the sources. Furthermore, most preprocessor invocations (63%) are found in header files. Header files, in turn, may include other header files (12% of all inclusions), resulting in long chains of dependencies. This result is not surprising, as C programming custom encourages placing common definitions into header files, i.e., use them as a poor man's module system. Table 4 corroborates our expectation by showing the ten most frequently included x86 header files. `module.h` is included in nearly half of all compilation units! Also consistent with expectation, header files provide most macro *definitions* (81%). At the same time, it is hard to statically count macro *invocations*. Macro invocations usually are in other files than their definitions. Macro invocations have the same basic appearance as regular C identifiers and function calls. And macro invocations may be nested within each other. In contrast, dynamic analysis can easily capture such invocations.

To capture a tool's view of preprocessor usage, we instrumented SuperC to count individual preprocessor operations and their interactions. Table 5 shows the results of this *dynamic* analysis; the table's organization loosely follows the summary of challenges in Table 1. Each row represents a preprocessor or C language feature, and each column shows the statistics for the feature and its interactions. Table entries characterize the distribution across `.c` files, i.e., full compilation units, with three percentiles: 50th · 90th · 100th. The data confirms that preprocessor usage is pervasive, and it demonstrates that many complications outlined in Section 2 do, in fact, occur. The most common complications are (1) nested macro invocations, (2) hoisting of conditionals, and (3) conditionals in C constructs. By comparison, computed includes are exceedingly rare and ambiguously-defined typedef names do not occur, likely because they lead to very confusing source code.

First, nested macro invocations, arising from macros appearing in other macros, are the most common complication. Furthermore, a comparison of the 50th percentile of total macro invocations (68k) with nested macro invocations (51k), both across all configurations, shows that most macros are invoked from within other macros. This makes it hard to statically analyze macro invocations, especially since macros, which look like C identifiers and function calls, may expand to entirely different C constructs. Second, hoisting is especially common for stringification, i.e., when a stringification argument is a conditional. Stringification is often used for optional debugging and tracing output, which likely explains why stringification hoisting occurs so often.

As shown in Table 5, compilation units contain thousands of conditionals. This raises the question of whether recognizing C code across conditionals is even feasible. Two factors determine feasibility: (1) the breadth of conditionals, which forces the forking of parser states, and (2) the incidence of partial C constructs in conditionals, which prevents the merging of parser states. The number of subparsers per iteration of FMLR's main loop (lines 3–27 in Figure 1) precisely captures the combined effect of these two factors. Consequently, Figure 8 shows the cumulative distribution

	Total	Interaction with Conditionals		Other Interactions	
Macro Definitions	29k · 41k · 102k	contained in	28k · 41k · 102k	redefinitions	307 · 462 · 3,360
Macro Invocation	68k · 96k · 251k	trimmed hoisted	10k · 15k · 39k 88 · 162 · 718	nested invocation built-in macros	51k · 72k · 186k 134
Token Pasting	3k · 4k · 24k	hoisted	0 · 0 · 35		
Stringification	5k · 8k · 20k	hoisted	713 · 1,141 · 3,190		
File Inclusion	3k · 5k · 12k	hoisted	27 · 45 · 90	reincluded headers	735 · 1,298 · 4,138
Conditionals	7k · 10k · 25k	hoisted max depth	120 · 162 · 419 29 · 33 · 42	non-boolean subexpressions	12k · 16k · 25k
Error Directives	32 · 44 · 131				
C Statements & Declarations	53k · 73k · 181k	FMLR	See Figure 8		
Typedef Names	687 · 966 · 2,595			ambiguously-defined typedef names	0 · 0 · 0

Table 5. A dynamic analysis of preprocessor operations and interactions. Entries show the per-file distribution at three percentiles: 50th · 90th · 100th.

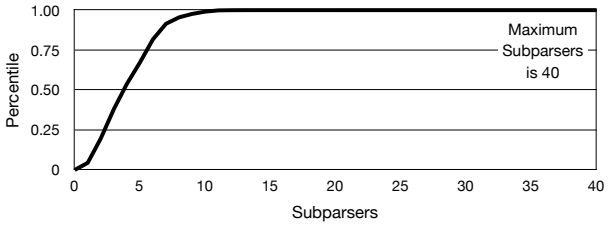


Figure 8. The number of subparsers used while parsing the x86 Linux kernel as a cumulative distribution of parser loop iterations.

of subparsers for the x86 Linux kernel—but not per `.c` file, as the other data in this section, but per iteration of FMLR's main loop. While FMLR needs a maximum of 40 subparsers, the parsing engine rarely uses more than 10 subparsers, the 99th percentile. Considering the potential for exponential parser state explosion, this result is extremely encouraging. It demonstrates that recognizing C code across all conditionals is practical, even for a system as large and flexible as Linux.

5.2 Performance of SuperC and TypeChef

To evaluate SuperC, we compare our tool's performance with that of TypeChef when processing Linux. To ensure a fair comparison, we reuse TypeChef's testing framework, included with its open source release, and test case, version 2.6.33.3 of the x86 Linux kernel, as prepared by its developers. We already validated SuperC's functionality on the latest stable version of the x86 Linux kernel, when collecting the statistics presented in the previous subsection. Since SuperC was performing fine-grained data collection and reporting, in addition to lexing, preprocessing, and parsing kernel sources, we do not report performance numbers for that version of the kernel.

To run either SuperC or TypeChef, some preconfiguration is necessary. As already discussed in Section 2, both tools need to be configured with the targeted compiler's built-in object-like macros. In addition, TypeChef's testing harness preconfigures parts of the kernel to limits its variability. It includes over 300 manually predefined macros. 30 of these provide values for non-boolean macros and 2 of these replace such macros with multiply-defined boolean equivalents; this is necessary, since TypeChef's support for non-

	Per-File Centile (Sec.)			
	50th	90th	100th	Total (Hrs.)
SuperC	5.8	6.6	14.2	12.4
TypeChef	17.9	26.5	652.2	45.3

Table 6. Comparison of SuperC and TypeChef's per-file latency for the x86 Linux kernel with 7,665 C files.

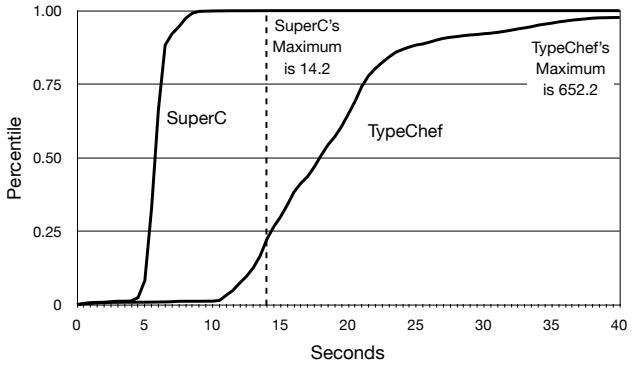


Figure 9. Comparison of SuperC and TypeChef's per-file latency for the x86 Linux kernel as a cumulative distribution.

boolean macros is incomplete. Furthermore, TypeChef assumes that only macros with the `CONFIG_` prefix may be variable, treating all other macros (which have neither been predefined nor start with the prefix) as undefined. For our performance comparison, SuperC reuses TypeChef's predefined macros, but otherwise treats all other macros as variable. For the previous subsection's experiment, we made no such predefinitions—with one exception: The x86 Linux kernel includes a macro, whose integral value is token pasted to the integer suffix `UL` when appearing within a C construct. We did predefine that single macro (and compiler built-ins) for all experiments. As a final preparation step, we manually removed all statements that print debugging information from both tools.

Table 6 summarizes our performance results and Figure 9 plots the cumulative distribution of per-file latencies. As for our dynamic

analysis of Linux preprocessor usage, we only consider `.c` files, i.e., complete compilation units. While TypeChef's testing harness spawns a fresh Java virtual machine (JVM) instance for each file, we do not believe this skews the results. In our measurements with a no-op program, JVM startup and teardown takes only 0.04 seconds on average. When considering the 50th and 90th percentiles of per-file latency, both tools perform reasonably well. While SuperC is between 3.1 to 4.0 times faster than TypeChef, both tools show a mostly linear increase in cumulative distribution, which is consistent with a normal latency distribution. However, the plateau in TypeChef's cumulative distribution at about 23 seconds and the subsequent long tail (up to almost 11 minutes!) indicate pathological behavior on a substantial number of compilation units. TypeChef's authors attribute this behavior to files with a large number of headers being conditionally included, which results in especially large presence conditions [21]. In contrast, SuperC is not only faster, but its performance is also consistent: it scales across the different compilation units and their different preprocessor feature interactions.

6. Related Work

Our work joins a good many attempts at solving the problem of parsing C with arbitrary preprocessor usage [1, 3–5, 12, 16, 21, 22, 25, 27, 31, 35]. However, only our own work and TypeChef [21, 22] provide a conceptually complete solution, even if TypeChef's implementation does not (yet) handle all complications. Since we already provided a detailed comparison between SuperC and TypeChef in Section 2 and 5, we focus on the other efforts in this section.

Previous, and incomplete, work on recognizing all of C can be classified into three categories. First are tools, such as Xrefactory [35], that process source code one configuration at a time, after full preprocessing. This approach is also taken by Apple's XCode IDE [7]. However, due to the exponential explosion of the configuration space, this is only practical for small source files with little variability. Second are tools, such as CRefactory [16], that employ a fixed but incomplete algorithm. This approach is also taken by the Eclipse CDT IDE [17]. It is good enough—as long as source code does not contain idioms that break the algorithm, which is a big if for complex programs such as Linux. Third are tools, such as Yacfe [27], that provide a plug-in architecture for heuristically recognizing additional idioms. However, this approach creates an arms race between tool builders and program developers, who need to push both preprocessor and C language proper to wring the last bit of flexibility and performance out of their code—as amply demonstrated by Ernst et al. [11], Tartler et al. [32], and this paper's Section 5.

Considering parsing more generally, our work is comparable to efforts that build on the basic parsing formalisms, i.e., LR [24], LL [29], and PEG [6, 13], and seek to improve expressiveness and/or performance. Notably, Elkhound [26] explores how to improve the performance of generalized LR (GLR) parsers by falling back on LALR for unambiguous productions. Both SDF2 [8, 34] and *Rats!* [18] explore how to make grammars modular by building on formalisms that are closed under composition, GLR and PEG, respectively. *Rats!* also explores how to speed up PEG implementations, which, by default, memoize intermediate results to support arbitrary back-tracking with linear performance. Finally, ANTLR [28] explores how to provide most of the expressivity of GLR and PEG, but with better performance by supporting variable look-ahead for LL parsing.

At a finer level of detail, Fork-Merge LR parsing relies on a DAG of parser stacks, just like Elkhound, but for a substantially different reason. Elkhound forks its internal state to accept ambiguous *grammars*, while SuperC forks its internal state to accept ambiguous *inputs*. Next, like several other parser generators, SuperC relies on annotations within the grammar to control AST building. For instance, ANTLR, JavaCC/JJTree [19], *Rats!*, SableCC [15], and SDF2 provide comparable facilities. Finally, many parsers for C employ an ad-hoc technique for disambiguating typedef names from other names, termed the "lexer hack" by Roskind [30]. SuperC's context management is currently also hard-coded in the parser engine; though we plan to refactor the implementation to provide a more principled solution, similar to that provided by *Rats!*.

7. Conclusion

This paper explored how to parse all of C. First, we systematically explored the challenges posed by interactions between C preprocessor and language proper. Our anecdotal and empirical evidence from the x86 Linux kernel demonstrates that meeting these challenges is not just an academic exercise. Linux represents a treasure trove of preprocessor (ab)use. Second, we outlined the functionality of a configuration-preserving preprocessor and presented a novel parsing algorithm, Fork-Merge LR parsing. The algorithm recognizes languages with static conditionals appearing between arbitrary tokens. To this end, it forks subparsers at the beginning of conditionals and merges them again when reaching the same state for the same input position. That way, it produces a well-formed AST, which contains only regular nodes for language constructs and static choice nodes for conditionals. Since Fork-Merge LR parsing incorporates LR parsing as its core, it enables the direct reuse of existing grammars and parse table generators, including for recognizing other preprocessed languages. Third, we presented the SuperC tool, which implements our preprocessing and parsing techniques, and demonstrated its effectiveness on the x86 Linux kernel. In summary, forty years after C's invention, we finally lay the foundation for efficiently processing *all of C*.

References

[1] B. Adams, W. de Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. 8th ACM International Conference on Aspect-Oriented Software Development*, pp. 243–254, Mar. 2009.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, Aug. 2006.

[3] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke. Re-engineering C++ component models via automatic program transformation. In *Proc. 12th IEEE Working Conference on Reverse Engineering*, pp. 13–22, Nov. 2005.

[4] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software: Practice and Experience*, 30(8):907–924, July 2000.

[5] I. D. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. 8th IEEE Working Conference on Reverse Engineering*, pp. 281–290, Oct. 2001.

[6] A. Birman and J. D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, Aug. 1973.

[7] R. Bowdidge. Performance trade-offs implementing refactoring support for Objective-C. In *Proc. 3rd ACM Workshop on Refactoring Tools*, Oct. 2009.

[8] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proc. 2004 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, Oct. 2004.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[10] F. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, Oct. 1982.

[11] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec. 2002.

[12] J.-M. Favre. Understanding-in-the-large. In *Proc. 5th IEEE International Workshop on Program Comprehension*, pp. 29–38, Mar. 1997.

[13] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pp. 111–122, Jan. 2004.

[14] Free Software Foundation. Bison. `http://www.gnu.org/software/bison/`.

[15] É. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Mar. 1998.

[16] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proc. 21st IEEE International Conference on Software Maintenance*, pp. 379–388, Sept. 2005.

[17] E. Graf, G. Zgraggen, and P. Sommerlad. Refactoring support for the C++ development tooling. In *Companion 22nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 781–782, Oct. 2007.

[18] R. Grimm. Better extensibility through modular syntax. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, pp. 38–51, June 2006.

[19] java.net. JJTree reference documentation. `http://javacc.java.net/doc/JJTree.html`.

[20] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In *Proc. 35th ACM Symposium on Theory of Computing*, pp. 355–364, June 2003.

[21] C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial preprocessing C code for variability analysis. In *Proc. 5th ACM Workshop on Variability Modeling of Software-Intensive Systems*, pp. 127–136, Jan. 2011.

[22] 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. In *Proc. 2011 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2011.

[23] G. Klein, S. Rowe, and R. Décamps. JFlex: The fast scanner generator for Java. `http://jflex.de/`.

[24] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec. 1965.

[25] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proc. 10th European Software Engineering Conference*, pp. 21–30, Sept. 2005.

[26] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. 13th International Conference on Compiler Construction*, vol. 2985 of *Lecture Notes in Computer Science*, pp. 73–88, Mar. 2004.

[27] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. 18th International Conference on Compiler Construction*, vol. 5501 of *Lecture Notes in Computer Science*, pp. 109–125, Mar. 2009.

[28] T. Parr and K. Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proc. 32nd ACM Conference on Programming Language Design and Implementation*, pp. 425–436, June 2011.

[29] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proc. 1st ACM Symposium on Theory of Computing*, pp. 165–180, May 1969.

[30] J. Roskind. Parsing C, the last word. The comp.compilers newsgroup, Jan. 1992. `http://groups.google.com/group/comp.compilers/msg/c0797b5b668605b4`.

[31] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, Nov. 2003.

[32] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. 6th European Conference on Computer Systems*, pp. 47–60, Apr. 2011.

[33] M. Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6), Dec. 2007.

[34] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[35] M. Vittek. Refactoring browser with preprocessor. In *Proc. 7th IEEE European Conference on Software Maintenance and Reengineering*, pp. 101–110, Mar. 2003.

A. The FOLLOW Set

Algorithm 2 Compute the set of regular tokens immediately following a compound token starting a conditional.

```
1:  procedure FOLLOW(a)
2:      result ← ∅
3:      ▷ FIRST assumes well-nested conditionals. Its result
4:      ▷ indicates whether to keep on following.
5:      procedure FIRST(a)
6:          loop
7:              if a is a regular token then
8:                  result ← {a} ∪ result
9:                  return false
10:             else if a ends a branch then
11:                 return true
12:             else ▷ a starts a conditional
13:                 cont ← false
14:                 for all branches b in the conditional do
15:                     if FIRST(token after one starting b) then
16:                         cont ← true
17:                     end if
18:                 end for
19:                 if conditional has no #else then
20:                     cont ← true
21:                 end if
22:                 if not cont then ▷ No branch is empty
23:                     return false
24:                 end if
25:                 a ← next token after conditional
26:             end if
27:         end loop
28:     end procedure
29:     loop
30:         if not FIRST(a) then ▷ Done
31:             return result
32:         end if
33:         ▷ Get next token while stepping out of conditionals
34:         repeat
35:             if a is a regular token then
36:                 a ← next token in input
37:             else ▷ a is a compound token
38:                 a ← next token after conditional
39:             end if
40:         until a does not end a branch
41:     end loop
42: end procedure
```