PCp^3: A C Front End for Preprocessor Analysis and Transformation

Greg J. Badros*
gjb@cs.washington.edu

16 October 1997

Abstract

Though the C preprocessor provides necessary language features, it does so in an unstructured way. The lexical nature of cpp creates numerous problems for software engineers and their tools, all stemming from the chasm between the engineer's view of the source code and the compiler's view. The simplest way to reduce this problem is to minimize use of the preprocessor. In light of the data collected in a prior empirical analysis, this paper describes a tool to aid the software engineer in analyses targeted at replacing preprocessor constructs with language features. Existing tools for analyzing C source in the context of the preprocessor are unsuitable for such transformations. This work introduces a new approach: tightly integrating the preprocessor with a C language parser, permitting the code to be analyzed at both the preprocessor and syntactic levels simultaneously. The front-end framework, called PCp^3, combines a preprocessor, a parser, and arbitrary Perl subroutine "hooks" invoked upon various preprocessor and parser events. PCp^3's strengths and weaknesses are discussed in the context of several program understanding and transformation tools, including a conservative analysis to support replacing cpp's `#define` directives with C++ language features.

1 Introduction

More than twenty years ago, Dennis Ritchie designed the C language [KR88] to include a textual macro preprocessor called cpp [Har91, 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 may have been justified. For the last two decades, C programs have exploited cpp's capabilities for everything from manifest constants and type-less pseudo-inline functions, through 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" [Str94, 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 [Str94, p. 424].

*Supported by a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author, and do not necessarily reflect the views of the National Science Foundation.

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

The intrinsic problem with cpp is also its fundamental strength: it is a conceptually distinct first pass of textual 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 separate from the preprocessor) ultimately compiles.[1] Consider the program in Figure 1. Though a legal ANSI C program, its semantics are undefined in light of cpp. When compiled using `cc -Dprintf(x)=`, the program no longer outputs the familiar "Hello world" message. (Adapted from an example given by Stroustrup [Str94, p. 423].) What *you* see is not what the *compiler* gets.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello world");
  return 0;
}
```

Figure 1: Example of incomplete semantics of C source.

Experienced and novice C programmers alike have been frustrated by similar misunderstandings of source code due to the arbitrary transformations cpp performs before the language compiler ever is invoked.[2] Such confusions are easily eliminated by allowing the software engineer to see the code exactly as the compiler does. Unfortunately, that view of the program is a level of abstraction lower than the unprocessed source provides. Well-known identifiers such as `stderr` appear as the far less readable `(_IO_FILE*)(&_IO_stderr_)`, and useful encapsulations such as `assert` degenerate into sequences of symbols which are nonsense to a human programmer.

1.2 What this means for software tools

Because of the preprocessor's textual foundations, C source code cannot be parsed directly. Only after preprocessing is a C program in a grammatically usable form. Because of the relative ease in parsing preprocessed code, the majority of software engineering tools operate on exactly that view of the source, losing artifact information that was removed prematurely by the preprocessor. 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. This approach results in information that is precise for the input considered, but its usefulness for human program understanding is diminished because of changes in the source artifact from preprocessing.[3] Additionally, parsing requires a syntactically correct program and all header files to exist—these constraints are not realistic during many software maintenance

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

[2] Some more modern C compilers integrate the preprocessor with the lexical scanning phase, but, by necessity, it is still a separate first phase conceptually.

[3] In contrast, this approach is exactly right for the compiler which has little need of preserving high level abstractions in generating object code. However, in the presence of incorrect source code, the compiler itself is a program-understanding tool and preprocessing hinders its ability to accurately describe the problem. The `#line` directive is used to keep some rudimentary (but far too coarse) mapping to the original source to permit better errors and warnings.

activities such as porting or switching compilers. Another significant disadvantage of preprocessing is that it eliminates conditional compilation artifacts that are essential to the portability and versatility of the source code (see [KS94]). 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 choose a different tradeoff and operate instead on the unprocessed source code exactly as the programmer sees it. These tools cannot use a straightforward parser or construct an accurate abstract syntax tree because the source is not grammatically-correct C code. Lexical tools such as `etags`, LSME [MN96], and `font-lock-mode` for Emacs and approximate parsers such as `Genoa` [Dev92], and `LCLint` [Eva96] use this approach. In general, this leads to greater speed (especially for the lexical approach) and improved robustness in handling syntax errors and language variants (approximate parsers generally lose the current top level construct at worst, and often do better). Additionally, because the input is unprocessed, the extracted information is presented to the human software engineer at the same level of abstraction as the source with which she is working. Unfortunately, disregarding or only partially honoring the cpp directives leads to an extracted model of the source code which is only an approximation of the program's appearance to a compiler. The preprocessor can still wreak havoc by hiding arbitrary code in macro expansions or customizing the syntax of declarations or scoping constructs. Griswold and Atkinson studied mistakes in call graph extraction using various tools and found that macro expansion was a major cause of both false positives and false negatives [GAM96]. Such approximate tools are inappropriate for software engineering analyses that require exact or conservative information.

An obvious solution to the problems cpp presents is to just avoid it, and therefore C, entirely. By using a different language which directly provides features such as modules, constants, inline functions, generic functions, and other higher level abstractions that C emulates via its preprocessor, the above difficulties are avoided. However, casting away C in favor of almost any modern language would necessitate discarding many millions of lines of useful legacy code. The one notable exception is C++ [Com96]. For compatibility with C, C++ remains encumbered by cpp, but it also provides language-level support for many higher level constructs, which makes numerous cpp constructs redundant. Migrating C code to C++ potentially provides a path to reduce usage of the preprocessor to the benefit of tools and programmers. A fundamental motivation in the design of PCp^3 is to automatically assist in such transformations.

1.3 Outline

Section 2 highlights significant findings from an empirical study of C preprocessor use [EBN97a] with an emphasis on determining what fraction of its use in existing software artifacts can be replaced by C++ language features. Section 3 describes the tool developed to support accurate analysis of unprocessed code without losing the high level abstractions expressed with the preprocessor. Section 4 discusses the use of the tool to assist in automating safe conversion of some simple uses of cpp into modern language constructs. Section 5 describes related work, and section 6 discusses the contribution of this work, its shortcomings, and some directions for possible future progress. The two appendices provide more details about the PCp^3 interfaces.

2 How the preprocessor is used in practice

Many cpp constructs have analogues in newer language features. Carroll and Ellis list several preprocessor uses and explain what C++ features could be used instead [CE95]. For example,

#defines of simple numeric and string constants can often be replaced with enumeration declarations or static constant variable declarations (both newer features of ANSI C, not just of C++). Some function-like #defines of pseudo-inline functions can correspond to C++'s real inline functions (perhaps made generic through use of a template). An optimizing compiler performing dead code elimination can effect the same result as a conditional compilation directive guarding debug-only code. (The compiler-based mechanism does not assist in conditionally-included declarations, though.)

Over the last year, several colleagues and the author have investigated how the C preprocessor is used in a sample of 30 freely available software packages [EBN97a]. Our findings show that the preprocessor is used very heavily—almost 10% of the lines[4] in the packages analyzed contain preprocessor directives. About a third of these are macro definitions, a bit more than a third are conditional compilation directives, and the remainder are mostly #includes. Source code lines also are affected by the preprocessor if they are embedded inside any conditional compilation construct, or if they contain macros which will be expanded. About 28% of lines expand one or more macros, and over 40% of lines have some dependence on the preprocessor. These numbers confirm that the C preprocessor is used extensively.

Though cpp is used a great deal, not all uses of the preprocessor are equally disorienting to tools and programmers. For example, a fairly common idiom uses a __P macro to remove function prototypes for non-ANSI C compilers (see Figure 2). Although 50% of macro names are used fewer than three times (over 10% are never used), this macro is used frequently and can cause a parsing problem or a lexical mismatch at each function prototype. The declaration no longer appears to be a function prototype to tools operating on unprocessed source code. Interestingly, the use of the __P macro hardly affects readability of code at all; humans compensate easily for this alternate form of declarations.

```
#if defined (__STDC__)
#  if !defined (__P)
#    define __P(protos) protos
#  endif
#else /* !__STDC__ */
#  if !defined (__P)
#    define __P(protos) ()
#  endif
#endif

/* Below has no prototype for non-ANSI compilers */
int strcmp __P((const char *sz1, const char *sz2));
```

Figure 2: Common use of macro for backward compatibility with non-ANSI C compilers. Note the double parenthesization of the argument to macro __P; this permits in the entire formal parameter list to be used as the single argument to the macro. (Adapted from string.h header.)

Because of the generality of the macro expansion mechanism, the problems can be much worse. A macro can expand to mismatched parentheses or braces, or arbitrary symbols such as a semicolon. They can even manipulate their arguments as symbols using the cpp stringization

[4] Counts of lines always exclude lines which are blank or contain only comments, unless otherwise noted.

```
#define START_PARAMS (
#define END_PARAMS )
#define START_CALL (
#define END_CALL )
#define BEGIN {
#define END }
#define ARRAY []
#define NL ;
#define POINTER *
#define COMMA ,
#define STRING(x) #x

#include <stdio.h>

int main START_PARAMS int argc COMMA char POINTER argv ARRAY END_PARAMS BEGIN
  printf START_CALL STRING(Hello world\n) END_CALL NL
  return 0 NL
END
```

Figure 3: Legal ANSI C program which is difficult for tools and humans to understand without preprocessing. Though this example is representative of actual abuses of the preprocessor, people generally are more merciful in their use of cpp.

(#) and pasting (##) operators. Figure 3 shows how damaging overuse of such features can be to human program-understanding. Also, approximate parsers operating on the unprocessed code would have to expand many macros in order to make sense of the code, and lexical tools would be virtually useless.

Our study of how programmers actually use cpp observed that macros are often used in benign ways. Over 43% of macro names simply expand to constants, and another 31% expand to expressions. These types of macros generally provide few problems to software tools using unprocessed source since unexpanded macros simply appear to such tools as identifiers which are often transparently replaceable by more general expressions. Another 7% are null defines—macros whose expansions are empty, usually used either in conditional compilation directive guards (e.g., NDEBUG), or as a means of removing a keyword that might be unrecognizable to an older compiler (e.g., __const). The remaining 20% of macro names expand to statements (4%), arbitrary symbols (10%), involve types (1.4%), or could not be fit into any of the other categories. Only 0.3% of macro names affect syntax in unusual ways (e.g., containing only punctuation or mismatched parentheses or braces).

However, there is some bad news: the syntactic and type-related macros are used significantly more frequently than macro names with more simple expansions. While 90% of all macro names are expanded twenty or fewer times, only 55% of the syntactic macros are used that infrequently. In fact, 15% of them have 160 or more uses, each of which could cause problems for a software tool operating on unprocessed source and not expanding macros.

This data suggests that ignoring the preprocessor's macro expansions can result in significant missing syntactic features, reducing a tool's ability to glean information from the unprocessed source. Conversely, using a separate initial preprocessing pass discards too much information and destroys useful abstractions. PCp^3 tries to find a balance between these two extremes.

3 The PCp^3 Infrastructure

PCp^3 is built from three software components: a **P**arser, a **C** **p**re**p**rocessor, and a **P**erl action language.[5] These subsystems interact through well-defined interfaces, and together result in a powerful, expressive, and flexible framework for accurately analyzing C code in the context of its cpp directives.

3.1 Parser

The first major component of PCp^3 is an ANSI C compatible parser. Choosing a parser was difficult as there are many freely available parsers, often tightly coupled to a functional back-end, thus complicating reuse. The parser from `CTree` [Fli97], a freely available C front end was chosen to embed in PCp^3. Its lexical analyzer and parser both are mechanically generated by `flex` [Fou95, Lev92] and `bison` [Fou92, Lev92] (freely available implementations of `lex` and `yacc`, respectively). As `CTree` parses, it builds a complete abstract syntax tree of the preprocessed code. It also implements a fully scoped symbol table, assuming another essential duty for PCp^3.

The implementation of the `CTree` parser component of PCp^3 remains at about 5,000 lines of C code and `bison` and `flex` specifications. Most of the changes to the parser were to eliminate name conflicts[6] and introduce calls to the various action routines.

3.2 Preprocessor

Since it is essential that the framework mimic cpp exactly, the C preprocessing library from the GNU C compiler's (gcc) well-tested (and slightly extended) cpp [Fou97] is embedded in PCp^3. By using gcc's `cpplib`, PCp^3 is able to consider both the unprocessed and the preprocessed views of source code. Though cpp itself inserts `#line` directives in its output to provide a rudimentary mapping between the two views, PCp^3 keeps a much finer mapping, tracking what tokens came from source code directly, from expansions of macro bodies, arguments of macros, etc.

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 tighter correspondence between source and output.

3.3 Perl action language

Though `bison` parsers associate actions with parse rule invocation, PCp^3 requires a more general mechanism to support user-definable actions on preprocessor events as well. Griswold and Atkinson note that various software tools benefited from using a special-purpose interpreted action language [GAM96]. In general, interpreted languages can speed development time [Ous97], especially when prototyping analyses. For PCp^3, Perl [Wal97, WCS96] was chosen because it interfaces easily with C and is used widely. Though actually a compiled language, Perl's compilation phase is exceedingly fast and integrated with the execution; thus it has the same advantages in reducing developer time as interpreted languages, yet its performance is substantially better. Additionally, most of the tools for the empirical study of C preprocessor use [EBN97a] are written in Perl, enabling reuse.

[5] Hence its name, PCppP, shortened to PCp^3.

[6] Ironically, the preprocessor came to the rescue here; macros provided an easy way to rename `CTree`'s symbols from its automatically-generated parsers and lexical analyzer which conflicted with those used when embedding Perl.

The Perl interface of PCp^3 is composed of two parts:

- Action "hooks" written in Perl that PCp^3's preprocessor and parser engines invoke on various events. Each hook is directly passed a set of parameters relevant to the current action. Example actions include 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. There are forty hooks presently specified in PCp^3. See Appendix A, on page 18 for details.

- Subroutine "backcalls" written in PerlXS (the dialect of C used for Perl extensions) that the Perl hooks are free to call to access pre-specified C data structures or actively interact with the Parser or cpp components of PCp^3. Example subroutines 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. There are twenty-six backcalls presently permitted by PCp^3. See Appendix B, on page 22 for details.

The Perl subroutine hooks are written in a user-specifiable script file which registers a subroutine for each action it wants to process. That script is free to manage its local data structures arbitrarily, and may import 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. Additionally, PCp^3 accepts a `--noparse` option which turns off its parser and the calls to related hooks (parsing-related backcalls are permitted but generally do nothing). As a convenience, PCp^3 also provides a Perl module containing various useful (but optional) utility functions (e.g., an `AddHook` subroutine which manipulates the global hash dictionary that PCp^3 consults to invoke the appropriate subroutines upon each event). A complete PCp^3 analysis is shown in Figure 4.

```
use pcp3;
AddHook("TYPEDEF",\&Typedef);
AddHook("EXIT",\&Exit);

sub Typedef { my ($name) = @_;
  push @typedefs, $name;
  $num_typedefs{cpp::Fname()}++;
}

sub Exit { my ($return_value) = @_;
  print "Typedefs:\n", join("\n",sort @typedefs), "\n\n";
  print "Number per file:\n";
  foreach my $fname (sort keys %num_typedefs) {
    print $fname, " had ", $num_typedefs{$fname}, " typedefs\n";
  }
}
```

Figure 4: A complete PCp^3 analysis to list all the types defined in a translation unit, and counts of the number of definitions made in each of the files processed. Perl's list and hash data structures make the analysis especially easy to write.

The implementation of the main PCp^3 program, the backcalls, and the glue connecting the components totals about 1,800 lines of C code. About 60% of this code deals directly with passing arguments between C and Perl.

3.4 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.[7] 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:[8]

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 are used 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.[9] 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.

Consider the example illustrated in figures 5 and 6. 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

[7] 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.

[8] This description is necessarily implementation specific. See [Har91, Ch. 3] for details of what is required by the C language standard. Also note that details of stringization and pasting are omitted as they are infrequently used features [EBN97a].

[9] 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.

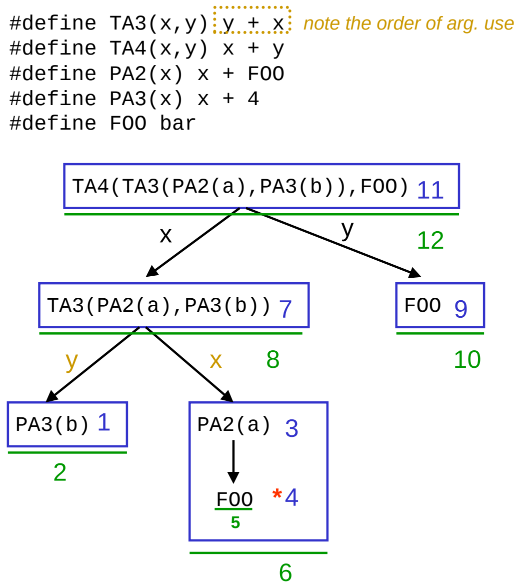

Figure 5: An example macro expansion and the related hooks that are called. Numbers indicate the order in which the hooks are called (corresponding to post-order traversal of the tree). Blue numbers are `EXPAND_MACRO` actions, green numbers are `MACRO_CLEANUP` actions; e.g., the fifth hook PCp[3] invokes 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).

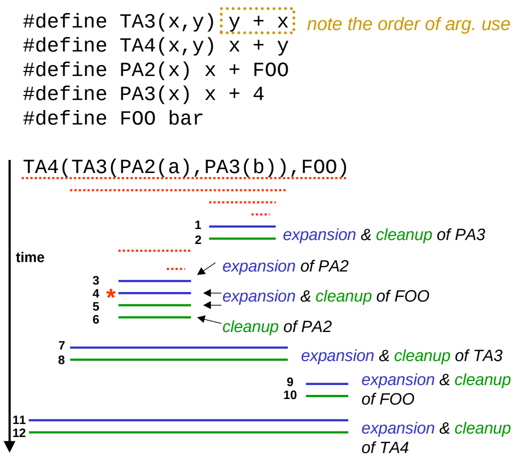

Figure 6: Another view of an example macro expansion and the hooks that are called. Red dotted lines are the nestings of macro expansions as the algorithm recurses. Blue lines are `EXPAND_MACRO` actions on the underlined macro invocation, green lines are `MACRO_CLEANUP`s.

of macro `PA2`. From Figure 6, the `@nests` list for a given expansion corresponds to the sequence of red dotted lines directly above and completely overlapping the blue line representing that expansion. Similarly, the list `MacroExpansionHistory()` returns can be visualized as the stack of blue expansion lines directly above the expansion in question (those that have not already been paired with a green line representing the completion of their expansion).

3.5 Program understanding support in Emacs

Some of the analyses PCp^3 has been used to perform generate large amounts of information that are not easy to comprehend in raw form. For example, the mapping between unprocessed source code and preprocessed code aids program understanding little when the correspondence exists only in a Perl data structure. The standard module of hook utilities provides a mechanism to output character-indexed annotations of the source code. These annotations are Emacs Lisp source code which manipulate text properties of character ranges when evaluated by Emacs [LLST94]. This provides a useful mechanism (indeed, essential during the debugging of the framework) for visualizing the results of the analyses within the context of the code analyzed. 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 7 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

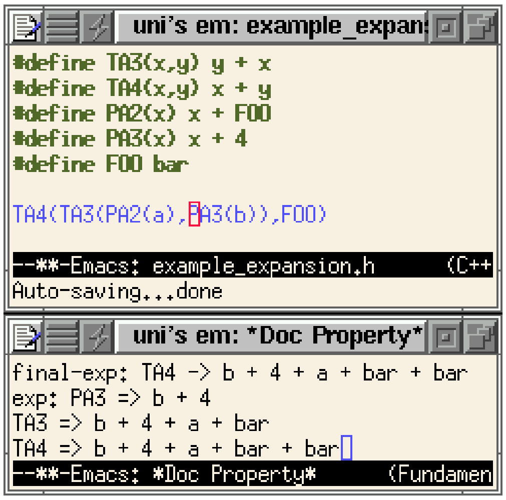

Figure 7: A view of Emacs using the author's "doc-property-display" feature to dynamically view textual annotations of results of the analyses of PCp^3. The red outline box in the top frame indicates the user's cursor position, and the lower frame lists the various text properties attached to the character there. The bottom frame potentially changes after each cursor movement.

their own annotations supplied by PCp^3 analyses.

3.6 Interactions among components

Given a straightforward preprocessor and parser front end, some analyses are impossible. For example, since the preprocessor will skip code if instructed by conditional compilation directives, that source will never be seen by the parser, leaving unprocessed code. The `DO_XIFDEF` and `DO_IF` hooks do receive the code that is skipped as an argument, but only as literal text. For an analysis to reason about the whole source program (not just one configuration), the skipped code must be exposed to the parser as if it were not removed by the preprocessor.

The general mechanism for inserting arbitrary source text to be processed is the `PushBuffer()` backcall. Since such program text can include arbitrary preprocessor directives and C code, we might 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. The data structures that must be preserved to avoid side-effects to the main processing (i.e., the version selected by the `#define`dness of macro names) are the preprocessor's hash table of macros and the parser's current stack of states and symbol table. 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 parser, the backcalls `EnterScope()` and `ExitScope()` provide similar functionality. Additionally, `YYPushStackState()` and `YYPopStackState()` save and restore the stack of parser states.

Numerous other backcalls provide additional support for querying and interacting with the parser. `ParseStateStack()` returns the list of states in the parsers 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 reset to a top level construct or attempt to reparse 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.

4 Converting preprocessor uses into language features

The PCp^3 framework makes performing many kinds of analysis particularly easy. Certain analyses are especially useful when considering source code transformations. To safely replace a `cpp` construct with a language feature, we must be sure that doing so does not break the existing code; semantics and performance should be preserved or improved. Compilers necessarily sit at one extreme of the transformation spectrum—they must absolutely preserve semantics of their source language as they lower the level of abstraction to machine instructions. Other tools perform source-to-source transformations mostly as an editing aid, largely disregarding semantics [Laf97, Til97].

Software engineering source-to-source transformations can afford to be slightly less stringent than a compiler, but should strive to be more than just an editing tool. PCp^3 provides an analysis framework powerful enough to be as accurate as a compiler, while requiring minimal programming effort to implement an analysis. Nothing inherent to the framework disallows completely automatic conversions of entire packages, but PCp^3 better supports interactive editing-based approach. This seems preferable especially while the analyses and transformations are still being refined.

A fairly simple transformation that is well within PCp^3's capabilities is replacing `#define`s of constant symbols with language-proper constant variables. Though the transformation sounds straightforward, there are several subtleties. Consider the example in Figure 8. Each of FOO, FUD, BAR, BAZ, BING, and BONG expand to integers, and so are candidates for replacement with C const ints.

```
#include <stdio.h>
#define CCD2
#ifdef CCD1
#  define FOO 1
#endif
#ifdef CCD2
#  define FOO 2
#endif

#define FUD 3
#define BAR 4
#define BAZ 5
#define BING 6

int
main(char **argv, int argc) {
  int fud = FUD;
  int BONG = -1;
  int rgi[BING];
  printf("%d, %d, %d, %d, %d, %d\n",FOO,FUD,BAR,BAZ,BING,BONG);
#define BAR -2
#define BONG 7
  printf("%d, %d, %d, %d, %d, %d\n",FOO,FUD,BAR,BAZ,BING,BONG);
  return 0;
}

#undef FUD
```

Figure 8: Example illustrating some subtleties when considering transforming #defines into constant declarations. See Figure 9 for the transformed version.

However, upon closer examination, there are several reasons why we conservatively should not simply replace some of the definitions with corresponding static variables:

- FOO has two definitions that are not necessarily mutually exclusive (they would have to be to ensure that the program would not produce cpp warnings).
- BAR is redefined (this definitely produces a cpp warning). Additionally, the location of the second definition is inappropriate for a C declaration (this is irrelevant if we are converting to C++).
- BING is used as an array bounds specifier in the declaration of rgi. Array bounds specifiers are required to be "constant expressions", which ANSI C does not consider a const int to be. C++ and gcc both do allow this form.[10]

[10]Though only recently did the C++ draft standard evolve to handle this form properly inside of a class de-

- `BONG` is used as a variable identifier before later being defined. Because the `#define` of `BONG`, like that of `BAR`, is at an inappropriate location for a C declaration, it needs to be moved earlier in the code. However, this would result in a name conflict with the integer variable `BONG`.

Using PCp^3 and a simple analysis, all of the above situations are recognized. The source code is annotated so that when the user's cursor is resting on a definition, Emacs presents the warnings and other messages related to that definition. Also tracked are the definition and use chains of the macro names. Thus, when the software engineer would like to perform the suggested transformation, a single Emacs command makes all the necessary changes. PCp^3 will suggest using a `static const int` variable if there are no uses in array-bounds specifiers, otherwise it will suggest using an anonymous `enum` declaration (which is an ANSI C constant expression). (Note that the former also raises some issues about declarations vs. definitions that we will ignore here.)

Converting a `#define` into a variable or `enum` declaration using the same identifier is not ideal. Naming convention dictates that only macros be all uppercase, so the identifier should be changed when it names a language-proper variable. If there are no name conflicts, the analysis will recommend the old name translated into all lowercase. For example, `BING` will be renamed `bing`, and all the uses will be marked for update. Note, though, that replacing `FUD` with `fud` would result in a name conflict not unlike the problem with `BONG`, above. In such instances, the analysis records the potential name conflict and does not alter the variable name. See the code resulting from the analysis's transformations in Figure 9

Because all of the heuristics for the transformation are implemented in Perl action code, they are easy to develop, test, and refine. Since the PCp^3 framework does not specifically provide mechanisms for altering source code, analyses supporting transformations can instead leverage the power of Emacs and its editing capabilities. When provided accurate annotations and appropriate instructions for safe transformations, the actual editing of the code can be done interactively with simple Emacs support overseeing the changes.

5 Related Work

Numerous tools exist for assisting the software engineer in understanding and manipulating source code. Griswold and Atkinson review a number of them while motivating their `TAWK` tool [GAM96]. `TAWK` uses C as its action language and matches patterns in the abstract syntax tree. They take significant pains to handle macros correctly, since they use a parser which could otherwise be too fragile. They try 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%, they expanded uses before feeding the resulting tokens to their parser. Griswold and Atkinson note that counting macro expansions is difficult, yet this is something a PCp^3 analysis can do in under five lines of code.

The Intentional Programming group at Microsoft Research, headed by Charles Simonyi, is interested in preserving preprocessor abstractions as they import legacy C code into their system. Rammoorthy Sridharan, a researcher in the group, developed a novel technique for handling preprocessor directives. Before preprocessing, conditional compilation directives are converted to

claration [Com96]. Because of the "misfeature" of such declarations still requiring definitions outside of the class, Stroustrup still recommends using `enums` for static integral constant members (i.e., the "enum hack") [Str97, p. 249].

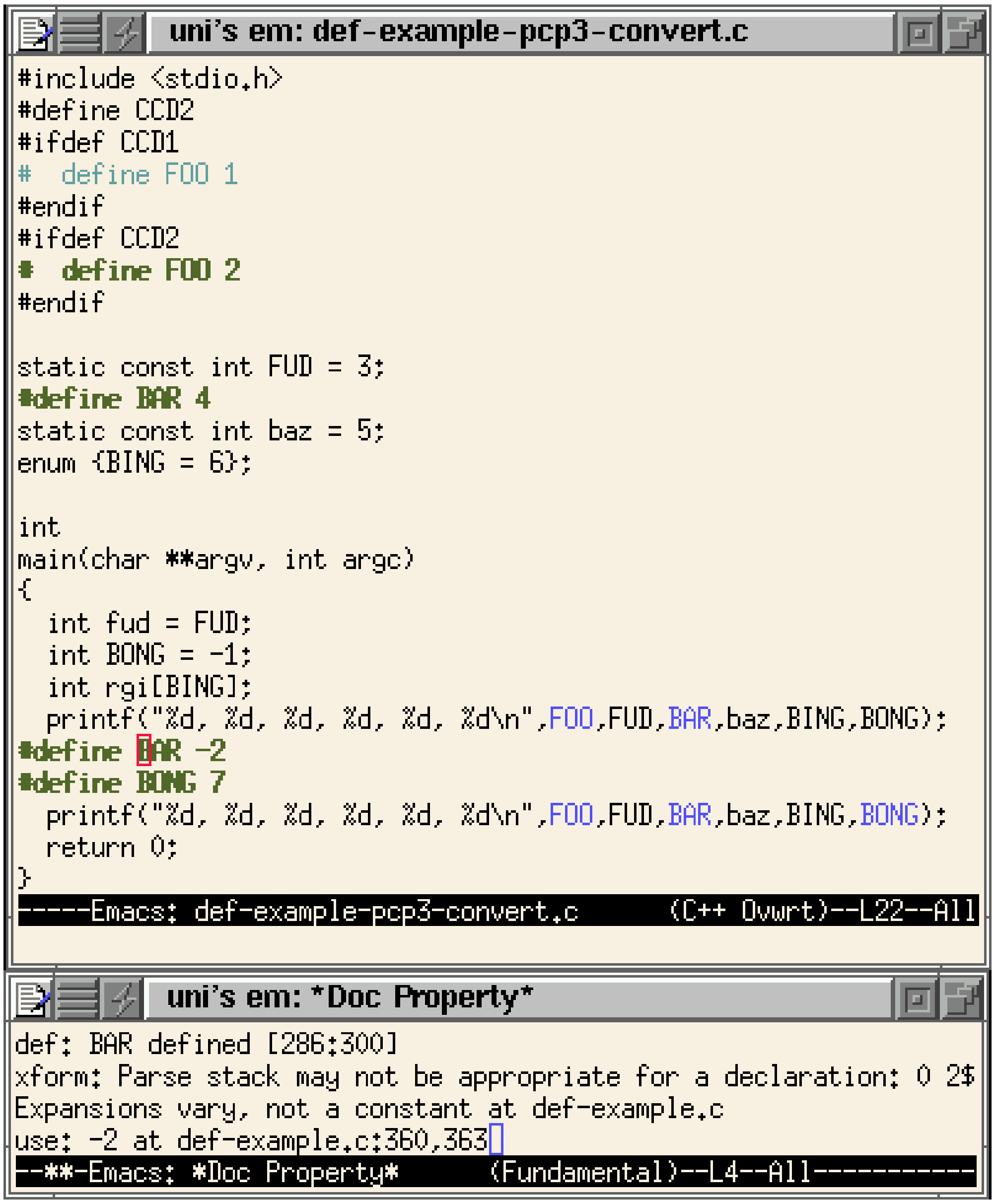

Figure 9: The code from Figure 8 after being transformed by the PCp^3 analysis. Note that `FOO`, `BAR`, and `BONG` are conservatively not converted because they are used specially. Text coloring is done by the analysis: blue text indicates macros that expand. Dark green text marks macro definitions, while light green text notates code that was skipped in the main parse (here because `CCD1` was not defined). The doc-property frame displays the annotations, including the warnings about complications, for the `BAR` macro.

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 [Sim97].

LCLint attempts to analyze macro definitions for dangerous or error-prone constructs. It allows the programmer to add annotations to the code in comments. These annotations give the LCLint checker extra information that it then checks (similar to types in many languages). 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 the analysis of macro expansions is fairly uninvolved [Eva96, Ch. 8].

Cordy and Carmichael designed `TXL` as a language to support arbitrary transformations on abstract syntax trees [CC93]. Their work is not targeted specifically to C code, and ignores issues relating to the preprocessor. `TXL` uses a grammar to parse the language and creates the tree, then applies some transformations directly on the tree, and uses an "unparser" to write the new source code back. Their transformations are similar to the syntax macros suggested for use with C by Weise and Crew [WC93]. These constructs resemble the macros provided by Lisp [Ste90], and have the desirable feature of syntactic safety, guaranteeing that macros can not introduce syntax errors. Crew also created `ASTLOG`, a Prolog-inspired language for examining abstract syntax trees. Again, he largely ignores the preprocessor, instead assuming a parse tree as the initial representation.

Stroustrup explains that making the preprocessor redundant was a design goal of C++ [Str94, p. 424].[11] Though C++ provides no feature to replace the obviously still-needed `#include` directive, Stroustrup discuss the form that the construct might take. Carroll and Ellis make the strong claim that "C++ provides mechanisms that obviate the need for most macros" [CE95, p. 147]. They list numerous uses of the preprocessor and describe the C++ language features that could replace them. Their advice is mostly prescriptive and is directed to the new C++ programmer who might be inclined to overuse cpp constructs as a crutch instead of exploiting C++ language features. They do not directly address converting existing use of cpp in legacy code. Reiss and Davis casually mention a research project that helps the user restructure programs by "translat[ing] an existing C program into C++" [RD95, p. 2], but nothing further is mentioned about the project.[12]

Siff and Reps also consider a transformation of C code to C++. They use type inference to generalize C functions and create a reusable template-based C++ function. Their technique applies only to C functions after they have been preprocessed.

Davis provides a thorough review of the issues involved in translating C++ language constructs to Java. He ignores the preprocessor and its contribution to C++ source code artifacts [Dav97]. Laffra's `C2J` and Tilevich's derived `C2J++` claim to convert C and C++ to Java code. Their approach is a combination of lexical and parser-based, and is focussed only on reducing tedious editing when porting an application to Java. They use cpp to preprocess, and claim to only convert about 85% of the code correctly.

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

[11] Stroustrup also discusses differences between C and C++ which could cause other problems when re-targeting legacy C code to a C++ compiler [Str97, p. 816-820].

[12] Personal inquiries of that paper's authors also led to no pointers.

#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 [KS94].

6 Conclusion

PCp^3 is a flexible infrastructure for exposing C preprocessor manipulations while performing an accurate parse and creating an abstract syntax tree. Though its applications are immature, the framework holds great promise due to the ease of programming in its action language, and the large number of details offloaded from the analysis itself. Nevertheless, there are several problems with the PCp^3 framework as it currently exists.

First, sophisticated analyses tend to be dependent on the order that action hooks are called. This in turn requires an intimate understanding of the tool, and its preprocessing and parsing peculiarities. Fortunately, the fast development time provided by the scripting language permits easy exploration and testing.

The hooks and backcalls interfaces are incomplete. It would be useful to have a hook associated with each parse rule. This might best be done by modifying `YACC` to insert code to call a Perl hook in each of the C actions it outputs when generating the parser. Additionally, not all of the data structures from the parser and preprocessor are made available to the Perl code; ideally more of these would be shared through "magic" Perl variables, instead of the existing function call interface.

The abstract syntax tree provides a significant area for enhancements. Currently, the AST that the `CTree` parser constructs is largely unused. More backcalls and Perl utility subroutines need to 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 would need to be constructed so that preprocessor-level annotations and constructs could be represented. Using the `Tk` widget set's Perl interface could provide a reasonable platform for experimentation with visualizing and manipulating the AST.

Better support for versioning would be useful. The mechanisms for handling multiple configurations are primitive—there is a distinguished path (a version) through the source code which 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 `#ifdef`ed out to be properly influenced by prior blocks of code using the same guard. Krone and Snelting's work suggests that the number of distinct paths through the source is reasonably bounded [KS94]. A generalized symbol table could encapsulate information about which versions contain each symbol, and how the type depends on macro names used in conditional compilation guards. A similar generalization could be made for the preprocessor name table.

Because PCp^3 works like a compiler, it handles only a single translation unit at a time. This complicates whole-program analysis. The Emacs-based text annotations provides only rudimentary support for combining information derived from various translation units. To better support reasoning about an entire source code artifact, a database approach similar to `CIA++` [GC92] 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 [AG96]. Extending PCp^3 to permit specified variables to persist between invocations on separate translation units could be implemented easily by reusing one of the existing data persistency modules, and would provide such flexibility.

Converting to language features need not be limited to using C++. Many are interested in converting legacy C code into Java code. Such transformations would also greatly benefit from a generalized AST since the TXL paradigm of parse, process, unparse [CC93] seems more appropriate for converting between source languages with significant differences.

Although no tuning has been done yet, PCp^3 exhibits performance comparable to an optimizing compiler. For gcc to compile and optimize the 5,000 lines of the bc package (an arbitrary precision arithmetic language interpreter) required 38 seconds on the author's Pentium Pro 200 Linux machine. Running PCp^3 with its test analysis consisting of 600 lines of hooks which exercise every event required 4 minutes for the bc source. Removing hooks for CPP_OUT and TOKEN (the most frequently invoked hooks) reduced the running time to 50 seconds. With all action code turned off, the running time was reduced by a factor of 5, requiring less than 10 seconds. Using the --noparse option hardly reduced the running time at all.

The PCp^3 framework distinguishes itself from other software engineering tools by providing an accurate parse without disregarding the significant impact of 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 software engineer. This approach empowers analyses, and provides a foundation to further investigate automating the translation of cpp constructs into language features. This reduction in preprocessor usage will benefit both software engineers trying to understand their code, and also other tools which take a less complete approach to analysis.

Acknowledgments: Thanks to David Notkin and Michael Ernst for their insight and guidance through this project. Also, thanks to Douglas Zongker, Brian Michalowski, and Craig Kaplan for their thoughtful comments on early drafts of this paper.

A Action hooks called by PCp[3]

- `STARTUP` ()
  Called immediately after the perl parser is initialized.
- `STARTPARSE` ()
  Called just before the bison parser is started (i.e., before yyparse()).
- `EXIT` (`$return_exit_code`)
  Called just before the system exit() call.
- `HANDLE_DIRECTIVE` (`$directive_name`)
  Called exactly once for each directive. $directive_name is the directive as it appeared in the source code with the leading # removed.
- `DO_DEFINE` (`$s_start, $s_end, $name_args_body`)
  Called exactly once for each #define. Arguments give the source code character offsets of the line and the unprocessed rest of the line including the name of the macro, its arguments and its definition.
- `PRE_DO_UNDEF` (`$s_start, $s_end, $mname`)
  Called exactly once for each #undef. Arguments give the source code character offsets of the line and the name of the macro being undefined. This hook is called just before the macro is undefined, so its definition still exists in the preprocessor's state.
- `DO_UNDEF` (`$s_start, $s_end, $mname`)
  Called exactly once for each #undef. Arguments give the source code character offsets of the line and the name of the macro being undefined. This hook is called immediately after the macro is undefined.
- `DO_INCLUDE` (`$s_start, $s_end, $filename_given, $filename_resolved, $flags`)
  Called exactly once for each #include, #include_next, or #import directive. Source coded character offsets of the line are given. Also, $filename_given is the name of the file to be included as written in the source; $filename_resolved is the fully qualified path name of the file to be read; and $flags is a bitmap with three relevant masks: $ANGLE_BRACKETS, for whether the filename appeared in angle brackets signifying a system include file; $SKIP_DIRS, for whether this is an #include_next directive; and $IMPORTING, for whether this is an #import directive.
- `DO_IF` (`$s_start, $s_end, $conditional, $skipped, $value`)
  Called exactly once for each #if (not #ifdef or #ifndef) directive seen by the preprocessor. In particular, note that normally, nested #ifXX-s that are ignored will not invoke this hook. Arguments give the source code character offsets of the directive; $conditional is the guard checked, $skipped is the literal text that is skipped (conditional was false if this is non-empty), and $value is 1 iff the $conditional evaluated to true (defined), 0 otherwise.
- `DO_XIFDEF` (`$s_start, $s_end, $kind, $conditional, $trailing_garbage, $skipped, $fSkipping, $s_-branch_start`)
  Called exactly once for each #ifdef or #ifndef. The arguments are the same as for the DO_IFDEF hook, except $kind is either "IFDEF" or "IFNDEF" depending on the type of conditional; $fSkipping replaces the $value argument and is non-zero iff the conditional fails (i.e., an IFDEF with a not-defined macro, or an IFNDEF with a defined macro); $s_branch_start gives the source code character offset of the start of the branch taken. Note that this hook is called after the DO_IFDEF or DO_IFNDEF hook.
- `DO_IFDEF` (`$s_start, $s_end, $conditional, $trailer, $skipped, $value`)
  Called exactly once for each #ifdef (not #if or #ifndef) directive seen by the preprocessor. In particular, note that normally, nested #ifXX-s that are ignored will not invoke this hook. Arguments give the source code character offsets of the directive; $conditional is the name checked for defined-ness; $trailer is whatever follows that name on the same line (usually empty); $skipped is the literal text that is skipped (conditional was false if this is non-empty), and $value is 1 iff the name from the $conditional was defined, 0 otherwise.
- `DO_IFNDEF` (`$s_start, $s_end, $conditional, $trailer, $skipped, $value`)
  Called exactly once for each #ifndef (not #if or #ifdef) directive seen by the preprocessor. In particular, note that normally, nested #ifXX-s that are ignored will not invoke this hook. Arguments give the source code character offsets of the directive; $conditional is the name checked for defined-ness; $trailer is whatever follows that name on the same line (usually empty); $skipped is the literal text that is skipped (conditional was false if this is non-empty), and $value is 1 iff the name from the $conditional was not defined, 0 otherwise.
- `DO_ELSE` (`$s_start, $s_end, $orig_conditional, $trailer, $skipped, $fSkipping, $s_start_branch`)
  Called exactly once for each #else directive seen by the preprocessor. In particular, note that if the directive is skipped due to another outer conditional, this hook does not get called. Arguments give the source code

character offsets of the directive; $orig_conditional is the guard in the matching #ifXX directive; $trailer is whatever follows #else on the same line (usually empty); $skipped is the literal text that is skipped (if any); $fSkipping is 1 iff this #else clause's guarded text got skipped; $s_start_branch is a souce code character offset of where the branch begins, if it's used, or the character following the skipped text if not.

- `DO_ELIF ($s_start, $s_end, $already_did_clause, $conditional, $skipped, $value)`
  Called exactly once for each #elif directive seen by the preprocessor. In particular, note that if the directive is skipped due to another outer conditional, this hook does not get called. Arguments give the source code character offsets of the directive; $already_did_clause is 1 iff one of the earlier #ifXX or #elif-s succeeded (thus implying the code guarded will be skipped); $conditional is the conditional tested; $skipped is the skipped code (if any); and $value is what $conditional evaluated to. The code guarded is included iff !$conditional and $value, in which case $skipped will be empty.

- `DO_ENDIF ($s_start, $s_end, $orig_conditional, $trailer)`
  Called exactly once for each #endif directive seen by the preprocessor. In particular, note that if the directive is skipped due to another outer conditional, this hook does not get called. Arguments give the source code character offsets of the directive; $orig_conditional is the guard of the matching #ifXX directive; $trailer is the text following the #endif (often empty);

- `CREATE_PREDEF ($mname, $expansion, $num_args, $internal_expansion, $file, $line, $r_argnames, $flags, $internal_expansion_args_uses... )`
  Called once for each predefined macro CPP installs. $mname is the name of the macro, $expansion is its expansion [as literal text], $num_args is the number of arguments it takes, $internal_expansion is an internal representation of the expansion (which is interpreted in the context of $internal_expansion_args_uses, see cpplib.c), $file and $line can safely be ignored, $r_argnames is a comma separated list of the argument names in reversed order, and $flags is a bitmask of two flags $flags & $PREDEFINED is true if this is a predefined macro (always for this hook) and $flags & $RESTARGS is true if this macro's last argument swallows remaining arguments (special gcc cpp feature).

- `CREATE_DEF ($s_start, $s_end, $mname, $expansion, $num_args, $internal_expansion, $file, $line, $r_argnames, $flags, $internal_expansion_args_uses... )`
  Called once for each macro definition read by CPP. $s_start and $s_end are the source code character offsets of the definition, $mname is the name of the macro, $expansion is its expansion [as literal text], $num_args is the number of arguments it takes, $internal_expansion is an internal representation of the expansion (which is interpreted in the context of $internal_expansion_args_uses, see cpplib.c), $file and $line give the filename and line number where the definition occurs, $r_argnames is a comma separated list of the argument names in reversed order, and $flags is a bitmask of two flags $flags & $PREDEFINED is true if this is a predefined macro (never for this hook) and $flags & $RESTARGS is true if this macro's last argument swallows remaining arguments (special gcc cpp feature).

- `DELETE_DEF ($mname, $fExists)`
  Called once for each macro name that is undefined for any reason (this is a low level hook called when you attempt to remove a macro name from the table of macros, also HI_DO_UNDEF for the high level hook). $mname is the name of the macro, and $fExists is 1 iff that macro name used to be defined.

- `SPECIAL_SYMBOL ($symbol, $enum_node_type)`
  Called once for each special symbol (e.g., __FILE__) that gets expanded. $symbol is the literal text of the symbol, $enum_node_type is an index into @enum_node_type giving the C constant corresponding to that symbol (usually better to just use $symbol instead).

- `EXPAND_MACRO ($s_start, $s_end, $mname, $expansion, $length, $raw_call, $has_escapes, $cbuffersDeep, $cnested, @nests, $cargs, @args)`
  Called once for each macro expansion in C source (i.e., not for expansions in #ifXX guards). $s_start and $s_end are the source code character offsets of the macro invocation; $mname is name of the macro, $expansion is what replaces $mname in the text (subject to further expansion, of course); $length is the number of characters in that expansion, $raw_call is how the call appeared in the source (includes the macro name), $has_escapes is 0 if the source buffer is actual source text (i.e., if this is a top level expansion), otherwise it is a 1 (escapes has marked with @- preceding the macro name); $cbuffersDeep is how many levels of expansion the text has undergone beyond the source text (0 means the macro occurrence appeared directly in the source code); $cnested tells how many elements are in @nests (so you can separate its arguments from @args); @nests is non-empty when this expansion is an argument to another macro—see the body of the paper for a complete explanation. $cargs tells how many arguments $mname takes; and @args contains argument expansion and use information. Each argument contributes 7 elements to @args plus a pair of elements for each use; e.g., if the first argument was used 3 times it would have 7 + 3*2

= 13 elements in @args describing its expansion. The first 6 arguments are $raw,$r_offset,$expanded,$e_offset,$stringified,$s_offset which correspond to the raw, expanded, and stringified representations of the argument. The $X_offset elements are internal offsets into a token buffer and can usually be ignored. $stringified may be empty if the argument was never stringified during the macro expansion. The seventh element for each argument tells how many times that argument was used in the expansion, and successive pairs of elements give character offsets to the appearance of those arguments in the expansion. For example if argument 1 is used 3 times, there could be (3, 5,6, 9,11, 14,18) as the final elements of the @args list. Note that because the third expansion is longer than the first two (4 characters instead of just two), it must have been stringified.

- `MACRO_CLEANUP` (`$s_start, $s_end, $mname, $c_nested, @nests`)
  Called once for each macro expansion buffer as it is removed from the stack of text buffers. $s_start and $s_end give the source code character position offset of the text which expanded into the text that is in the buffer currently being cleaned and pop-ped; $mname is the name of the macro that resulted in that expansion; $c_nested and @nests are obsoleted. Note that to get the full text of the expansion, you must track successive strings passed to the CPP_OUT hook, and reset the string used to accumulate the expansion when the CbuffersBack backcall is 1 in this hook (meaning you are cleaning up a top level macro expansion). See cpphook.pm for an example.

- `IFDEF_MACRO` (`$s_start, $s_end, $mname, $expansion, $length, $raw_call, $has_escapes, $cbuffersDeep, $cnested, @nests, $cargs, @args`)
  Called once for each macro expansion in `#`if directives. The arguments passed to the hook are the same as those for the EXPAND_MACRO hook, see its documentation for details. Note, though, that macros are not usually expanded in `#`ifdef and `#`ifndef directives, they are simply tested for defined-ness by lookup in the symbol table, thus triggering the IFDEF_LOOKUP_MACRO hook, not this one. A form such as "`#`if FOO" will trigger this hook, however.

- `IFDEF_LOOKUP_MACRO` (`$mname, $f_defined`)
  Called once for each macro name lookup done during conditional compilation guard checking. The arguments are: $mname is the macro name being looked up; $f_defined is non-zero iff the macro has been defined. Use backcalls to get the expansion if it is desired.

- `COMMENT` (`$s_start, $s_end, $comment_text, $how_terminated, $c_lines`)
  Called once for each comment in the source code seen by preprocessor (i.e., comments conditionally-compiiled out will not normally invoke this hook). $comment_text is the text of the comment, $how_term is "*/" for C-style comments, "nl" for a C++ style comment, or "eof" if the comment was ended by the end of file. $c_lines is the number of lines that the comment spans, including partial lines.

- `STRING_CONSTANT` (`$s_start, $s_end, $string, $lines`)
  Called once for each string in the source code or macro/special symbol expansion. Note that both stringization and symbols like __FILE__ will invoke this hook with their string expansions. Arguments give the source code character offsets of the string, $string is the string, and $lines is the number of lines (or partial lines) that the string spans. $s_start == $s_end if the string does not actually appear in the top level source code (e.g, __LINE__ or gcc's __FUNCTION__). The source code offsets include the string delimiters, the $string argument has them stripped, but has not had backslash sequences replaced.

- `CPP_ERROR` (`$nominal_fname, $line_number, $message, @extra_args`)
  Called once each time CPP issues an error. The hook does not interfere with normal error handling. Arguments give the current filename and line number where the error occurred; $message is the string (containing printf percent-escapes) to be printed, and @extra_args are the values to be interpolated into the string.

- `CPP_WARN` (`$nominal_fname, $line_number, $message, @extra_args`)
  Called once each time CPP issues a warning. The hook does not interfere with normal warning handling. Arguments give the current filename and line number where the warning occurred; $message is the string (containing printf percent-escapes) to be printed, and @extra_args are the values to be interpolated into the string.

- `CPP_PEDWARN` (`$nominal_fname, $line_number, $message, @extra_args`)
  Called once each time CPP issues a pedantic warning. The hook does not interfere with normal pedantic warning handling. Arguments give the current filename and line number where the pedantic warning occurred; $message is the string (containing printf percent-escapes) to be printed, and @extra_args are the values to be interpolated into the string.

- `CPP_OUT` (`$string`)
  Called for each sequence of characters cpp is outputting. This is the same text as would go to the output

file when using just cpp (as in gcc -E). Full tokens and/or whitespace are emitted per each call to the hook. Use the CchOutput() and CchOffset() backcalls for output character offset and source code character offset, respectively.

- `ADD_IMPORT` (`$filename, $filedes_num`)
  Called for each filename that is `#`import-ed. Arguments give that file name and the number of the file-descriptor that file was opened on (negative iff the file was not found).

- `INCLUDE_FILE` (`$filename, $system_include`)
  Called for each file that is `#`include-d (not necessarily for each `#`include directive; this hook won't be called if the file cannot be found). Arguments are the filename and $system_include, which is non-zero if the filename appears to be a system include file (absolute pathname and in a known system directory); it is exactly 2 iff the file is a C-language system header file for which C++ should assume extern "C".

- `DONE_INCLUDE_FILE` (`$filename`)
  Called upon completion of parsing of an included file. The filename that what just finished being parsed is the only argument.

- `TOKEN` (`$token, $raw, $macro_name, $arg_num, @history`)
  Called once for each token CPP reads. $token is the type of token, with a leading CPP_; $raw is the raw characters which constitute that token; $macro_name is the name of the macro that expanded to create this token, or is empty if the token did not result from macro expansion; $arg_num is -1 if the token did not come from a macro expansion, 0 if it came from the body of a macro expansion (i.e., not an argument), or positive and equal to the argument number that produced the token, if from an substitution of an argument in a macro expansion; @history is obsoleted, use the MacroExpansionHistory() backcall instead. Note that the token "CPP_POP" means that an expansion buffer has been popped from the stack of expansions, and is not a real lexical token. This may be removed from future version so use the POP_BUFFER hook to handle those events.

- `FUNCTION` (`$name, $fStatic`)
  Called once after an entire function is parsed. Arguments give the name of the newly defined function, and $fStatic is non-zero iff the function was declared to be static (i.e. not global).

- `FUNC_CALL` (`$name`)
  Called once after each function call is parsed. Only argument is the name of the function being called.

- `TYPEDEF` (`$name`)
  Called once after each typedef is parsed. Argument is the name of the new type.

- `VARDECL` (`$name`)
  Called once after each variable declaration is parsed. Argument is the name of the declared variable.

- `ANNOTATE` ()
  Obsoleted

- `POP_BUFFER` (`$cbuffersDeep`)
  Called once for each buffer that is popped off of the stack of buffers to be parsed. The only argument is the new (after the pop) number of non-file buffers deep the stack is. In particular, when that number is 0, the parser is not parsing a macro expansion any longer.

- `POP_PERL_BUFFER` (`$cbuffersDeep`)
  Called once for each perl-pushed buffer that is popped off of the stack of buffers to be parsed (i.e., those pushed with the PushBuffer() backcall) The only argument is the new (after the pop) number of non-file buffers deep the stack is. In particular, when that number is 0, the parser is not parsing a macro expansion any longer.

B Subroutines provided by PCp^3

- `SzToken` (`$id`) **returns** `$token_type`
  Returns a string identifying what kind of token *id* (an integer) represents. Gives the source code character position offset and the number of expansions deep that we currently are.
- `CbuffersBack` () **returns** `$cbuffersback`
  Return the number of macro expansions deep that we currently are in the current expansion. This returns 0 if we are not expanding a macro. The number returned is the number of non-file buffers in the current stack of expansions (so it is not increased by nested #include-s).
- `MacroExpansionHistory` () **returns** `@expanded_macros_list`
  Returns a list of strings of the form *macro_name*#*arg_num*[*cchOffsetEnd*]. Each string explains the contents of a current expansion buffer, and the first string is the top of the expansion stack. For example, if there is a single item in the returned list which is the string "MAX#1[3]", this means that the current token came from the expansion of macro MAX and was the first argument to that macro, ending at character offset 3. Multiple items in the returned list mean that a macro expanded to some other macro which was subsequently expanded. Argument number 0 signifies that the token came from the literal body of the macro. Returns the number of the argument from which the current token came. Prefer using the $argno parameter passed to the TOKEN hook.
- `CchOffset` () **returns** `$cch_source_offset`
  Returns the character position offset into the current source file. Use Fname() to get the source file name.
- `InFname` () **returns** `$filename`
  Returns a string giving the name of the main input file (the one that appeared on the command line.
- `Fname` () **returns** `$filename`
  Returns a string giving the name of the current input file. This tracks include-d files and, combined with CchOffset, gives a unique location in the package. "@NONE@" returned if no current file.
- `FnameNominal` () **returns** `$filename`
  Returns a string giving the filename corresponding to the current buffer. This will be "@NOFILE@" if there is no such file (as is the case when reading tokens from a macro expansion). Otherwise it will correspond to Fname().
- `ExpansionLookup` (`$macro_name`) **returns** `$encoded_expansion`
  Returns the internally-encoded (cpplib-specific) expansion of the macro with name `$macro_name`. Note that this the string returned is delimited on either end by "@ " and omits argument occurrences. "@NOT-FOUND@" is return if there is no current macro of the given name.
- `CchOutput` () **returns** `$cchOutput`
  Returns the number of characters already output. Only non-zero when –noparse option is given. Returns the number of characters that have been read from the input files. Prefer using the offsets passed in the various hooks.
- `FExpandingMacros` () **returns** `$fExpandingMacros`
  Returns TRUE iff macros are currently being expanded. Returns FALSE otherwise. As arguments are read, macros are not expanded, and this will return FALSE.
- `ParseStateStack` () **returns** `@parse_state_list`
  Returns the current stack of parse state numbers. The first element is the top state on the stack. See the gram.output file for a listing of what the numbers correspond to.
- `SetParseStateStack` (`@parse_state_list`)
  Resets the current stack of parse states so that it is @parse_state_list. This has the potential to break the parse, and often will outside of the trivial case of setting the stack of states to what it already is.
- `PushBuffer` (`$source_code_string, $s_start`)
  Adds $source_code_string to the being-processed text as if it existed in the input at the current location in the current file. $s_start tells where the code to be processed is in the current source file, or is negative if it does not exist there.
- `EnterScope` ()
  Enters a new scoping level; useful in combination with PushBuffer() for causing code to be parsed without affecting the current symbol table directly.
- `ExitScope` ()
  Exits the current scoping level; useful in combination with PushBuffer() for causing code to be parsed without affecting the current symbol table directly.

- `PushHashTab` ()
  Copies the current cpp hash table and begins using the copy; the copy can later be thrown out using PopHashTab()
- `PopHashTab` ()
  Throw out a previously pushed cpp hash table; revert to the prior hash table
- `SetParseDebugging` ()
  Set the yydebug flag to TRUE for the underlying parser. This results in extra state change information being sent to stderr. It has no affect if –noparse is used. Initially, parser debugging is off.
- `ResetParseDebugging` ()
  Reset the yydebug flag to FALSE for the underlying parser. This terminates sending extra state change information to stderr. It has no affect if –noparse is used.
- `YYPushStackState` ()
  Save the entire current state of the parse stack onto a meta stack.
- `YYPopAndRestoreStackState` ()
  Restore the entire current state of the parse stack from the meta stack.
- `YYPopAndRestoreStackState` ()
  Throws out the top state stack from the meta stack.
- `YYSwapStackState` ()
  Switch the current stack state and the one on the top of the meta stack.
- `YYPushDupTopStackState` ()
  Push another copy of the top element of the meta stack of stack states onto the meta stack.
- `YYFCompareTopStackState` () **returns** `$fStacksEqual`
  Return TRUE iff the top of the meta stack of state stacks is identical to the current state stack. Return FALSE otherwise.
- `FLookupSymbol` (`$symbol_id`) **returns** `$fDefined`
  Return TRUE iff $symbol_id is found in the current scope. Return FALSE otherwise.

References

[AG96] 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.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[CC93] James R. Cordy and Ian H. Carmichael. *The TXL Programming Language Syntax and Informal Semantics, Version 7*. Software Technology Laboratory, Queen's University at Kingston, Kingston, Canada, June 1993.

[CE95] Martin Carroll and Margaret Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, 1995.

[Com96] C++ Standards Committee. C++ standard, committee draft 2. Electronic web pages, December 1996. ftp://ftp.bby.com.au/pub/wphtml-cd2.tar.gz.

[CS92] James R. Cordy and Medha Shukla. Practical metaprogramming. In *Proceedings of the 1992 Centre for Advanced Studies Conference*, November 1992.

[Dav97] Mark Davis. Java cookbook: Porting C++ to Java. Taligent White Paper, 1997. http://www.taligent.com/Technology/WhitePapers/PortingPaper.

[Dev92] Prem Devanbu. GENOA – a customizable, language and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, May 1992.

[EBN97a] 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.

[EBN97b] 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.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[Eva96] David Evans. *LCLint User's Guide*. MIT Laboratory for Computer Science, Cambridge, MA, v2.2 edition, August 1996.

[Fla96] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.

[Fli97] Shaun Flisakowski. CTree distribution, v0.5. Freely available software package, July 1997. ftp://ftp.kagi.com/flisakow/ctree_05.tar.gz.

[Fou92] Free Software Foundation. Bison distribution, v1.25. Freely available software package, April 1992. ftp://prep.ai.mit.edu/pub/bison-1.25.tar.gz.

[Fou95] Free Software Foundation. Flex distribution, v2.5. Freely available software package, March 1995. ftp://prep.ai.mit.edu/pub/flex-2.5.tar.gz.

[Fou97] 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.

[GAM96] 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.

[GC92] Judith E. Grass and Yih-Farn Chen. The C++ information abstractor. In *Proceedings of the USENIX 1992 C++ Conference*, Portland, Oregon, August 1992.

[Gli97] Bob Glickstein. *Writing GNU Emacs Extensions*. O'Reilly & Associates, Inc., Sebastopol, California, 1997.

[Har91] Samuel P. Harbison. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, 3rd edition, 1991.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[KS94] 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.

[Laf97] Chris Laffra. C2J. Freely available software package, 1997. http://members.aol.com/laffra/c2j.html.

[Lev92] John R. Levine. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 1992.

[LLST94] 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.

[MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):263–291, July 1996.

[Ous97] John Ousterhout. Scripting: Higher level programming for the 21st century. Web document, March 1997. http://www.sunlabs.com/people/john.ousterhout.

[RD95] Steven P. Reiss and Tony Davis. Experiences writing object-oriented compiler front ends. Available from author, January 1995. Available from author.

[Sim97] Charles Simonyi. Personal conversation, 9 July 1997.

[Ste90] Guy L. Jr. Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[Til97] Ilya Tilevich. C2J++. Freely available software package, 1997. http://pacevm.dac.pace.edu/ ny971734/c2j.html.

[Wal97] Larry Wall. Perl language, v5.004. Freely available software package, June 1997. ftp://ftp.perl.com/pub/perl/src/CPAN/5.0/perl5.004.tar.gz.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the ACM conference on Programming Language Design and Implementation*, Alburquerque, NM, 1993.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.