A Framework for Preprocessor-Aware

C Source Code Analyses

Greg J. Badros and David Notkin

Dept. Computer Science & Engineering

University of Washington

Box 352350, Seattle WA 98195-2350 USA

+1-206-543-1695

{gjb,notkin}@cs.washington.edu

14 November 1999

Abstract

Analyses of C source code usually ignore the C preprocessor because of its complexity. Instead, these analyses either define their own approximate parser or scanner or else they require that their input already be preprocessed. Neither approach is entirely satisfactory: the first gives up accuracy (or incurs large implementation costs), while the second loses the preprocessor-based abstractions. We describe a framework that permits analyses to be expressed in terms of both preprocessing and parsing actions, allowing the implementer to focus on the analysis. We discuss an implementation of such a framework that embeds a C preprocessor, a parser, and a Perl interpreter for the action "hooks." Many common software engineering analyses can be written surprisingly easily using our implementation, replacing numerous ad-hoc tools. The framework's integration of the preprocessor and the parser further enables some analyses that otherwise would be especially difficult to perform.

Keywords

Parsing, preprocessor, C, lexical analysis, syntactic analysis, Perl.

Introduction

More than twenty years ago, Dennis Ritchie designed the C language [1] to include a textual macro preprocessor called cpp [2, Ch. 3]. Given the simplicity of the language and the state of the art in compiler technology in the mid-1970s, his decision to provide some language features in this extra-linguistic tool was justified. For the last two decades, C programs have exploited cpp's capabilities for everything from manifest constants and type-less pseudo-inline functions to modularization and symbol generation. Bjarne Stroustrup, the designer and original implementor of C++, notes that "without the C preprocessor, C itself ... would have been stillborn" [3, p. 119]. Certainly cpp contributes to C's expressiveness and portability, but perhaps at too large a cost. Stroustrup recognizes this tradeoff:

> Occasionally, even the most extreme uses of cpp are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and tool builders [3, p. 424].

Why **cpp** is good ... *and* bad

The intrinsic problem with cpp is also its fundamental strength: it is a distinct first pass of textual (non-syntactic) processing over the source code. This introduces significant differences between the code that the programmer sees and what the compiler proper (i.e., the C compiler distinct from the preprocessor) ultimately compiles.[1]

Experienced and novice C programmers alike are frustrated by misunderstandings of source code due to the arbitrary transformations cpp performs. A software engineer might want to be sure that a function foo is not invoked in a given block of code. By simply visually scanning that code segment, though, she may

[1] To avoid ambiguity, we will use *unprocessed* to refer to the original source code, and *preprocessed* to refer to the view of the code after running cpp on it.

be oblivious to a seemingly-unrelated macro that happens to expand to an invocation of foo, the function intended to be avoided.

Such confusions are easily reduced, if not eliminated, by allowing the software engineer to see the code exactly as the compiler does. Unfortunately, that view of the program is at a level of abstraction lower than the unprocessed source. Well-known identifiers such as stderr may appear as the far less readable (_IO_FILE*)(&_IO_stderr_), and useful encapsulations such as assert degenerate into sequences of symbols that are less meaningful to a human programmer. Every non-trivial C program uses the preprocessor, and an empirical study of numerous packages written in C documents the extensive use of cpp constructs [4].

Though cpp is sometimes used by other languages (e.g., Java programs and the X11 xrdb resource database language) and causes similar difficulties whenever used, we focus on cpp's use with C code exclusively. We are interested in the huge body of C programs, and helping programmers understand and extend those software artifacts. cpp is a necessary and integral part of the C programming language specification. In practice, the C preprocessor is almost never separated from C code, thus the issues relating the cpp arise with virtually all C programs. Though we focus on C, the framework we describe could be applied to analyze other uses of cpp (and even other textual preprocessing languages) in a fairly direct way.

cpp and Software Tools

Because of the preprocessor's textual foundations, unprocessed C source code cannot be parsed directly. For example, identifiers that are macro-expanded may hide braces, parentheses, or other important tokens from the parser. Only after preprocessing is a C program in a grammatically usable form. Since parsing preprocessed code is relatively easy and well-understood, most software engineering tools operate on exactly that view of the source, losing abstractions that are expressed in cpp.

Various tools including source-level debuggers and call graph extractors either run cpp as the first stage in their analysis, or use representations derived from a compiler operating on the preprocessed code. For example, Siff and Reps implement a function-generalization transformation tool that operates on preprocessed code, but note that future versions of their tool must drop that requirement so that macro abstractions are not lost [5]. Using preprocessed code for program understanding or transformation tasks is fraught with difficulties due to changes in the source artifact from preprocessing.[2]

Another disadvantage of preprocessing is that it eliminates conditional compilation directives that are essential to the portability and versatility of the source code [6]. Preprocessing forces tools to limit their analysis to a single configuration of the source code, instead of permitting global reasoning about the entire artifact.

Some tools instead operate on the unprocessed source code exactly as the programmer sees it. This technique improves robustness in handling syntax errors and language variants. Because the input is unprocessed, the extracted information is presented to the human software engineer at the same level of abstraction as the source. Additionally, the unprocessed source still contains the preprocessor directives that are essential to the portability and flexibility of many C programs.

However, these tools cannot use a straightforward parser or reliably construct an abstract syntax tree because the source is not grammatically-correct C code. Lexical tools such as `etags`, LSME [7], and `font-lock-mode` for Emacs and approximate parsers such as `Field` [8] and `LCLint` [9] use this approach. However, disregarding or only partially honoring the `cpp` directives leads to an extracted model of the source code that is only an approximation of the program's appearance to a compiler. Certain uses of the preprocessor can cause substantial problems: the syntax of declarations or scoping constructs can be customized and arbitrary code can be hidden in macro expansions. Macro expansion was a major cause of both false positives and false negatives in call graph extraction [10]. Such approximate tools are inappropriate for software engineering analyses that require exact or conservative information.

We introduce a new approach that integrates the C preprocessor and a parser in a flexible framework for statically analyzing C source code. The framework makes it easy for tool builders to produce analysis tools for C source code that are capable of reasoning about the preprocessor.

[2] In contrast, this approach is exactly right for the compiler, where there is no need to preserve high level abstractions while generating object code.

The Framework

The idea behind our framework is simple: provide an integrated parser and preprocessor that controls the scanning of the source code while executing user-defined callbacks when specified "interesting" actions occur. Though similar to the way the `yacc` parser [11] associates actions with parse rule invocation, our framework provides callbacks on both parser and preprocessor actions. Additionally, instead of C as the language for the actions, we use the Perl scripting language for writing the hook subroutines. Griswold, Atkinson and McCurdy note that various software tools benefited from using a special-purpose action language [10], and interpreted languages can often significantly speed development time [12]. Additionally, Perl is sufficiently similar to C that it is easy to learn by C programmers. Perl also provides closures which make expressing and registering the callback routines very concise and readable. The nearly-immediate turn-around time that Perl provides is especially useful when prototyping and debugging new analyses. Our prototype implementation of the framework is called PCp^3, pronounced "pee-see-pee-cubed."

Callbacks can be installed on actions such as the scanning of preprocessor directives, the creation of a macro definition, the expansion (i.e., use) of a macro name, and the parsing of a variable declaration. Each action callback is passed arguments relevant to the event for which it was invoked. For example, the `TYPEDEF` action receives the name of the declared type as its first argument. See the first appendix for a more complete list of the hooks in our current implementation of the framework.

To supplement the directly-passed arguments, action subroutines may also invoke "backcalls" to access internal parser and preprocessor data structures. Example backcalls include getting the name of the currently-processed file, looking up a symbol in the parser's symbol table, inserting arbitrary code for the preprocessor to process, and instructing the parser to enter a new scope. See the second appendix for a more complete list of the backcalls in our current implementation of the framework.

The callback and backcall interfaces combined with a general purpose scripting language provide a concise and flexible means for easily writing static analyses of C source code. The framework itself is extensible with additional callbacks and backcalls by re-compiling the analysis tool after inserting the new callback invocations and backcall functions. Extending the framework is intended to be a far less frequent need than using the framework to perform a new analysis, so the additional developer time in writing, compiling, and testing C code extensions is acceptable.

Example: Call-Graph Extraction

One common software engineering analysis is to extract a call graph from a body of C source code. Our framework permits this analysis to be written in about ten lines of code, as shown in Figure 1.

To perform the analysis, we attach subroutines to three parsing callbacks. When a function specifier (that is, the signature specification of a function definition) is parsed, the `FUNC_SPEC` hook is activated. That callback resets the set of functions invoked by the current function. For each function call parsed, the `FUNC_CALL` hook is activated, and we add the name (passed to the subroutine as the zeroth argument) of the called function to the set. Finally, when the entire function definition has been parsed, the `FUNCTION` reduction occurs and we output the set of called functions we accumulated while scanning the body of the function definition.

The script does not implement the main control flow of the analysis. Also, it need not handle any preprocessor directives. Since this simple analysis deals only with parser actions, the view the analysis sees is exactly the same as if we had analyzed the preprocessed code.

Revising the Extractor

The simple call-graph extractor used as an example in the preceding section suffers from many of the same shortcomings as tools that operate on preprocessed code: it reasons about only a single configuration of the source and eliminates all macro abstractions from the extracted model. As Murphy et al. discuss, there are many degrees of freedom for call-graph extractors [13]. As written, the analysis counts all function calls in macro expansions as occurring from the function in which the macro is expanded—the macro expansion itself is omitted from the call graph. This behaviour may not always be what the software engineer desires. However, a substantial strength of our framework is that it lets a tool-builder easily fine-tune the extraction to derive the desired view.

For example, often macros are used to express inline functions. If we choose, we can have macro expansions included in the extracted call-graph:

```
# $_[2] is the macro name being expanded
AddHook("EXPAND_MACRO",
     sub { $func_calls{$_[2]}++; }
```

Given this revised extractor, a macro that expands into code that includes function calls will expose those function calls as callees from the function definition in which the macro was expanded. For some tasks, this may be exactly what we want. If instead we prefer to hide those nested calls, our framework supports that behaviour as well. We simply ignore function calls that we parse while expanding macros (see Figure 2).

Another possible extension of the analysis is to query the symbol table for the type of the identifier being called. If the variable is a pointer to a function, the enclosing function could conservatively be marked as "possibly calling all functions" (i.e., $\perp$).[3]

Because the activations of the preprocessor and parser hooks are intermingled, the analysis can reason about the preprocessed version of the source within the context of how it was preprocessed.

Preprocessor and Parser Interactions

Given a straightforward preprocessor and parser front end, some analyses are impossible. For example, since the preprocessor will skip code as instructed by conditional compilation directives, that source will be left unprocessed. If our analysis is intended to find all invocations of a function, we will often want conditional-compilation branches that are unused for the current platform to still undergo the analysis (as they would by a lexical tool such as `grep`, but unlike the behaviour achieved by operating on preprocessed code).

To permit handling the otherwise-skipped code hidden by conditional compilation directions such as `#ifdef` and `#if`, our framework provides a general mechanism for inserting arbitrary source text to be processed. Action subroutines can pass a string to the `PushBuffer` backcall. A hook, `DO_XIFDEF`, is called for all conditional compilation directives and provides an argument that is the text that would *not* be included. The action routine for that hook may then use `PushBuffer` to ask the preprocessor and parser to use that string as input and process it normally.

[3] A more aggressive analysis could maintain data on which functions are ever assigned to the variable throughout the program.

However, since such program text can include arbitrary preprocessor directives and C code, we may want to ensure that the state of neither the preprocessor nor the parser is permanently changed after the "for analysis only" parsing of the skipped code. Three data structures must be preserved to avoid side-effects to the main processing (i.e., the version selected by the #definedness of macro names): the preprocessor's hash table of macros, the C symbol table, and the parser's current stack of states.

The backcalls PushHashTab and PopHashTab save and restore the preprocessor's table of definitions so that preprocessor directives in skipped code will not affect cpp when normal processing resumes. For the symbol table, the backcalls EnterScope and ExitScope provide similar functionality. Finally, YYPushStackState and YYPopStackState save and restore the stack of parser states. In under fifty lines of boiler-plate code, an analysis tool can manage these complexities and expose the full source to the analysis. This template code is included with PCp^3.

Other related backcalls provide additional support for querying and interacting with the parser. ParseStateStack returns the list of states in the parser's stack; this exposes information about what constructs might be legal at the current location (for example, determining whether declarations may be permitted). SetStateStack permits explicitly changing the parser's stack of states, perhaps to prepare for handling a top level construct or to re-parse text using a different configuration of the parser. Using YYFCompareTopStackState, an action hook can efficiently check whether a previously saved parser state matches the current configuration. For example, an analysis can easily provide a warning whenever the "if" block and the "else" block of an #ifdef directive leave the parser in different states.

One minor complexity with analyzing both paths of a conditional compilation directive involves the #include directive. Often header files that might recursively include themselves are protected against entering an infinite inclusion loop by being enclosed in an #ifndef that skips the entire file when re-included:

```
#ifndef FOO_H__
#define FOO_H__


/* body of foo.h header */
#include "header-that-includes-foo.h"
```

```
#endif
```

If an analysis were always to evaluate the code inside the #ifndef, it would repeatedly and inifinitely re-analyze foo.h. To avoid this difficulty, an analysis can simply choose to not include a header file more than once. The DO_INCLUDE callback permits the software engineer to return FALSE to signify that the inclusion directive should be ignored. If an analysis is using PushBuffer to process code that would otherwise be skipped, then only the first attempt to include a given file should be allowed:

```
sub do_include {
  my (undef,undef,undef,$file_as_resolved,undef) = @_;
  return FALSE if ($already_included{$file_as_resolved});
  $already_included{$file_as_resolved} = $true;
  return TRUE;  # permit the inclusion
}
```

Once the file is included, the analysis continues from the new file. When that file is completely analyzed, the DONE_INCLUDE_FILE callback is invoked. This permits the analyses to track file inclusions and perform any necessary cleanup or per-header-file processing.

Further Examples

Preprocessor-specific analyses are generally especially difficult to write. The complexities and subtleties of cpp must be duplicated in each tool. Not surprisingly, our framework is ideal for analyzing preprocessor constructs.

Macro Expansion Mappings

To support useful interactions between the parser and preprocessor, it is essential that PCp^3 maintain an accurate mapping between the unprocessed source and the preprocessed source. Macro expansions are the most complicated aspect of this correspondence. Macro arguments can themselves be macros, and macros can expand to other macros in need of expansion.[4] To effectively exploit the macro expansion hooks, the details of the expanding and substituting process must be understood. For gcc's preprocessor, and consequently for PCp^3, macro expansion takes place as follows:[5]

1. The macro definition's body is checked to see which arguments it uses.

2. Those actual arguments that are used in the definition body are expanded if and only if they contain macros. They are expanded completely (i.e., macros in their expansions are expanded), and the text of the expansion is saved. Identical arguments are expanded independently; for example FOO(BAR,BAR) will expand BAR twice if the expansion of FOO uses both of its arguments. They are expanded in the order that they syntactically appear in the body of the expansion (not from first formal argument to last).

3. The body of the top level macro is copied left to right; arguments are replaced with the text from their expansions. Macro names previously expanded are escaped (using a prefix of the distinguished symbols "@-") in this pseudo input buffer to prevent recursion.

4. That entire text is rescanned, and un-escaped macros are expanded further.

The EXPAND_MACRO hook is called for each macro name as it is expanded. The parameters to the hook include the exact location of the start and end of the macro invocation in the source code.[6] Other parameters describe the "nesting" of the expansion, and a backcall MacroExpansionHistory() describes the current history of expansions. The nesting of an expansion is the trail through arguments of other macros that led to this expansion, while the expansion history is a list of macros that were expanded en route to this macro being expanded.

[4] In ANSI C's preprocessor, recursion is prohibited; as a macro name is expanded, that name is disabled from future expansions generated by the original expansion.

[5] This description is necessarily implementation specific. The C language standard provides details of what is required [2, Ch. 3]. Also note that details of stringization and pasting are omitted as they are infrequently used features [14].

[6] Or, if the invocation does not directly appear in the source (i.e., the macro appears in the expansion of another macro), the location is an offset within the prior macro's expansion.

Consider the example illustrated in figures 3 and 4. When the MACRO_EXPAND hook is called for FOO (marked with an asterisk in the figures), we have:

```
@nests                  == ( TA3#1; TA4#1 )
MacroExpansionHistory()  == ( PA2#Body )
```

The @nests list tells us we are expanding the first argument of macro TA3, which itself was the first argument of macro TA4. The MacroExpansionHistory() backcall provides the remaining information about this expansion: that the expansion came from the body of the earlier expansion of macro PA2. From Figure 4, the @nests list for a given expansion corresponds to the sequence of dotted lines directly above and completely overlapping the dark solid line representing that expansion. Similarly, the list MacroExpansionHistory() returns can be visualized as the stack of dark solid expansion lines directly above the expansion in question (those that have not already been paired with a light solid line representing the completion of their expansion).

This macro-expansion analysis generates a large amount of information that is not easy to comprehend in raw form. To support visualizing these data we provide a Perl module of hook utilities that outputs character-indexed annotations of the source code. These annotations are Emacs Lisp source code that manipulate text properties of character ranges when evaluated by Emacs [15]. As the cursor is moved over source code that has been annotated, a subsidiary Emacs frame dynamically displays the annotations made to the current character. See Figure 5 for an example.

Annotating text makes more sense than annotating the abstract syntax tree for several reasons. First, cpp operates at a textual level. As we have seen, the unprocessed source code cannot necessarily be viewed as an ordinary abstract syntax tree. Even if a generalized tree could be constructed, no available interface provides adequate means of interacting with the large, complicated trees that inevitably result from realistic packages. Additionally, text permits using Emacs as the target interaction environment. This allows software engineers to augment Emacs's other powerful source code understanding tools (e.g., font-lock-mode, etags, OOBrowser, etc.) with their own annotations supplied by PCp^3 analyses.

Marking Macro Expansions

For an empirical study of C preprocessor use [4], we wanted to identify macro expansions in the unprocessed source code. Recognizing identifiers that are macro-expanded is exceedingly useful as suggested by the nearly universal convention of using all-uppercase identifiers for macro names. Previous tools have difficulty even just counting macro expansions [10]. For the empirical study, our first approximation of identifying macro expansions did not use this framework. That analysis was overly conservative: we marked as an expansion all occurrences of each identifier that was ever (in any source file of the program) `#define`d to be a macro. To gain more accurate information, we used PCp^3 framework for the subsequent version of the analyses for that study.

Since the marking of expansions should cover the entire unprocessed source code, we use the boiler-plate code mentioned previously to expose all the code to the tool. Because we wanted a conservative analysis (i.e., it should over-mark when imprecise), we treat an identifier as a macro at a program point if it has been `#define`d on any path and not yet definitely `#undef`ined. In particular, the improved analysis properly limits the effects of macros that are defined for a segment of code and subsequently undefined.

Our improved analysis using the framework is written in under 90 lines of code, most of which is the boiler-plate for handling all conditionally-included code.

Other Possibilities

Our framework has proven useful for several software engineering analyses. Other ad-hoc tools could benefit from our framework, as well. For example, Emacs's `etags` [15] creates a database of function definitions in unprocessed source but can be easily confused by package-specific macros used in function definition headers.[7] Also, searching for uses of global variables could be enhanced using our framework since it will not omit references to globals hidden by macros.

Another possible use of the framework is in identifying "tricky" uses of the preprocessor as an aid to determining, for example, which `#define`s can be replaced with language-proper features such as constant

[7] To circumvent this limitation, `etags` permits the programmer to manually specify an arbitrary regular expression that it uses to find definitions to mark.

variables or C++ inline functions.

PCp^3 Tool Details

Our framework combines a **P**arser, a **C** **p**re**p**rocessor, and a **P**erl action language. Thus, we have named the tool that implements the framework PCppP or PCp^3. The latest version of PCp^3 is available from ftp://ftp.cs.washington.edu/homes/gjb/pcp3.tgz.

Parser

Choosing a parser was difficult as there are many freely available parsers, often tightly coupled to a functional back-end, thus complicating reuse. We chose the parser from CTree [16], a freely available ANSI C front end, to embed in PCp^3 largely because of its simple implementation and fully-scoped symbol table. Its lexical analyzer and parser are mechanically generated by flex and bison [17, 11] (freely available implementations of lex and yacc, respectively). As CTree parses, it builds a complete abstract syntax tree of the preprocessed code.

The implementation of the CTree parser component of PCp^3 is about 5,000 lines of C code and bison and flex specifications. Most of the changes to the parser were to eliminate name conflicts and to call the Perl hooks as part of various reduction rules.

Preprocessor

Since PCp^3 must mimic cpp exactly, the C preprocessing library from the GNU C compiler's (gcc) well-tested (and slightly extended) cpp [18] is embedded in PCp^3.

The implementation of cpplib grew from about 7,000 lines of code as distributed with gcc to almost 8,000 lines. Most of the changes involved modifying data structures and function calls to maintain the extra macro expansion information to support the character-by-character correspondence between the source and output.

Action Language

We chose Perl as the action language for PCp^3 because it interfaces easily with C and is used widely. Additionally, Perl's built-in data structures, closures, and higher-order functions make it especially well-suited for use with the framework.

The Perl subroutine hooks are written in a user-specifiable script file that registers a subroutine for each action it wants to process. That script is free to manage its local data structures arbitrarily, and may import reusable modules as an ordinary Perl program would. However, the C data structures of the preprocessor and parser are hidden behind the hooks and backcalls interfaces. All command line options accepted by `cpp` are also accepted by PCp^3 (this often makes it easier to use the tool in place of a compiler for analyzing complete packages as described by Murphy et al. [13]). Additionally, PCp^3 accepts a `--noparse` option that turns off its parser and the calls to related hooks. PCp^3 provides over currently over forty action hooks (see the first appendix).

The implementation of the main PCp^3 program, the roughly thirty backcalls (see the second appendix), and the glue connecting the components totals about 1,800 lines of C code. About 60% of this code could be generated automatically and deals directly with passing arguments between C and Perl.

Performance

The performance of PCp^3 depends on the complexity of the analysis. For comparison, for `gcc` to compile and optimize the 5,000 lines of the `bc` package [19] (an arbitrary precision arithmetic language interpreter) required 38 seconds on a Pentium Pro 200 Linux machine. Running PCp^3 with its test analysis consisting of 600 lines of hooks that exercise every event required 4 minutes for the `bc` source. Removing hooks for `CPP_OUT` and `TOKEN`[8] reduced the running time to 50 seconds. With all action code turned off, the running time is less than 10 seconds. The useful analyses described in this paper involve only a handful of callbacks, and thus execute very quickly.

[8] These hooks are invoked on every string that `cpp` outputs and on every token it inputs, respectively. They are the most frequently invoked hooks.

Related Work

Numerous tools exist for assisting the software engineer in understanding and manipulating source code. Griswold, Atkinson and McCurdy review a number of them while motivating their `TAWK` tool [10]. `TAWK` uses C as its action language and matches patterns in the abstract syntax tree. `TAWK`, like PCp^3, operates on unprocessed source code. Instead of embedding a preprocessor, `TAWK` tries to parse each macro definition as an expression, allowing macro arguments to be types, as well. If that succeeds, the macro is left unexpanded in the code and becomes a node resembling a function call in their AST. About 92% of macro definitions in the two packages they studied parsed acceptably. For the remaining 8%, `TAWK` expands the macro uses before feeding the resulting tokens to their parser.

The Intentional Programming group at Microsoft Research [20], headed by Charles Simonyi, is interested in preserving preprocessor abstractions as they import legacy C code into their system. They developed a novel technique for handling preprocessor directives.[9] Before preprocessing, conditional compilation directives are converted to stylized variable declarations. Then, the source text is preprocessed and all macros are expanded while marking each token with its textual derivation by the preprocessor. These declarations and the other source are then run through a C++ parser to create an AST. The annotations decorate the tree, and "enzymes" privy to the meaning of the stylized declarations process the tree in an attempt to identify abstractions. When macro expansions vary from use to use (e.g., `__LINE__`), the non-constant text is considered an extra argument to the macro, and the different expansions are explicitly passed at the invocation sites. Especially unusual uses of conditional compilation directives cause problems because of constraints on where C++ declarations may go, but generally the group is optimistic about the possibilities for their approach. Some of their techniques might be applicable to a future version of PCp^3, especially as the AST-related backcalls mature.

Several systems including NewYacc [21] and ASTLog [22] parse preprocessed source code and generate an abstract-syntax tree that can then be annotated and queried during analyses. Unlike PCp^3, these systems are not useful for an analysis that deals with preprocessor directives as well as C language-level constructs.

LCLint [9] attempts to analyze macro definitions for dangerous or error-prone constructs. It allows the

[9] Charles Simonyi and Rammoorthy Sridharan, personal conversation.

programmer to add annotations to the code in comments. These annotations give the LCLint checker extra information that it then verifies. For example, a macro argument can be marked that it needs to be side-effect free at an invocation site, and LCLint will generate a warning message if that constraint is violated. Evans's focus is on finding errors, and dealing with macro expansions is largely ignored [9, Ch. 8]. Unlike PCp^3, LCLint is not designed to be a general framework for analyses

Krone and Snelting analyze conditional compilation directives in the context of a lattice-theoretic framework for inferring configuration structures from the source code. They study how `#if` guards depend on and relate to each other, and provide a means of visualizing the relationships with the goal of improving the programmer's understanding of the structure and properties of the configurations [6].

Limitations

Our framework provides a concise and flexible infrastructure for analyzing unprocessed C code. It permits reasoning about the entire source artifact and eliminates the need of individual analyses to mimic the preprocessor. Nevertheless there are several weaknesses of the framework and our implementation in PCp^3.

First, some sophisticated preprocessor analyses (e.g., the analysis in section) are dependent on the order that action hooks are called. This in turn requires an intimate understanding of the implementation's preprocessing and (to a lesser extent) parsing peculiarities. Fortunately, the fast development time provided by the scripting language permits easy exploration and testing of analyses.

Because PCp^3 works like a compiler, it handles only a single translation unit at a time, complicating whole-program analysis. The analyses discussed in section output character-indexed annotations of the input that a separate tool later applies to the source code to perform the final transformation or to permit interactive exploration. These extra tools can also combine information derived from various translation units (e.g., marking expansions in a header file that is included by multiple source files). To better support reasoning about an entire source code artifact, a database approach similar to `CIA++` [23] could be used. Atkinson and Griswold mention the importance of flexibility in allowing the user to select the appropriate balance between precision and performance of a whole-program analysis [24]. One could provide this flexibility by using one of Perl's data-persistency modules to permit specified data structures to be shared among invocations on separate translation units.

Better support for multiple conditional compilation versions would be useful. PCp^3's mechanisms for handling multiple source configurations are primitive—a single distinguished path (a version) through the source code is treated specially. Ideally, multiple versions could be considered with more uniform handling of the various paths. This would permit future blocks of code that are hidden via an `#ifdef` to be properly influenced by prior blocks of code using the same guard. Krone and Snelting suggest that the number of distinct paths through the source is reasonably bounded [6]. A generalized symbol table could track which configurations contain each symbol, and how the type of a variable depends on conditional compilation guards. A similar generalization could be made for the preprocessor name table.

Finally, although PCp^3's `CTree`-based parser constructs an abstract syntax tree, there is currently no easy way to access the AST from action callbacks. More backcalls and utility subroutines could be written to permit useful manipulation of the abstract syntax tree to avoid many of the problems created by the current limitation of a single pass over the source code. To make the AST more useful, some generalization of the tree could permit representation of preprocessor-specific annotations.

Conclusion

Our framework distinguishes itself from other software engineering tools by providing an accurate parse without disregarding the C preprocessor. By maintaining a tight mapping between the unprocessed and preprocessed code, analyses requiring expanded code can be considered in terms of the unprocessed code familiar to the programmer. This approach empowers and simplifies analyses while relieving the tool builder from the burden of partially re-implementing `cpp` for each desired analysis.

Acknowledgments

This paper was supported by a National Science Foundation Graduate Fellowship and by NSF Grant CCR-9506779. Thanks to Michael Ernst, Craig Kaplan, Brian Michalowski, Gail Murphy, and Douglas Zongker

for their thoughtful comments on earlier revisions of this paper.

Appendix A: Abridged Hooks Reference

The below lists action hooks and the conditions under which the framework invokes the corresponding callback. We omit partially redundant and less useful hooks. Passed parameters generally include source code character offsets, $s_start and $s_end, and other relevant details of the invoking action.

`STARTUP()`, `STARTPARSE()`, `EXIT($return_exit_code)`

Initialization of `cpp`, initialization of parser, and conclusion of processing.

`HANDLE_DIRECTIVE($directive_name)`

Reading of a `cpp` directive.

`DO_XIFDEF(..)`, `DO_IF(..)`, `DO_ELIF(..)`, `DO_ELSE(..)`, `DO_ENDIF(..)`

Handling of conditional compilation directives. Each of `DO_XIFDEF`, `DO_IF`, and `DO_ELIF` is invoked with ($s_start, $s_end, $conditional, $text_skipped, $value). `DO_ELSE` also reports a boolean, $fSkipping, instead of $value, and also includes $s_start_branch, the source code offset of the corresponding `#if` or `#ifdef` conditional. `DO_ENDIF` is invoked with just $s_start, $s_end, and $orig_conditional.

`CREATE_PREDEF(..)`, `CREATE_DEF(..)`

Predefined (e.g., `__GCC__`) and user-defined macro definitions (from `#define`s). These are invoked with the following arguments: $s_start, $s_end, $mname, $expansion, $num_args, $internal_expansion, $file, $line, $r_argnames, $flags, and $internal_expansion_args_uses.

`DELETE_DEF($mname,$fExists)`

`#undef` of a macro; $fExists reports whether the macro was previously `#define`d.

`SPECIAL_SYMBOL($symbol)`

Expansion of a special symbol (e.g., `__FILE__`).

`EXPAND_MACRO(..)`

In-code expansion of a macro. Arguments to this hook are: $s_start, $s_end, $mname, $expansion,

$length, $raw_call, $has_escapes, $cbuffersDeep, $cnested, @nests, $cargs, and @args. @nests and @args are Perl arrays containing macro nesting information, and the arguments of the current expansion, respectively.

`MACRO_CLEANUP($s_start, $s_end, $mname, $c_nested, @nests)`

Completed expansion of a macro (this action hook and `EXPAND_MACRO` nest).

`IFDEF_MACRO(..)`, `IFDEF_LOOKUP_MACRO($mname,$fDefined)`

Expansion of a macro inside a conditional compilation directive, or test for definedness. `IFDEF_MACRO` is invoked with the same arguments as `EXPAND_MACRO` when, e.g., an `#if FOO` is encountered. `IFDEF_LOOKUP_MACRO` is invoked instead when a macro is simply tested for definedness, e.g., by an `#ifdef FOO`.

`COMMENT($s_start, $s_end, $comment_text, $how_terminated, $numlines)`

Reading of a comment.

`STRING_CONSTANT($s_start, $s_end, $string, $lines)`

Reading of string literal.

`CPP_ERROR(..)`, `CPP_WARN(..)`, `CPP_PEDWARN(..)`

cpp errors, warnings, and pedantic warnings. Each gets the arguments $filename, $line_number, $message.

`CPP_OUT($string)`, `CPP_TOKEN(..)`

Outputting a sequence of characters written, and inputting a token. `CPP_TOKEN` reports from where the token arose. It is invoked with the arguments $token, $raw_text, $macro_name (or empty if no macro expanded to create this token), and $arg_num (or -1 if no macro expanded to create this token, or 0 if it was part of the literal body of the macro definition).

`INCLUDE_FILE($filename)`, `DONE_INCLUDE_FILE($filename)`

Inclusion of a file, and conclusion of reading an included file (these action hooks nest).

`FUNCTION($func_name,$fStatic)`, `FUNC_CALL($func_name)`

Parsing of a function definition and function invocation.

`TYPEDEF($new_name)`, `VARDECL($new_name)`

Parsing of a typedef and variable declaration parsed.

`POP_PERL_BUFFER($num_buffers_deep)`

Completion of processing a Perl-pushed (via the `PushBuffer` backcall) buffer. When $num_buffers_deep is 0, the parser is no longer parsing a macro expansion.

Appendix B: Abridged Backcalls Reference

The below backcalls are grouped by functionality. Parameters, if any, are indicated.

`CbuffersBack`, `MacroExpansionHistory`

Return the number of macro expansions deep in the current expansion, and the list of the stack of the current expansion history (the nesting of the various macro invocations).

`FExpandingMacros`

Return true if a macro is currently being expanded.

`CchOffset`, `CcchOutput`

Return a character position offset into the current input source file, and the character position offset into the output file.

`InFname`, `Fname`

Return the main filename (e.g., the one given on the command line), and the filename of the input currently being processed.

`ExpansionLookup`, `LookupSymbol`

Takes a macro or variable name; return the expansion of that macro or the symbol table entry of the identifier.

`EnterScope`, `ExitScope`

Push and pop new symbol table scopes.

`PushHashTab`, `PopHashTab`

Push and pop new macro definition hash table scopes.

`ParseStateStack`, `SetStateStack`

Return and set the stack of states for the parser (the latter takes a list of parser states).

`YYGetState`, `YYSetState`

Return and set the current state for the parser (the latter takes a parser state).

References

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[2] Samuel P. Harbison. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, 3rd edition, 1991.

[3] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[4] Michael Ernst, Greg Badros, and David Notkin. An empirical study of C preprocessor use. Technical Report UW-CSE-97-04-06, University of Washington, April 1997.

[5] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Sigsoft 96*. Sigsoft, 1996.

[6] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society Press, May 1994.

[7] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):263–291, July 1996.

[8] Steve Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Hingham, MA, 1995.

[9] David Evans. *LCLint User's Guide*. MIT Laboratory for Computer Science, Cambridge, MA, v2.2 edition, August 1996.

[10] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE 1996 Workshop on Program Comprehension*, March 1996.

[11] John R. Levine. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 1992.

[12] John Ousterhout. Scripting: Higher level programming for the 21st century. Web document, March 1997. http://www.sunlabs.com/people/john.ousterhout.

[13] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[14] Michael Ernst, Greg Badros, and David Notkin. An empirical study of C preprocessor use. To replace technical report UW-CSE-97-04-06 upon completion., October 1997.

[15] Bil Lewis, Dan LaLiberte, Richard Stallman, and The GNU Manual Group. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, Cambridge, Massachusetts, 2nd edition, May 1994.

[16] Shaun Flisakowski. CTree distribution, v0.5. Freely available software package, July 1997. ftp://ftp.kagi.com/flisakow/ctree_05.tar.gz.

[17] Free Software Foundation. Bison v1.25 and flex v2.5 distributions. Freely available software package, April 1992,1995. ftp://prep.ai.mit.edu/pub/.

[18] Free Software Foundation. GCC distribution, v2.7.2.2. Freely available software package, January 1997. ftp://prep.ai.mit.edu/pub/gcc-2.7.2.tar.gz.

[19] Free Software Foundation. bc distribution. Freely available software package, June 1998. ftp://prep.ai.mit.edu/pub/bc-1.05a.tar.gz.

[20] Charles Simonyi. The intentional programming overview. Technical report, Microsoft Corporation, 1996. http://www.research.microsoft.com/research/ip/main.htm.

[21] James J. Purtilo and John R. Callahan. Parse-tree annotations. *Communications of the ACM*, 32(12):1467–1477, December 1989.

[22] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[23] Judith E. Grass and Yih-Farn Chen. The C++ information abstractor. In *Proceedings of the USENIX 1992 C++ Conference*, Portland, Oregon, August 1992.

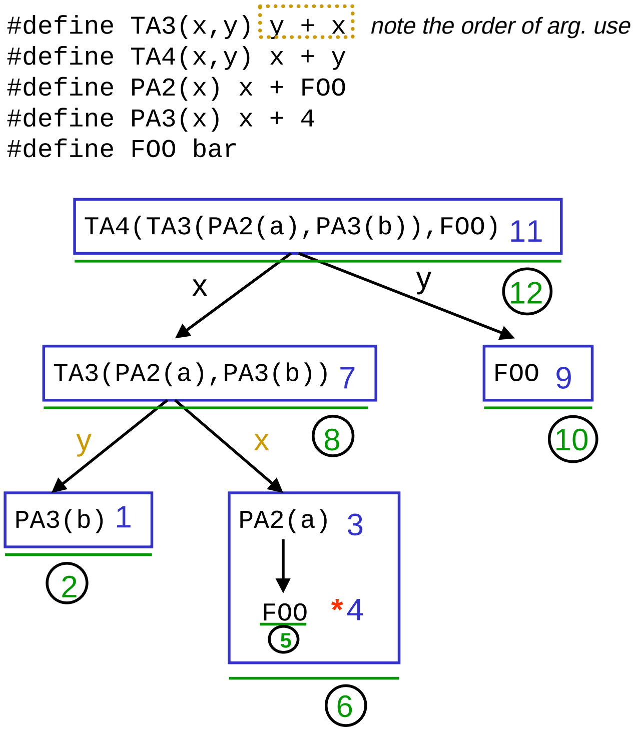
[24] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, March 1996.

```
use pcp3;
my %func_calls = ();  # the dictionary of callees for the current function
                      # maps called function name to number of times in appears
                      #   in the definition of the calling function

AddHook("FUNC_SPEC", sub { %func_calls = (); } );
AddHook("FUNC_CALL", sub { # $_[0] argument is the name of function invoked
                           $func_calls{$_[0]}++; } );
AddHook("FUNCTION",
   sub { # $_[0] argument is the name of function just defined
     if (scalar (keys %func_calls) > 0) {
       print "$_[0] calls ", join(", ", sort keys %func_calls), "\n";
     }
   } );
```

Figure 1: A complete static analysis to extract a call graph.

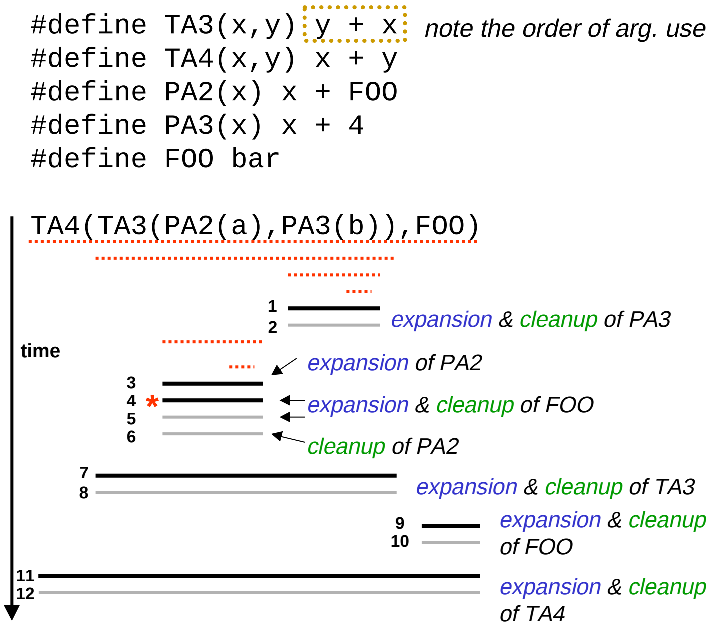
```
use pcp3;
my %func_calls = ();  # the set of callees for the current function
my $expanding = 0;    # are we expanding a macro?

AddHook("FUNC_SPEC", sub { %func_calls = (); });
AddHook("FUNC_CALL", sub { # $_[0] argument is the name of function invoked
                           $func_calls{$_[0]}++ if !$expanding; } );
AddHook("EXPAND_MACRO", sub { # $_[2] argument is macro name being expanded
                              $func_calls{$_[2]}++ if !$expanding;
                              ++$expanding} );
AddHook("MACRO_CLEANUP", sub { --$expanding; } );

AddHook("FUNCTION",
  sub { # $_[0] argument is the name of function just defined
    if (scalar (keys %func_calls) > 0) {
      print "$_[0] calls ", join(", ", sort keys %func_calls), "\n";
    }
  } );
```

Figure 2: A revised call graph extractor that treats macros as functions.



```
#define TA3(x,y) y + x
#define TA4(x,y) x + y
#define PA2(x) x + FOO
#define PA3(x) x + 4
#define FOO bar
```

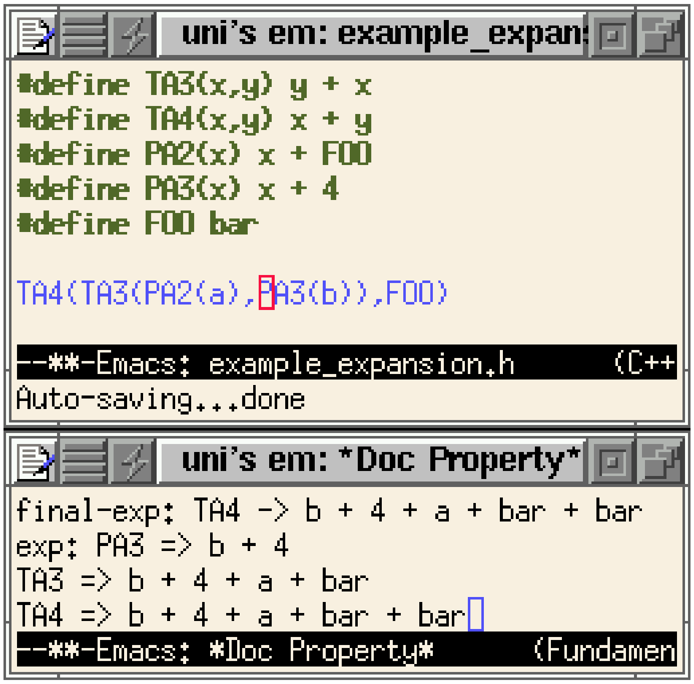
Figure 3: An example macro expansion and the related hooks that are called. Numbers indicate the order in which the hooks are called (corresponding to post-order traversal of the tree). Uncircled numbers are `EXPAND_MACRO` actions, circled numbers are `MACRO_CLEANUP` actions; e.g., the fifth hook PCp^3 invokes is `MACRO_CLEANUP` for "`FOO`". The "cleanup" means that the macro has been fully expanded and is ready to be substituted into the output text (or parsed by PCp^3). In general, the tree need not be binary. Note that the leaves of a node are the arguments in the order of appearance in that node's macro expansion (not their order in the list of formal parameters).



```
#define TA3(x,y) y + x
#define TA4(x,y) x + y
#define PA2(x) x + FOO
#define PA3(x) x + 4
#define FOO bar

TA4(TA3(PA2(a),PA3(b)),FOO)
```

note the order of arg. use
time
1
2
expansion & cleanup of PA3
expansion of PA2
3
4 *
5
6
expansion & cleanup of FOO
cleanup of PA2
7
8
expansion & cleanup of TA3
9
10
expansion & cleanup of FOO
11
12
expansion & cleanup of TA4

Figure 4: Another view of an example macro expansion and the hooks that are called. Dotted lines are the nestings of macro expansions as the algorithm recurses. Dark solid lines are EXPAND_MACRO actions on the underlined macro invocation, light solid lines are MACRO_CLEANUPs.



```
#define TA3(x,y) y + x
#define TA4(x,y) x + y
#define PA2(x) x + FOO
#define PA3(x) x + 4
#define FOO bar

TA4(TA3(PA2(a),PA3(b)),FOO)

--**-Emacs: example_expansion.h        (C++
Auto-saving...done
```

```
final-exp: TA4 -> b + 4 + a + bar + bar
exp: PA3 => b + 4
TA3 => b + 4 + a + bar
TA4 => b + 4 + a + bar + bar
--**-Emacs: *Doc Property*        (Fundamen
```

Figure 5: A view of Emacs using the authors' "doc-property-display" feature to dynamically view textual annotations of results of the analyses of PCp^3. The outline box in the top frame indicates the user's cursor position, and the lower frame lists the various text properties attached to the character there. The bottom frame potentially changes after each cursor movement.