An Empirical Analysis of C Preprocessor Use

Michael D. Ernst Greg J. Badros David Notkin

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350 USA
{mernst,gjb,notkin}@cs.washington.edu
18 January 2000

Abstract

This is the first empirical study of the use of the C macro preprocessor, Cpp. To determine how the preprocessor is used in practice, this paper analyzes 26 packages comprising 1.4 million lines of publicly available C code. We determine the incidence of C preprocessor usage — whether in macro definitions, macro uses, or dependences upon macros — that is complex, potentially problematic, or inexpressible in terms of other C or C++ language features. We taxonomize these various aspects of preprocessor use and particularly note data that are material to the development of tools for C or C++, including translating from C to C++ to reduce preprocessor usage. Our results show that while most Cpp usage follows fairly simple patterns, an effective program analysis tool must address the preprocessor.

The intimate connection between the C programming language and Cpp, and Cpp's unstructured transformations of token streams, often hinder programmer understanding of C programs and tools built to engineer C programs, such as compilers, debuggers, call graph extractors, and translators. Most tools make no attempt to analyze macro usage, but simply preprocess their input, which results in a number of negative consequences; an analysis that takes Cpp into account is preferable, but building such tools requires an understanding of actual usage. Differences between the semantics of Cpp and those of C can lead to subtle bugs stemming from the use of the preprocessor, but there are no previous reports of the prevalence of such errors. Use of C++ can reduce some preprocessor usage, but such usage has not been previously measured. Our data and analyses shed light on these issues and others related to practical understanding or manipulation of real C programs. The results are of interest to language designers, tool writers, programmers, and software engineers.

1 Coping with the Preprocessor

The C programming language [KR88, HS95] is incomplete without its macro preprocessor, Cpp. Cpp can be used to define constants, define new syntax, abbreviate repetitive or complicated constructs, support conditional compilation, and reduce or eliminate reliance on a compiler implementation to perform many optimizations. Cpp also permits system dependences to be made explicit, resulting in a clearer separation of those concerns. In addition, Cpp permits a single source to contain multiple different dialects of C, such as both K&R-style and ANSI-style declarations.

While disciplined use of the preprocessor can reduce programmer effort and improve portability, performance, or readability, Cpp is widely viewed as a source of difficulty for understanding and modifying C programs. Cpp's lack of structure — its inputs and outputs are token streams — engenders flexibility but allows arbitrary source code manipulations that may complicate understanding of the program by programmers and tools. In the worst case, the preprocessor makes merely determining the program text as difficult as determining the output of an ordinary program. The designer of the C++ language, which shares C's preprocessor, also noted these problems: "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].

Given the wide range of possible uses of the preprocessor, our research addresses the question of how it is actually used in practice. Our statistical analysis of 26 C programs comprising 1.4 million lines of code provides significant insights with respect to this question. We are not aware of any similar data or analysis in the literature.

We had three initial motivations for pursuing this line of research. First, we wanted to evaluate the potential for reducing preprocessor usage when converting a program from C to C++. Second, we wanted to know how difficult it would be to produce a framework for preprocessor-aware tools. Third, we wanted to develop a tool for identifying common pitfalls in the use of macros.

These motivations drove our selection of the data we extracted and the analyses that we performed. Our data, our analyses, and our insights take substantive steps towards addressing these three issues. Overall, our analysis confirms that the C preprocessor is used in exceptionally broad and diverse ways, complicating the development of C programming support tools. About two-thirds of macro definitions and uses are relatively simple, of the variety that a programmer could understand through simple but tedious effort or that a relatively unsophisticated tool could manage (although in practice very few even try). Though these simple kinds of macros predominate, the preprocessor is so heavily used that it is worthwhile to understand, annotate, or eliminate the remaining one-third of the macros; these are the macros which are most likely to cause difficulties.

Different programmers have different backgrounds and different tasks. These differences lead to substantial variations not only in how programmers use the preprocessor, but also in their attitudes towards the preprocessor and what data about preprocessor use they find of interest. We have found repeatedly that data that confirms one person's intuition comes as a surprise to another — and every component of our data has had both effects on different individuals. Therefore, we provide a broad range of analyses with the expectation that different readers will focus on different parts, and will perhaps choose to extend our work in specific directions.

Section 2 provides additional detail about the difficulties imposed by the preprocessor. Section 3 describes our experimental methodology. Sections 4–7 present the bulk of results about macro preprocessor use. Section 8 discusses related work, and Section 9 suggests techniques for mitigating the negative impact of Cpp on program understanding. Section 10 presents avenues for future work, and the concluding section discusses the relevance of the research.

2 Background

Tools — and, to a lesser degree, software engineers — have three options for coping with Cpp. They may ignore preprocessor directives altogether, accept only post-processed code (usually by running Cpp on their input), or attempt to emulate the preprocessor by tracking macro definitions and the value of conditional compilation tests. Each approach has different strengths and weaknesses.

- Ignoring preprocessor directives is an option for tools that produce approximate information, such as those based on lexical or approximate parsing techniques. However, if accurate information about function extents, scope nesting, declared variables and functions, and other aspects of a program is required, the preprocessor cannot be ignored.

- Operating on post-processed code, the most common strategy, is simple to implement, but the tool's input differs from what the programmer sees. Even when line number mappings are maintained, other information is lost in the mapping back to the original source code. Source-level debuggers have no symbolic names or types for constants and functions introduced via `#define`, nor can tools trace or set breakpoints in function macros, as they can for ordinary functions (even those that have been inlined [Zel83]). An example of a tool working on the post-processed code is the use of type inferencing to produce C++ function templates from C; however, the input "has been preprocessed so that all include files are incorporated and all macros expanded" [SR96, p. 145]. Such preprocessing may limit the readability of the resulting C++ templates by converting terse, high-level, well-named constructs into verbose, low-level code. Preprocessing may also limit reusability: the macro-expanded code will perform incorrectly when compiled on a system with different settings for the macros. Another example is that call graph extractors generally work in terms of the post-processed code, even when a human is the intended consumer of the call graph [MNL96].

A tool that manipulates post-processed code cannot be run on a program that will not preprocess on the platform on which the tool is being run. Some such tools also reject ill-formed programs that will not compile without errors. These constraints complicate porting and maintenance, two of the situations in which program understanding and transformation tools are most likely to be needed. Additionally, a tool supplied with only one post-processed instantiation of the source code cannot reason about the program as a whole, only about the version that results from one particular set of preprocessor variables. For instance, a bug in one configuration may not be discovered despite exhaustive testing or analysis of other configurations.

- The final option, emulating the preprocessor, is fraught with difficulty. Macro definitions consist of complete tokens but need not be complete expressions or statements. Conditional compilation and alternative macro definitions lead to very different results from a single original program text. Preprocessing adds complexity to an implementation, which must trade off performing preprocessing against maintaining the code in close to its original form. Extracting structure from macro-obfuscated source is not a task for the faint-hearted. Despite these problems, in many situations only some sort of preprocessing or Cpp analysis can produce useful answers.

Choices among these options are currently made in the absence of an understanding of how Cpp is used in practice. While Cpp's potential pitfalls are well-known, no previous work has examined actual use of the C preprocessor to determine whether it presents a practical or merely theoretical obstacle to program understanding, analysis, and modification. This paper fills that gap by examining Cpp use in 26 programs comprising 1.4 million lines of source code.

The analysis focuses on potential pitfalls that complicate the work of software engineers and tool builders:

high total use (Sections 4, 6.1, and 7): Heavy use of either macro substitution or conditional compilation can overwhelm a human or tool. Lines that depend on many macros or macros that affect many lines are more likely to be problematic.

complicated bodies (Section 5.1): A macro body need not expand to a complete C syntactic entity (such as a statement or expression).

extra-linguistic features (Section 5.2): A macro body may exploit features of the preprocessor not available in C, such as stringization, token pasting, or use of free variables.

macro pitfalls (Section 5.3): Macros introduce new varieties of programming errors, such as function-like macros that fail to swallow a following semicolon and macros that modify, or fail to parenthesize, uses of their arguments.

multiple definitions (Sections 5.4–5.6): Uncertainty about the expansion of a macro makes it harder to confidently understand the actual program text. Even more problematically, two definitions of a macro may be incompatible, for instance if one expands to a statement and the other to an expression or type.

inconsistent usage (Section 6.2): A macro used both for conditional compilation and to expand code is harder to understand than one used exclusively for one purpose.

mixed tests (Section 6.3): A single Cpp conditional (`#if` directive) may test conceptually distinct conditions, making it difficult to perceive the test's purpose.

variation in use: Preprocessor use varies widely. In the absence of a clear pattern of use or commonly-repeated paradigms, no obvious point of attack presents itself to eliminate most complexity with little effort. We examine this issue throughout the article.

We report in detail on each of these aspects of preprocessor use, indicating which are innocuous in practice and which problematic uses appear more frequently. We also taxonomize macro bodies, macro features, macro errors, and conditional tests. These taxonomies are more detailed, and they more accurately reflect actual use, than previous work.

Package	Version	Physical lines	NCNB lines	Files	Description
bash	1.14.7	55079	38111	128	Command shell
bc	1.03	11193	8026	28	Desktop calculator
bison	1.25	10799	7260	29	Parser generator
cvs	1.9	56902	39273	108	Revision control system
emacs	19.34	132929	89335	115	Text editor
flex	2.5.3	15475	10648	17	Scanner generator
fvwm	2.0.43	42953	32517	111	Window manager
gawk	2.15.6	21291	14963	22	GAWK interpreter
gcc	2.7.2.2	346395	235237	193	C and C++ compiler
genscript	1.3.2a	11546	7969	23	Text-to-PostScript converter
ghostview	1.5	11214	8762	22	PostScript previewer
gnuchess	4.0pl77	15183	12532	28	Chess player
gnuplot	3.50.1.17	40800	30247	71	Graph plotter
gs	5.10	182933	127136	594	PostScript interpreter
gzip	1.2.4	8148	5186	19	File compressor
m4	1.4	15316	9386	22	Macro expander
mosaic	2.6	82791	55194	190	WWW browser
perl	5.003	69722	61090	82	Perl interpreter
plan	1.7.1	30894	24439	77	Schedule planner
rasmol	2.5	21863	17845	23	Molecular visualization
rcs	5.7	18444	12134	29	Revision control system
remind	3.00.16	15611	11086	29	Schedule reminder
workman	1.3	13486	9928	58	Audio CD player
xfig	3.1.4	52400	41259	118	Drawing program
zephyr	2.0.4	42008	28016	240	Notification system
zsh	3.0.5	47244	36298	75	Command shell
Total		1372619	973877	2451	26 packages

Figure 1: Analyzed packages and their sizes. NCNB lines are non-comment, non-blank lines. All of these packages are publicly-available for free on the Internet. We give version numbers to permit reproducing our results.

3 Methodology

We used programs we wrote to analyze 26 publicly-available C software packages that represent a mix of application domains, user interface styles (graphical vs. text-based, command-line vs. batch), authors, programming styles, and sizes. We intentionally omitted language-support libraries such as `libc`, for they may use macros differently than application programs do.

Figure 1 describes the packages and lists their sizes in terms of physical lines (newline characters) and non-comment, non-blank (NCNB) lines. The NCNB figure disregards lines consisting of only comments or whitespace, null preprocessor directives ("`#`" followed by only whitespace, which produces no output), and lines in a conditional that cannot evaluate to true (such as `#if 0 && anything`; all our analyses skip over such comments, which account for 0.9% of all lines). All of our per-line numbers use the NCNB length.

We built each package three times during the course of our analysis, then ran our analysis programs over a marked-up version of the source code created by the third build. The first compilation was a standard build on a RedHat-4.x-based (libc5) GNU/Linux 2.0.x system to generate all the source files for the package. For example, the `configure` script prepares a package for compilation by creating header files such as `config.h`, and often other source files are also automatically generated.[1] The second compilation identified global variables. We made all variables non-`static` (by making `static` a preprocessor macro with an empty expansion), then recompiled (but did not link, as it would likely fail because of multiple definitions of a variable) and used the `nm` program to read the global symbols from the resulting object files and executables. (Static file-global variables would not have shown up in the compilation output.)

[1]In a few cases we modified the build process to preserve some files; for instance, the Makefile for `gs` creates `gconfig.h`, compiles some packages, and then deletes it once again. Our tools automatically informed us of such problems by detecting a use of a macro without any corresponding definition (which was in the deleted file).

The third compilation used PCp^3 [BN98], an extensible version of the C preprocessor, in place of a compiler. This step has three main purposes. First, it identifies the code and header files processed by the compiler (see below for details). Second, it saves information regarding the flags passed to the compiler that indicate which preprocessor macros are defined or undefined on the command line. Third, our extension to PCp^3 creates a copy of the source code in which identifiers that are possibly macro-expanded are specially marked (but no macros are expanded or conditionals discharged). This PCp^3 analysis performs a conservative reaching-definitions analysis: it examines both sides of every Cpp conditional, and an identifier is marked as possibly-expanded if any `#define` of the identifier occurs before the use site (even if on only one side of a Cpp conditional), unless a `#undef` definitely was encountered subsequently (outside any conditional or on both sides of a conditional).

While we examine code in all Cpp conditionals that may evaluate to true, our PCp^3 analysis treats `#include` directives just like the real preprocessor does: the included file is processed only if any Cpp conditional guards evaluate to true. We thus avoid attempts to include header files not present on our system. This analysis examines only files actually compiled into the application; it omits platform-dependent (e.g., MS-DOS or VMS) files and source code used only during the build process. As a result, we do not see multiple versions of system-dependent macros, but we do analyze all possible configurations of a software package under one operating system and hardware setup.

After marking the potentially expanded macros, we processed all the source files using a program that we wrote for this study. This program collects the statistics about macro definitions, uses, and dependences that are reported in the paper; more details of its operation are reported as appropriate along with those results. Our tool includes approximate, Cpp-aware parsers for expressions, statements, and declarations. It performs approximate parsing because the input may not be a valid C program; as a result, we may misinterpret some constructs, but we can cope with uncompilable C and with partial constructs in conditional compilation branches.

The results reported in this paper omit macros defined only in external libraries (such as the `/usr/include/` hierarchy), even when used in the package source code; we also omit all macro uses in libraries. There are many such macros and uses; omitting them prevents library header files and uses of macros defined in them from swamping the characteristics of the package code, which is our focus in this study. The programmer generally has no control over libraries and their header files, and may not even know whether a library symbol is defined as a macro.

The raw data, which includes considerable data not reported here, and the programs used to generate and manipulate them, are available from the authors. The packages are widely available on the Internet or from the authors.

4 Occurrence of preprocessor directives

Figure 2 shows how often preprocessor directives appear in the packages we analyzed. The prevalence of preprocessor use makes understanding Cpp constructs crucial to program analysis. Preprocessor directives make up 8.4% of program lines. Across packages, the percentage varies from 4.5% to 22%. These figures do not include the 25% of lines that expand a macro or the 37% of lines whose inclusion is controlled by `#if`; see Section 7.

Conditional compilation directives account for 48% of the total directives in all packages, macro definitions comprise 32%, and file inclusion makes up 15%. Packages are not very uniform in their mix of preprocessor directives, however. (If they were, each group of bars in Figure 2 would be a scaled version of the top group.) In particular, the prevalence of `#include` is essentially independent of incidence of other directives. The percentage of directives that are conditionals varies from 16% to 73%, the percentage of directives that are `#define`s varies from 18% to 45%, and the percentage of directives that are `#include`s varies from 3.5% to 49%. This wide variation in usage indicates that a tool for understanding Cpp cannot focus on just a subset of directives.

4.1 `#line`, `#undef`, and other directives

The definedness of a macro is often used as a boolean value. However, `#undef` is rarely used to set such macros to false: 32% of `#undef` directives precede an unconditional definition of the just-undefined macro,

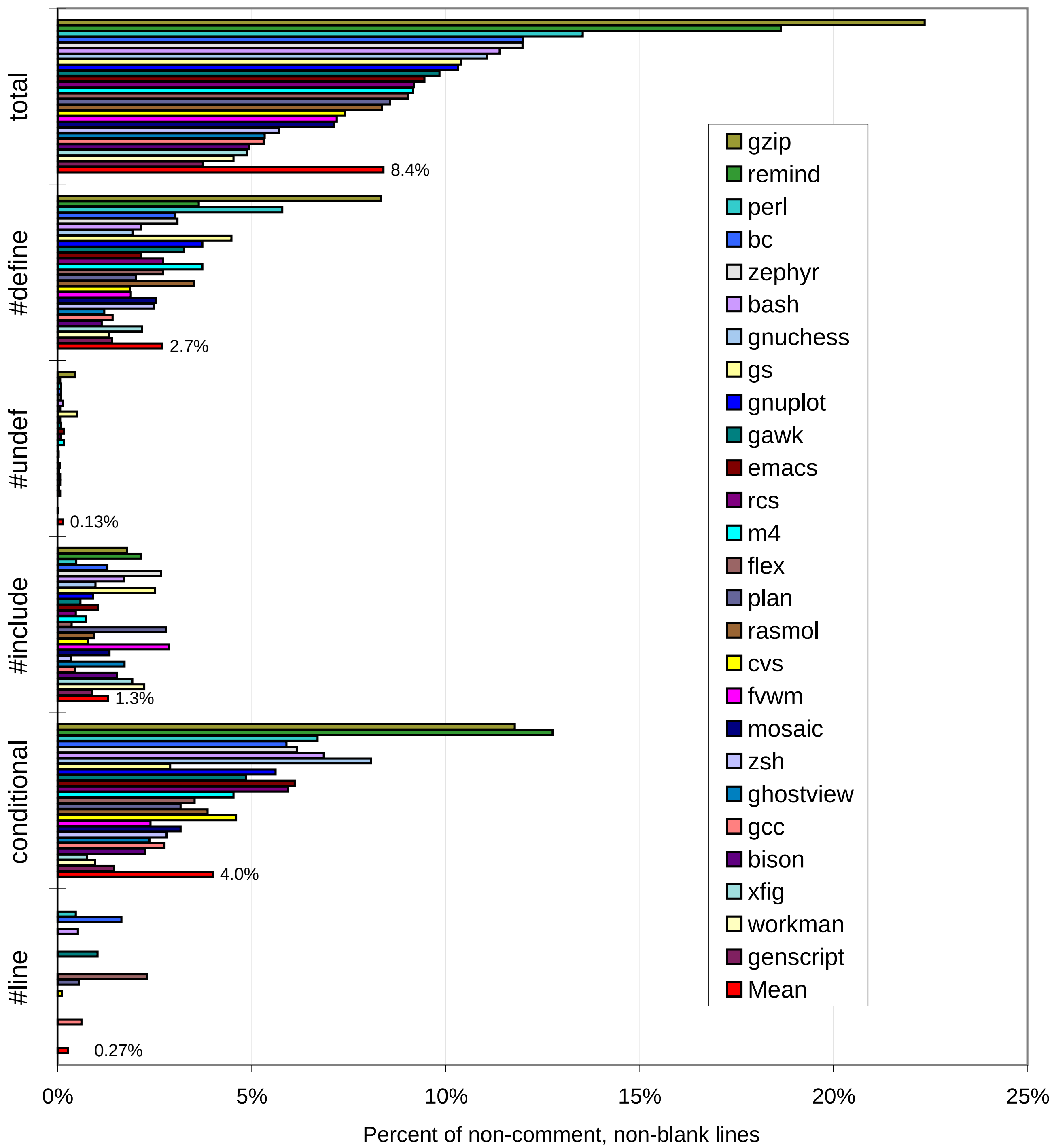

Figure 2: Preprocessor directives as a fraction of non-comment, non-blank (NCNB) lines. Each group of bars represents the percentage of NCNB lines containing a specific directive. Conditional compilation directives (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`) are grouped together. For example, the top bar of the fourth group indicates that 1.8% of gzip's NCNB lines are `#include`s, and the bottom bar of the fifth group indicates that 4.0% of all lines across the packages are conditional compilation directives.

generally to avoid preprocessor warnings about incompatible macro redefinitions, and 43% of #undef directives unconditionally follow a definition of the macro, with 81% of such uses in gs alone. This usage limits a macro definition to a restricted region of code, effectively providing a scope for the macro. When such macros appear in the expansions of macros used in the code region, the result is a kind of dynamic binding.

Every use of #line appears in lex or yacc output that enables packages to build on systems lacking lex, yacc, or their equivalents. For instance, flex uses itself to parse its input, but also includes an already-processed version of its input specification (that is, C code corresponding to a .l file) for bootstrapping.

Figure 2 omits unrecognized directives and rarely-appearing directives such as #pragma, #assert, and #ident. Among the packages we studied, these account for 0.017% of directives.

4.2 Packages with heavy preprocessor use

The gzip, remind, and perl packages deserve special attention for their heavy preprocessor usage — 22%, 19%, and 14% of NCNB lines, respectively.

gzip #defines disproportionately many macros as literals and uses them as system call arguments, enumerated values, directory components, and more. These macros act like const variables. gzip also contains many conditional compilation directives, since low-level file operations (such as accessing directories and setting access control bits) are done differently on different systems. In bash, which is also portable across a large variety of systems, but which uses even more operating system services, 97% of the conditional compilation directives test the definedness of a macro whose presence or absence is a boolean flag indicating whether the current system supports a specific feature. The presence or absence of a feature requires different or additional system calls or other code.

remind supports speakers of multiple natural languages by using #defined constants for basically all user output. It also contains disproportionately many conditional compilation directives; 55% of these test the definedness of HAVE_PROTO, in order to provide both K&R and ANSI prototypes.

perl's high preprocessor usage can be attributed in part to #define directives, which make up 43% of its preprocessor lines. Of these, 38% are used for namespace management, to permit use of short names in code without colliding with libraries used by extensions or applications that embed perl. perl also frequently uses macros as inline functions or shorthand for expressions, as in

```
#define sb_iters cx_u.cx_subst.sbu_iters
#define AvMAX(av) ((XPVAV*)  SvANY(av))->xav_max
```

5 Macro definition bodies

This section examines features of macro definitions that may complicate program understanding. The results indicate the necessity and difficulty of a thorough understanding of macro definitions to a software engineer or tool. For example, 12% of macro bodies expand to a partial or unidentifiable syntactic entity (not a symbol, constant, expression, or statement; see Section 5.1); 14% of macros take advantage of Cpp features that lie outside the C programming language (see Section 5.2); and 23% of macros contain latent bugs (see Section 5.3). The second half of this section considers macro names with multiple definitions, which can also complicate understanding. In the packages we examined, 14% of macro names have multiple definitions (see Section 5.4), although only 8.9% of macro names have definitions with different abstract syntax trees (see Section 5.5). Of macros with multiple definitions, 4% have syntactically incompatible definitions that cannot be substituted for one another in code (see Section 5.6). Given categorizations (according to Section 5.1) of macro definitions, Section 5.4 shows how to categorize macro names with multiple definitions.

5.1 Macro body categorization

We categorized macro bodies into 28 categories, although for simplicity of presentation, this paper coalesces these into 10 higher-level categories, then omits one of them as insignificant. We started with a set of categories that we expected to occur frequently (similar to other macro taxonomies [Str94, CE95]), then iteratively refined them to break up overly broad categories and add unanticipated ones.

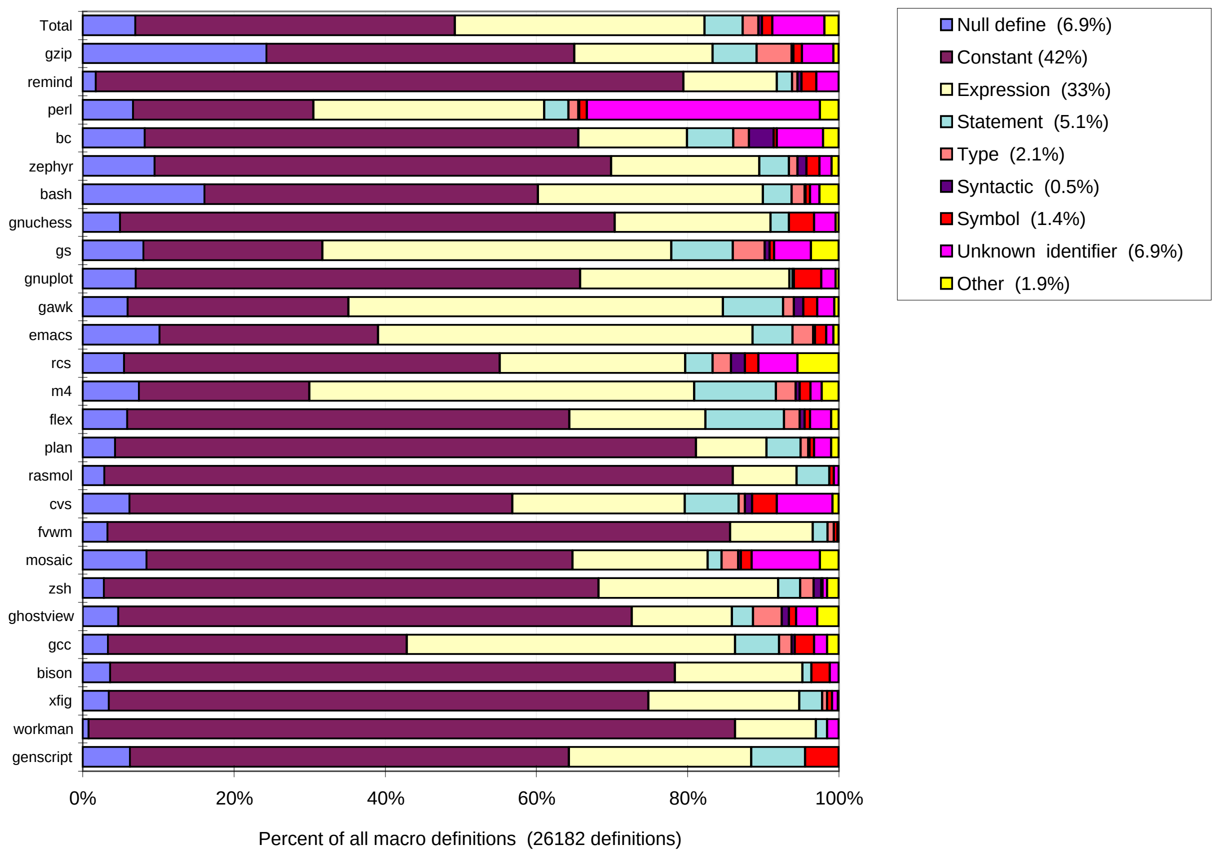

Figure 3: Categorization of macro definition bodies. The legend numerically represents the information in the top row. The packages are ordered from most preprocessor directives per line to fewest (as in Figure 2).

Figure 3 reports, for each package, how many definitions fall into each category. Macros that act like C language constructs — such as variables or functions — are easiest to analyze, understand, and perhaps even translate into other language constructs. Thus, the 75% of macros whose bodies are expressions and the 5.1% that are statements may be handled relatively easily by people and tools. Other macros, especially those that do not expand to a complete syntactic construct, are more problematic.

The ten macro body categories are as follows. Each example in this article comes from the packages studied.

Null define The macro body is empty, as in `#define HAVE_PROTO`. In Cpp conditionals, such macros frequently act as boolean variables. When expanded in code, they often represent optional syntax. For instance, macro `private` may expand either to `static` or to nothing, depending on whether a debugging mode is set.

Constant The macro body is either a literal (96% of this category) or an operator applied to constant values (4% of this category). For instance, `#define ARG_MAX 131072` and `#define ETCHOSTS "/etc/hosts"` define literals, while `#define RE_DUP_MAX ((1<<15)-1)` and `#define RED_COLS (1 << RED_BITS)` (where `RED_BITS` is a constant, possibly a literal) define constants. These macros act like `const` variables. This category includes both macros whose value is invariant across all configurations of the package and those that depend on other compile-time values.

Expression The macro body is an expression, as in `#define sigmask(x) (1 << ((x)-1))` or `#define mtime mailfiles[i]->mod_time`. Such a macro acts like a function that returns a value, though the macro need not take any arguments, so its uses may look syntactically unlike function calls.

Statement The macro body is a statement such as

```
#define SIGNAL_RETURN return 0
#define L_ORDINAL_OVERRIDE plu = ".";
#define FREE(x) if (x) {free(x); x=NULL;}
#define SWALLOW_LINE(fp) { int c; while ((c = getc(fp)) != '\n' && c != EOF); }
```

Such a macro is similar to a function returning `void`, except that when the macro body is a complete statement, its uses should not be followed by a semicolon.

To reduce the number of categories in our presentation, in this article, the statement category aggregates 6 categories that are distinct in our analysis: single complete statements (comprising 37% of the category), statements missing their final semicolon (49% of the category, as in `#define QUIT if (interrupt_state) throw_to_top_level()`), multiple statements (5.4%), multiple statements where the last one lacks its semicolon (6.5%), and partial statements or multiple statements where the last one is partial (together, 2.2% of the statement category, as in `#define ASSERT(p) if (!(p)) botch(__STRING(p)); else`).

Type The macro body is a type or partial type (55% of the category; for example, a storage class such as `static`), a declaration (2.4% of the category), or a declaration missing its terminating semicolon (43% of the category). Examples include

```
#define __ptr_t void *
#define __INLINE extern inline
#define FLOAT_ARG_TYPE union flt_or_int
#define CMPtype SItype
```

Partial types cannot be eliminated via straightforward translation (though C++ templates may provide some hope). Typedefs may be able to replace full types; for instance, `typedef void *__ptr_t` could almost replace the first example above. (Adding a `const` prefix to a use of `__ptr_t` would declare a constant pointer if the typedef version were used, instead of a pointer to constant memory.)

Syntactic The macro body is either punctuation (63% of this category; for example, `#define AND ;`) or contains unbalanced parentheses, braces, or brackets (37% of this category). The latter are often used to create a block and perform actions that must occur at its beginning and end, as for `BEGIN_GC_PROTECT` and `END_GC_PROTECT`; some other examples are

```
#define HEAPALLOC do { int nonlocal_useheap = global_heapalloc(); do
#define LASTALLOC while (0); if (nonlocal_useheap) global_heapalloc(); else global_permalloc()
#define end_cases() } }
#define DO_LARGE if ( pre->o_large ) { size = pre_obj_large_size(pre); {
#define DO_SMALL } } else { size = pre_obj_small_size(pre); {
```

Macros in this category are inexpressible as abstractions directly in the C programming language — they depend on the preprocessor's manipulation of uninterpreted token streams. They act like punctuation, or syntax, in the programming language. (C++ can enforce block exit actions via the destructor of a block-local variable.)

Symbol The macro body is a single identifier that is either a function name (95% of this category) or a reserved word (5% of this category, 65% of which are uses of variable names such as `true` or `delete` which are reserved words in another C dialect). Examples include

```
#define REGEX_ALLOCATE_STACK malloc
#define THAT this
```

A macro body that is a macro name inherits that macro name's categorization rather than appearing here.

Unknown identifier The macro expands to a single identifier that is not defined in the package or in any library header files included by the package. For example,

```
#define signal __bsd_signal
#define BUFSIZE bufsize
```

The symbol may be defined by compiler command arguments or may be used inside a conditional compilation guard because it is only meaningful with a particular architecture, system, or library (for which we did not have header files available).

Unknown identifiers can also be local variables, or variables or functions that we failed to parse. Our approximate parser can succeed where an exact parse would not (as for non-syntactic code that deviates from the language grammar or for entities interrupted by preprocessor directives), but occasionally it fails to recognize declarations or definitions.

In perl, unknown identifiers make up 31% of macro definitions, compared to 9.0% in mosaic, which has the the second largest proportion of unknown identifiers, and an average of 6.9%. In order to avoid link-time collisions when the Perl interpreter is embedded in another application, perl conditionally redefines over one thousand global symbols to add the prefix `Perl_`, as in `#define Xpv Perl_Xpv`.

Not C code The predominant use of such macros is for assembly code and for filenames and operating system command lines. The assembly code component includes only macros whose expansion is assembly code, not all expressions and statements that contain snippets of assembly code; however, we encountered such macros only in system libraries, not in the packages we examined.

Our tool processes only C source code, not Makefiles or other non-C files not compiled when a package is built. However, some header files are used both for code files, during the build process, to customize Makefiles[2]; it is those files that contribute macros expanding to filenames or command lines, as in

```
#define LIB_MOTIF -lXm -lXpm
#define LIB_STANDARD /lib/386/Slibcfp.a /lib/386/Slibc.a
```

The package-defined macros in this category all appear in emacs. They comprise only 17 definitions, 12 names, and 10 uses in C code files, so we omit them from all figures.

Other This category contains all macros not otherwise categorized. Of these macros, 12% either expand to a macro which was not defined in the package or have multiple definitions with conflicting categorizations (so that the macro being defined cannot be unambiguously categorized itself).

Some categorization failures resulted from limitations of our parser, which does not handle pasting (token concatenation) and stringization (converting a macro argument to a C string); together, these represent 3% of failures. Handling of partial entities is incomplete, so case labels (4% of failures), partial expressions (6%), and some declarations (14%) are left uncategorized, as are bodies that pass a non-first-class object (such as a type or operator) to another macro (7%). The biggest problem is macros with adjacent identifiers, which is generally not permitted in C (35% of macros). These macro bodies often use a macro argument that is a statement or operator or expand to a partial declaration or cast. Examples include

```
#define zdbug(s1) if (zdebug) syslog s1;
#define PNG_EXPORT(type,symbol) __declspec(dllexport) type symbol
#define gx_device_psdf_common gx_device_vector_common; psdf_distiller_params params
```

There were also some cases in which uses of macros categorized as "other" caused macro bodies in which they appeared to be similarly categorized.

[2]Cpp's specification states that its input should be syntactic C code, so it can avoid performing replacements in comments and string constants. In practice, uses of Cpp on non-C files have forced many C preprocessors to relax their assumptions about their input.

Our categorization is conservative: we categorize a macro body as (for example) a statement only if it can be definitively parsed as such, not merely if it has some tokens that predominantly appear in statements or if some arguments passed to it will result in a statement. As a result, our "other" category contains macros that might otherwise have been heuristically placed in some other category, at increased risk of miscategorizations of other macros. Because the number of categorization failures is small overall (1.9% of the 26182 definitions; 8 of the 26 packages have no such macros, and 10 more have only one or two such definitions), and because no variety of failure stands out among these classification failures, a more extensive categorization is unlikely to affect our conclusions.

5.2 Extra-linguistic capabilities

The C preprocessor has capabilities outside the C programming language; indeed, this is a primary motivation for using Cpp. Such constructs can present special challenges to program understanding, and especially to reducing the use of the preprocessor by translation into C or C++. This section presents a list of extra-linguistic constructs whose effect is to provide a feature unavailable in the C language. (This is orthogonal to the "syntactic" category of Section 5.1, which contains macros that act like punctuation.) It also reports their frequency of appearance, both individually and in combination, in the code we analyzed. These capabilities are based on the effects of the macro's code, not the coarse categorization of its body; a full list of difficult-to-understand macros would include, for example, macros classified as syntactic as well as those described in this section.

We expected problems dealing with macros that use stringization (conversion of a preprocessor macro argument to a C string) and pasting (creating a new identifier from two existing identifiers), the two explicit extra-linguistic features of Cpp. However, these features appear in only 0.07% of all macro definitions. Far more prevalent, and more problematic for program understanding tools, is exploitation of Cpp's lack of structure to effect mechanisms not available in C. Cpp's inputs and outputs are uninterpreted token streams, so Cpp can perform unstructured transformations using non-first-class or partial syntactic constructs, such as types or partial declarations.

Overall, 14% of macros contain an extra-linguistic construct. Figure 4 breaks down these macros by the constructs they contain. In addition to showing the prevalence of each construct, the figure shows which ones occur together. The following list describes in detail the constructs appearing in Figure 4.

Free variables The macro body uses as a subexpression (that is, applies an operator or function to) an identifier that is not a formal argument, a variable defined in the macro body, a global variable, a reserved word, or a function, macro, or typedef name. Such identifiers are typically local variables at the site of macro invocation. Uses of local variables (in which the local definition in scope at the point of use captures the free variable in the macro body) can produce dynamic scoping, which C does not directly support. Examples of this usage include

```
#define INV_LINE(line) &invisible_line[L_OFFSET((line), wrap_offset)]
#define atime mailfiles[i]->access_time
```

Side effects The macro body directly changes program state via assignment (of the form `=`, *op*`=`, `--`, or `++`). Indirect side effects due to function or macro calls are not counted in this category. The side effect may be to a global variable, a variable local to the macro body, a macro parameter, or a local variable in scope at the point of macro invocation. Examples of the four varieties of side effects are:

```
#define INIT_FAIL_STACK() do { fail_stack.avail = 0; } while (0)
#define SWALLOW_LINE(fp) { int c; while ((c = getc(fp)) != '\n' && c != EOF); }
#define FREE_VAR(var) if (var) free (var); var = NULL
#define ADD_CMD(x) { if (cmd) specific_limits++; cmd |= (x); }
```

A macro that assigns a global variable presents little difficulties in understanding and may be translated into a C++ inline function. Assignments to variables local to the macro body is also easy to understand as the assignment may be ignored by users of the macro. A macro argument that is assigned to is similar to a pass-by-reference function argument and need only be noted in the macro's documentation;

Percentage of 26182 macro definitions	Free variables (8.5%)	Side effects (6.5%)	Use macro as type (1.4%)	Pass type as argument (0.46%)	Use argument as type (0.073%)	Pasting (0.038%)	Stringization (0.031%)	Self-referential (0.027%)
5.9% (1545)	■							
3.8% (995)		■						
2.5% (650)	■	■						
1.2% (303)			■					
0.39% (102)				■				
0.13% (35)		■	■					
0.057% (15)	■		■					
0.050% (13)	■			■				
0.034% (9)					■			
0.019% (5)		■	■		■			
0.019% (5)						■		
0.019% (5)							■	
0.015% (4)	■	■	■					
0.011% (3)	■	■				■		
0.011% (3)	■	■		■				
0.0076% (2)		■		■				
0.0076% (2)		■	■		■		■	
0.0076% (2)								■
0.0076% (2)		■	■					■
0.0076% (2)		■						■
0.0038% (1)	■	■			■			
0.0038% (1)			■		■			
0.0038% (1)	■				■			
0.0038% (1)		■				■		
0.0038% (1)	■	■				■	■	
0.0038% (1)			■					■

Figure 4: Usage of the extra-linguistic capabilities of the C preprocessor listed in Section 5.2. The table indicates usage of each feature and shows which features tend to be used together. These data assist in the interpretation of the overlapping uses (the sums of the column totals are larger than the total number of macros with any extra-linguistic feature). The features are listed across the top, along with the percentage of macro definitions exploiting each. Each row of the table reports the percentage and absolute number of macro definitions that use a particular combination of the capabilities, indicated by black squares. For instance, the sixth line indicates that 35 macro definitions — 0.13% of all definitions — both perform assignment and use the result of a macro invocation as a type, but use none of the other extra-linguistic features listed.

however, this may be unexpected, because C lacks reference arguments, so ordinarily a function call cannot change an argument's value. The remaining macro bodies with side-effects involve assignments to dynamically-bound variables. These macros make a bad situation worse: however unexpected dynamic binding is, modification of such variables is even more unexpected and harder to understand.

Use macro as type In this macro's body, the result of another macro invocation is used as a type — for instance, in a declaration or a type cast, as in

```
#define function_cell(var) (COMMAND *)((var)->value)
#define bhcd_declare_state hcd_declare_state;   int zeros
```

C cannot simulate this behavior, because its types are not first class and may not be passed to functions, returned as results, or otherwise manipulated.

Pass type as argument In this macro's body, a literal type is passed to another macro, as in `#define PTRBITS __BITS(char*)`. Like using a macro result as a type, this is impossible in C without the preprocessor.

Use argument as type This macro uses one of its arguments as a type, as in a declaration or cast. Like using a macro result as a type, this too is impossible in the C language proper.

```
#define dda_step_struct(sname, dtype, ntype) struct sname { dtype dQ; ntype dR, NdR; }
#define REVERSE_LIST(list, type) ((list && list->next) ? \
            (type)reverse_list ((GENERIC_LIST *)list) : (type)(list))
```

Not all uses can be unambiguously identified lexically, because our analysis focuses on macro definitions and potential uses, not only on uses that happen to appear in the program. For instance, the macro `#define MAKE_DECL(type, name) type name;` is not identified as necessarily using its first argument as a type, for it might be invoked as `MAKE_DECL(printf, ("hello world\n"))` or as `MAKE_DECL(x =, y+z)`.

Pasting The body uses symbol pasting (`##`), which treats its arguments not as tokens but as strings, constructing a new token out of their concatenation. After `#define _SIZEOF(x) sz_##x`, the macro invocation `_SIZEOF(int)` expands to the identifier `sz_int`. The resulting identifier might appear literally, or only as a pasted identifier, at its other uses. Pasting is often abstracted out into a separate macro — such as `#define __CONCAT(x,y) x ## y` — but such pasting macros are used less frequently than average in our test suite.

Stringization The body uses argument stringization (`#`), which replaces its argument (a preprocessor symbol) by the symbol's contents as a C string. After `#define FOO BAR BAZ`, the expression `#FOO` expands to `"BAR BAZ"`. Examples using stringization include

```
#define spam1(OP,DOC) {#OP, OP, 1, DOC},
#define REG(xx) register long int xx asm (#xx)
```

No C or C++ language mechanism can replace such macros. This feature is particularly useful in debugging, in order to record the exact operations being performed. Tables that interconvert internal names and strings can also be useful for serialization.

Self-referential The body refers to its own name, as in `#define LBIT vcat(LBIT)`. This feature can build a wrapper around an existing function or variable. Since the ANSI C preprocessor performs only one level of expansion on recursively defined macros, the expanded macro contains a reference to the original name. (Pre-ANSI implementations could loop forever when expanding self-referential macros.)

5.3 Erroneous macros

Differences between C's execution model and Cpp's macro expansion can give rise to unanticipated behavior from syntactically valid programming constructs. We call a macro erroneous if its functionality could be achieved by a C function, but in some contexts the macro behaves differently than that function would. Unlike the extra-linguistic constructs discussed in Section 5.2, these are generally bugs rather than uses of Cpp mechanisms to achieve results outside the abilities of the C language.

We verified that many current uses of erroneous macros in the packages we examined happen not to expand erroneously. For example, all arguments to a macro with a precedence error may be expressions with high precedence, or all arguments to a macro that evaluates its argument multiple times may be side-effect-free. However, future uses may well give rise to these dormant errors, especially by programmers not familiar with the (generally undocumented) caveats relating to use of each macro. If caveats for the macro's use were prominently documented (such as requiring arguments to be side-effect-free or parenthesized), then the macro definition could be argued to be error-prone rather than erroneous, but this was not the case in practice. We call the macros erroneous because they fail to adhere to their implied specification — the standard C execution model.

Because it flags such errors, our tool could play the role of a macro lint program. It discovered many more problems than we expected: 23% of all macro definitions triggered at least one macro lint warning, and 22% of macro names have a definition that triggers a warning. Figure 5 further breaks down the warnings, which are described below.

unparenthesized formal Some argument is used as a subexpression (i.e., is adjacent to an operator) without being enclosed in parentheses, so that precedence rules could result in an unanticipated computation being performed. For instance, in

any warning by name	4944	23%
any warning by definition	5768	22%
unparenthesized formal	2447	9.3%
multiple formal uses	2233	8.5%
free variables	2220	8.5%
unparenthesized body	1170	4.5%
dangling semicolon	535	2.0%
side-effected formal	333	1.3%
swallows else	245	0.93%
inconsistent arity by name	92	0.42%
null body with arguments	106	0.40%
bad formal name	19	0.072%

Figure 5: Macro lint: the frequency of occurrence of error-prone constructs in macro bodies. The second column gives absolute numbers and the third column gives percentages. Except where specifically noted in the leftmost column, the percentages refer to the number of macro definitions.

```
#define DOUBLE(i) (2*i)
... DOUBLE(3+4) ...
```

the macro invocation computes the value 10, not 14. This warning is suppressed when the argument is the entire body or is the last element of a comma-delimited list: commas have lower precedence than any other operator, making a precedence error unlikely. (C's grammar prohibits sequential expressions such as `a,b` as function arguments, so a precedence error can occur only for functions defined via the `varargs` or `stdarg` facilities that take multiple arguments.)

multiple formal uses Some argument is used as an expression multiple times, so any side effects in the actual argument expression will occur multiple times. Given a macro defined as

```
#define EXP_CHAR(s) (s == '$' || s == '`' || s == CTLESC)
```

an invocation such as `EXP_CHAR(*p++)` increments the pointer by three locations rather than just one, as would occur were `EXP_CHAR` a function. Even if the argument has no side effects, as in `EXP_CHAR(peekc(stdin))`, repeated evaluation may be unnecessarily expensive.

Some C dialects provide an extension for declaring a local variable within an expression. In GNU C [Sta96], this is achieved in the following manner:

```
#define EXP_CHAR(s) ({ int _s = (s); (_s == '$' || _s == '`' || _s == CTLESC) })
```

free variables Free variables are used to achieve dynamic scoping, as discussed in Section 5.2. We include them here because such uses can be error-prone; for instance, at some uses a macro may capture a local variable and other times a global variable, and it is difficult for a programmer to determine which is achieved, much less intended.

unparenthesized body The macro body is an expression that ought to be parenthesized to avoid precedence problems at the point of use. For instance, in

```
#define INCREMENT(i) (i)+1
... 3*INCREMENT(5) ...
```

the expression's value is 16 rather than 18.

This warning is applicable only to macros that expand to an expression (14% of expression macros contain the error) and is suppressed if the body is a single token or a function call (which has high precedence).

dangling semicolon The macro body takes arguments and expands into a statement or multiple statements (39% of statement macros contain the error), for instance by ending with a semicolon or being enclosed in `{ ... }`. Thus, its invocations look like function calls, but it should not be used like a function call, as in

```
#define ABORT() kill(getpid(),SIGABRT);
...
if (*p == 0)
  ABORT();
else ...
```

because `ABORT();` expands to two statements (the second a null statement), which is non-syntactic — disobeys the language grammar — between the `if` condition and `else`.

This warning is suppressed for macros without arguments (18% of statement macros), such as `#define FORCE_TEXT text_section();`, on the assumption that their odd syntax will remind the programmer not to add the usual semicolon.

The solution to this problem is to wrap the macro body in `do { ... } while (0)`, which is a partial statement that requires a final semicolon. To our surprise, only 276 macros (20% of statement macros) use this standard, widely-recommended construct.

side-effected formal A formal argument is side-effected, as in

```
#define POP(LOW,HIGH) do {LOW = (--top)->lo;HIGH = top->hi;} while (0)
#define SKIP_WHITE_SPACE(p) { while (is_hor_space[*p]) p++; }
```

This is erroneous if the argument is not an lvalue — a value that can be assigned to, like `a[i]` but unlike `f(i)`. A similar constraint applies to reference parameters in C++, which can model such macro arguments (though in C++ `f(i)` can be an lvalue if `f` returns a reference). While the compiler can catch this and some other errors, compiler messages can be obscure or misleading in the presence of macros, and our goal is to provide earlier feedback about the macro definition, not just some uses.

swallows else The macro, which ends with an `else`-less `if` statement, swallows any `else` clause that follows it. For instance,

```
#define TAINT_ENV() if (tainting) taint_env()
...
if (condition)
  TAINT_ENV();
else ...
```

results in the `else` clause being executed not if the condition is false, but if it is true (and `tainting` is false).

This problem results from a potentially incomplete statement that may be attached to some following information. It is the mirror of the "dangling semicolon" problem listed above, which resulted from a too-complete statement that failed to be associated with a textually subsequent token. The solution is similar: either add an else clause lacking a statement, as in

```
#define ASSERT(p) if (!(p)) botch(__STRING(p)); else
```

or, better, wrap statements in `{ ... }` and wrap semicolonless statements in `do { ...; } while (0)`.

An alternative solution would convert macros whose bodies are statements into semicolonless statements (wrapped in `do { ...; } while (0)`, as noted above). Invocations of such macros look more like function calls are are less error-prone. This solution requires notifying users of the change in the macro's interface and changing all existing macro uses.

inconsistent arity The macro name is defined multiple times with different arity; for example,

```
#define ISFUNC 0
#define ISFUNC(s, o) ((s[o + 1] == '(')  && (s[o + 2] == ')'))
```

This may indicate either a genuine bug or a macro name used for different purposes in different parts of a package, in which case the programmer must take care that the two are never simultaneously active (lest one override the other) and keep track of which one is active. The latter situation may be caught by Cpp's redefinition warnings, if the macro name is not subjected to `#undef` before the second definition.

null body with arguments The macro takes arguments but expands to nothing, of the form `#define name(e)`, which might have been intended to be `#define name (e)`. An empty comment is the idiomatic technique for indicating that the null definition is not a programming error, so a comment where the macro body would be suppresses this warning, as in

```
#define __attribute__(Spec) /* empty */
#define ReleaseProc16(cbp) /* */
```

bad formal name The formal name is not a valid identifier or is a reserved word (possibly in another dialect of C), as in

```
#define CR_FASTER(new, cur) (((new) + 1) < ((cur) - (new)))
```

This presents no difficulty to Cpp, but a programmer reading the body (especially one more complicated than this example) may become confused, and the code may not be as portable or easily translated to other dialects such as C++ where `new` is a keyword.

Our macro lint tool discovered a number of additional errors. There are some illegal constructs such as `#module` (which is not a meaningful Cpp directive) and `#undef GO_IF_INDEXABLE_BASE(X, ADDR)` (`#undef` takes a macro name, not the arguments as they appeared in the `#define` directive).

While ANSI C uses `##` for pasting, in K&R C one abuts two identifiers, with an empty comment in between, so that their expansions form a new identifier when the compiler removes the comment. For instance, in K&R C, `to/**/ken` is interpreted as a single token, and macros might be expanded on either side of the comment as well. This construct does not perform merging in newer implementations, so we warn users of its appearance. We do not report it in our statistics because use of `/**/`-style pasting is rarely a bug and is not uncommon, especially in `CONCAT` macros that provide portability across older and newer versions of the preprocessor.

A number of files we analyzed begin or end inside a brace scope or an `#if` scope. Some of these are intentional — as in files meant to be included by other files. Others are bugs (such as, in one case, a failure to close a `/* */` style comment) that were apparently not discovered because testing did not build the package under all possible configurations.

5.4 Multiple definitions

A package may contain multiple definitions of a macro, and a macro can even be redefined partway through preprocessing. Multiple compatible definitions of a macro do not complicate its use — such abstraction is often desirable. However, redefinitions make it harder to determine exactly which definition of a macro will be used at a particular expansion site, which may be necessary for program understanding or debugging; incompatible macro bodies introduce further complications. This section examines the frequency of macro redefinition, while the following sections consider whether multiple macro redefinitions are compatible with one another.

Our analysis does not distinguish sequential redefinitions of a macro from definitions that cannot take effect in a single configuration. Independent definitions may result from definitions in different branches of

	one configuration		all files	
	defs	differing defs	defs	differing defs
Null define	1.4	1.0	2.2	1.0
Constant	1.2	1.1	1.5	1.1
Expression	1.3	1.2	1.8	1.4
Statement	1.3	1.2	1.7	1.4
Type	1.5	1.3	2.2	1.5
Syntactic	2.1	1.6	3.2	1.7
Symbol	1.5	1.1	1.6	1.2
Unknown identifier	1.0	1.0	1.1	1.0
Other	1.7	1.5	5.9	3.7
Total	1.2	1.1	1.8	1.3

Figure 6: Average number of definitions of macro names in each category. The left pair of columns examines just the files that may be compiled on a RedHat-4.x-based (libc5) GNU/Linux 2.0.x system (as for all other values we report), whereas the right pair considers all C files in the package. The left column of each pair counts each definition, while the right column merges definitions that are identical modulo whitespace, comments, string and character literals, and formal argument names. The third line of the table indicates that macro names that are categorized as expressions have, on average, 1.3 different definitions in a single configuration, but those definitions include only 1.2 different abstract syntax trees. When we examine all files in the package, we find 1.8 definitions (1.4 different definitions) of each expression macro name. A macro name is categorized based on the categorizations of its definitions, as detailed in Figure 7.

a Cpp conditional, from intervening `#undef` directives, or from compilation conventions as when compiling different programs in a package or versions of a program.

Overall, 86% of macro names are defined just once; 10% are defined twice; 2.0% are defined three times; 0.8% are defined four times; and the other 1.2% are defined five or more times. The most frequently redefined macros are those most complicated to understand: the "other" and "syntactic" categories. The more definitions a macro has, the more likely it is that one of those definitions cannot be categorized, or is miscategorized, by our system, resulting in a failure to categorize the macro name. Syntactic macros include those expanding only to punctuation. These are frequently used to support variant declaration styles (such as ANSI C declarations and K&R C declarations); as such, they require a definition for each variety, and they are frequently redefined to ensure that their settings are correct.

The least frequently redefined macros are those categorized as unknown identifier. This is partially due to our method of coalescing multiple definitions: macro definitions categorized as unknown identifier are overridden by definitions with any other categorization (see Figure 7). Our analysis included enough library header files to include some recognizable definition of most common macros.

In 22 of our 26 packages (all but gawk, gnuchess, mosaic, and remind), at least 98% of all macros are defined four or fewer times. Half of all packages have no macros defined more than 12 times, and the overall redefinition behavior of most packages approximates the mean over all packages. Notable exceptions are bc, remind, and gcc. bc is very sparing with multiple definitions: with the exception of some Yacc macros, every macro is defined either one or two times. By contrast, remind defines 10% of its macros more than 10 times (but none more than 15). It supports ten different natural languages (and various character sets) by using macros for all user output strings. The tail of gcc's graph is longest of all: 1.1% of macros are defined more than 5 times, including over 30 definitions of `obstack_chunk_alloc` and `obstack_chunk_free`. (These figures involve only a single configuration; for all of gcc's source code, including various architectures and operating systems but excluding libraries, 4% of macros are defined 20 times and 0.5% are defined 50 times.)

5.5 Multiple differing definitions

Section 5.4 counted the number of definitions of a given macro name, providing an upper bound on the difficulty of mapping uses of the macro to its definition. Multiple definitions are less worrisome if their bodies are lexically similar or identical; this section reports how often that is the case.

Figure 6 provides data regarding multiple definitions of macros, both when each definition is counted

To unify two macro body categories:

- If the categories are the same, use that.
- If one is an unknown identifier (or the name of an undefined macro), use the other on the premise that the unseen definition is likely to be similar to the available one, which is true for well-behaved macros.
- If one is a null define, use the other. (For instance, a type modifier may be present or absent. In order to either perform an action or do nothing, macros not uncommonly expand to either a statement or to nothing — though it would be more robust to expand to a null statement in the latter case. Additionally, a macro used as a boolean variable that is checked for definedness may be set via a null define or by being assigned a constant, generally 1. This practice is an error if the macro is used outside the Cpp `defined` operator, but is also frequent and generally innocuous.)
- If one is a constant and the other is an expression, use the latter.
- If one is an ambiguous list of space-separated identifiers and the other is a reserved word or type, use the latter. Sequences of identifiers are difficult to definitively identify in isolation, but the other definitions of the same name indicate the intended usage.
- If one is an expression or constant and the other is a semicolonless statement, use the latter, for a semicolon can be added to any expression to make a statement. In particular, function calls are categorized as expressions but may be intended to be used for side effect rather than for value.
- If one is a statement or partial statement, and the other is the corresponding plural form (i.e., if the other consists of some number of complete statements followed by a statement or partial statement, respectively), then use the latter.
- Otherwise, there is a conflict; return “other”.

Figure 7: Rules for unifying two macro definition categories. These rules are used when combining categories of multiple definitions of a macro in order to assign a category for the macro name.

individually and when only definitions with differing canonicalized bodies are counted. The canonicalization eliminates all comments and whitespace, canonically renames all formal arguments, and considers all character and string literals to be identical. This transformation approximates comparing abstract syntax trees and is less strict than the rules used by Cpp when determining whether to issue a warning about redefinition.

The number of differing canonical redefinitions is dramatically lower than the number of redefinitions, indicating that multiple definitions are not so troublesome as they initially appear. Syntactic macros are particularly reduced: most of the multiple definitions are one of just a few alternatives. Additionally, most macros in remind canonicalize identically — usually, only string contents differed.

5.6 Inconsistent definitions

This section continues our analysis of multiply-defined macros. Section 5.5 considered the syntactic structure of multiple definitions of a particular name; this section refines that analysis by considering the categorization of the macro bodies described in Section 5.1. A software engineering tool may be able to take advantage of higher-level commonalities among the macro definitions (at the level of the categorizations of Section 5.1) more effectively than if it relied on syntactic similarity, as in Section 5.5.

In 96% of cases, multiple definitions of a macro are compatible (often, the bodies are lexically identical). Incompatibilities usually indicate bugs or inconsistent usage in the package, or failures of our categorization technique.

A macro name is categorized by merging its definitions pairwise. When all definitions of a name fall into the same category or are all consistent with a category, the name is assigned to that category; otherwise, the name is assigned to the “other” category. Figure 7 details the rules precisely.

The category breakdown by macro name (detailed in the legend of Figure 10) differs from the by-definition breakdown of Figure 3 in several ways. The number of null definitions is lower, as null definitions are often found in conjunction with other types of definition and are eliminated by the category merging. (Macros defined only via null definitions are generally used only in Cpp conditionals.) The number of statements is lower, largely because some macros names with statement definitions have other, incompatible definitions, so the macro name is categorized as “other”. The percentage of unknown identifiers is higher because such macros tend to have very few definitions, so are more prominent in a breakdown by name than by definition. The number of “other” categorizations increased because it includes any macro with a definition categorized

Percentage of 3054 multiply-defined macro names	Null define	Constant	Expression	Statement	Type	Syntactic	Identifier	Unknown identifier	Other	Name categorization
32% (982)		■								constant
28% (869)			■							expression
6.0% (183)		■	■							expression
5.8% (176)	■									null define
3.4% (103)				■						statement
2.4% (74)	■			■						statement
2.0% (60)					■					type
1.7% (51)			■	■						statement
1.6% (50)	■		■							expression
1.6% (50)	■				■					type
1.2% (38)	■	■								constant
1.1% (35)						■		■		syntactic
1.1% (35)								■		unknown identifier
1.1% (33)			■					■		expression
1.1% (33)							■			identifier
1.0% (32)									■	other
0.92% (28)			■				■			expression
0.65% (20)	■								■	other
0.65% (20)		■							■	constant
0.52% (16)	■							■		null define
0.52% (16)			■						■	other
0.46% (14)							■	■		identifier
0.43% (13)					■				■	other
0.39% (12)			■	■						other
0.29% (9)		■						■		constant
0.26% (8)						■				syntactic
0.23% (7)				■						other

Figure 8: Categorization of definitions for each macro name with more than one definition. For instance, for 869 macro names (28% of multiply-defined macro names), all definitions fall into the expression category, and for 183 macro names (6.0% of multiply-defined macro names), all definitions are either expressions or constants. Rows less than 0.2%, representing 5 or fewer macro names, are omitted. The rightmost column indicates the categorization of the macro name; 4.4% of all multiply-defined macro names, and 2.0% macro names overall, are are categorized as "other". This chart shows which different macro definition categories tend to occur together and assists in understanding the reasons for macro names whose multiple definitions cause them to be categorized as "other".

as "other" as well as any with incompatible definitions.

Figure 8 gives more detailed information for the 14% of macro names that have multiple definitions. Macros are grouped by whether all of their definitions fall into the same set of categories.

The arrows in the chart indicate which macro names have "other" categorizations. Using 9 presentation categories rather than the 28 categories distinguished by our tool makes this table manageable, but does hide some information. For instance, there are two "expression + statement" groups, one at 1.7% and one at 0.39%. The more prevalent one includes expressions and semicolonless statements, categories that are compatible with one another; those macro names are categorized as semicolonless statements. The second group includes definitions that are expressions, semicolonless statements, and complete statements; those twelve macro names are categorized as "other" and, with one exception, represent bugs in the packages. One such bug in zephyr is

```
#define adjust_size(size) size -= 0x140000000
#define adjust_size(size) size -= (unsigned int) &etext;
```

Likewise, the "statement" row at 0.23% is marked as a conflict because it includes both semicolonless and full statements, which are not interchangeable. All seven of these are bugs; an example in gcc is

```
#define TARGET_VERSION fprintf (stderr, " (i860, BSD)")
#define TARGET_VERSION fprintf (stderr, " (i860 OSF/1AD)");
```

6 Macro usage

The previous section demonstrated ways in which macro definitions complicate program understanding; now we turn to macro uses. First, heavy macro usage makes macro analysis more important by amplifying the effect of each macro; Section 6.1 addressees this issue. Macro usage in the packages we analyzed varies from perl's 0.60 macros per line of code to workman's 0.034. While 50% of macro names are used two or fewer times, 1.7% of macros (364 macros) are used at least 100 times. Macros categorized as syntactic and type-related are expanded 10 times more frequently than simpler macros defining constants or expressions.

Second, consistency of use can resolve some of the ambiguities inherent in macro definitions, while inconsistent use has the opposite effect. A macro used in a limited way can be replaced — in a tool's analysis or in the source — by a simpler mechanism. Section 6.2 demonstrates that this approach shows promise: less than 3% of macros are both tested for definedness and expanded in source code.

Finally, which macros appear in a conditional compilation test can reveal the programmer's intention underlying that test. For tests not specific to a particular package, the separation of concerns is generally good: Section 6.3 shows that only 4% of Cpp conditionals test multiple macros with different purposes.

6.1 Frequency of macro usage

Figure 9 illustrates how frequently each package uses macros. Macros pervade the code, with 0.28 uses per line, though individual packages vary from 0.034 to 0.60 uses per line. Heavy preprocessor use (high incidence of preprocessor directives, as reported in Figure 2) is only weakly correlated with heavy macro usage, even though many preprocessor directives use macros. The language implementations in our study (perl, gcc, and gs) use macros the most.

Figure 10 illustrates that 50% of macros are expanded no more than twice and 10% are never used at all. Many of these unused macros appear in incomplete or obsolete code. For example, gnuplot, which does not use 22% of the macros it defines, includes several partially implemented terminal types, such as `tgif`.

The most frequently used macros are those most likely to cause difficulty for a tool or software engineer: 39% of syntactic macros (those expanding to punctuation or containing unbalanced delimiters, and that are difficult to parse) and 12% of type-related macros are used more than 32 times. The long tails of the frequency distribution result from pervasive use of some syntactic or type macros (e.g., at every variable declaration), which makes understanding these macros critical. By contrast, macros that act like C variables by expanding to a constant or expression generally appear only a few times — 58% of macros defining constants occur no more than twice.

6.2 Macro usage contexts

Macros have two general purposes: they can control the inclusion of lines of code (by appearing in a Cpp conditional that controls that line) or can change the text of a line (by being expanded on that line). Each of these uses may correspond to language features — conditional statements and expressions (`if` and `?:`) or `const` and `inline` declarations (for certain types of substitution). Understanding is inhibited when a macro is used in both ways, for there is no easy mapping to an existing language feature.

We split macro uses into three contexts:

- uses in C code. The macro's expansion involves textual replacement.
- uses in `#if`, `#ifdef`, `#ifndef`, and `#elif` conditions. In this section, we disregard uses in Cpp conditionals whose only purpose is to prevent redefinition. More specifically, we ignore uses in a condition that tests only a macro's definedness and whose body only defines that macro.

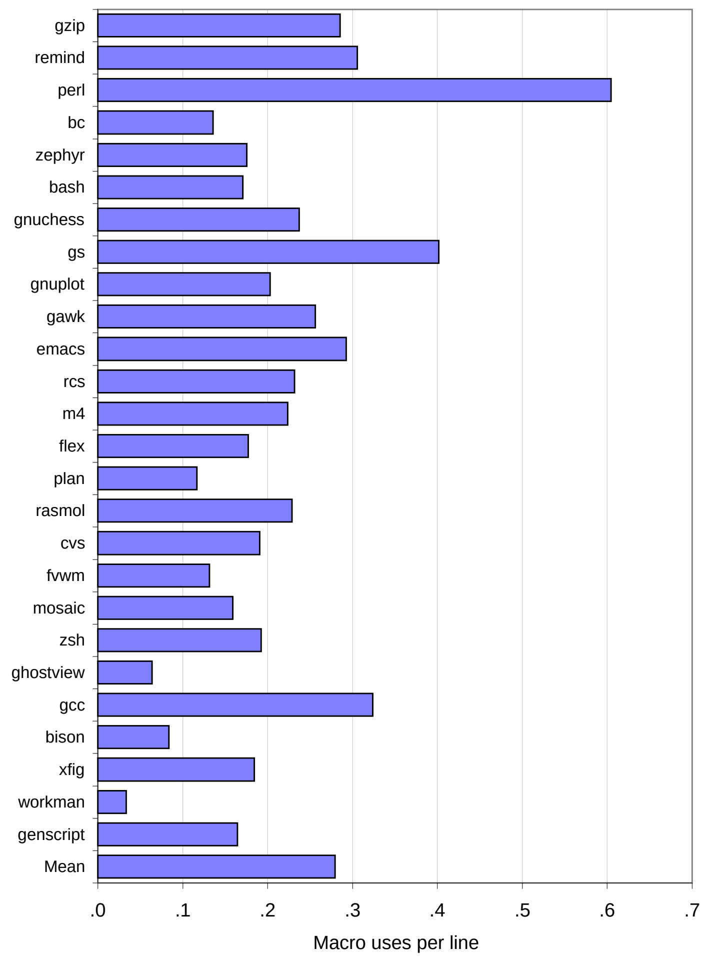

Figure 9: Number of macro uses divided by number of NCNB lines. The packages are ordered from most preprocessor directives per line to fewest (as in Figure 2).

- uses in the body of a macro definition. Macros used in such contexts eventually control either textual replacement or code inclusion (according to uses of the macro being defined). Overall, 18% of macros appear in macro bodies, and uses in macro bodies account for 6.0% of macro uses.

Figure 11 reports in which of these three contexts macro names are used. In general, packages use macros either to direct conditional compilation or to produce code, but not for both purposes. Only 2.4% of macros (the fourth group of Figure 11) both expand in code and are used in conditional contexts. Macros are expanded an order of magnitude more often than they control source code inclusion (75.9% in the first group vs. 6.5% in the second). Conditional compilation accounts for 48% of Cpp directives but only 6.5% of macro usage. However, each use in a conditional directive can control many lines of code, whereas each use in code affects the final program text for just that line; see Section 7.

6.3 Macro usage in conditional control

Cpp conditionals are used to control inclusion of code for portability, debugging, efficiency, and other purposes. The programmer intention behind a `#if` line can often be inferred from its structure, its context, or the purpose of the macros it uses.

Figure 12 shows the heuristically determined purpose of each Cpp conditional in our test suite. First, the heuristic classified some conditionals according to their structure or context, as follows:

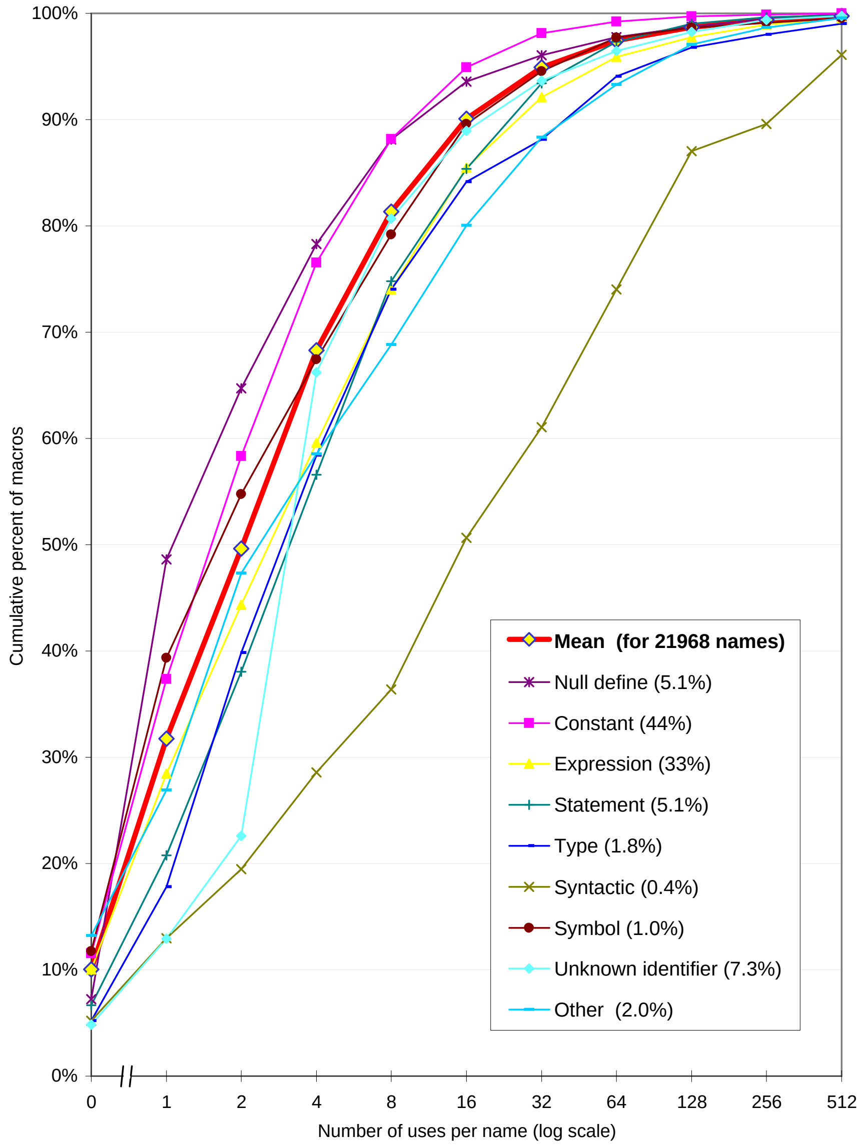

Figure 10: Number of expansions per Cpp macro. The numbers in the graph represent the percentage of identifiers that are expanded a given number of times or fewer. For example, 50% of all macros are expanded two or fewer times. In this chart, higher lines indicate less usage: syntactic macros are used the most, null defines and constants the least. Percentages in the legend represent the total number of macro names falling in each category; Figure 3 gives similar information broken down by macro definition.

Commenting These guards either definitely succeed and have no effect as written (e.g., `#if 1`), or definitely fail and unconditionally skip a block (e.g., `#if (0 && OTHER_TEST)`). These guards comment out code or override other conditions (e.g., to unconditionally enable or disable a previously experimental feature).

Redefinition suppression These guards test non-definedness of identifiers, and control only a definition of the same identifier, thus avoiding preprocessor warnings about a redefinition of a name (e.g., `#ifndef FOO` followed by `#define FOO ...` and `#endif`). The purpose is to provide a default value used unless another part of the system, or the compilation command, specifies another value.

For Cpp conditionals not classified by the above rules, the purpose of each macro name appearing in the conditional is determined from the system properties it reifies. If each macro in the conditional has the

Code	63.3%
Code, macro	12.6%
Cond.	6.2%
Cond., macro	0.3%
Macro	4.9%
Code, cond.	1.7%
Code, cond., macro	0.7%
No uses	10.3%

Figure 11: Macro usage contexts. Macros may be used in C code, in macro definition bodies, in conditional tests, or in some combination thereof. The 10.3% of "No uses" is the same number as the 0 uses value of the Mean line in Figure 10. This figure groups (for example) macros used in code only with macros used in both code and macro bodies, on the assumption that uses in macro bodies are similar to other uses of the macro.

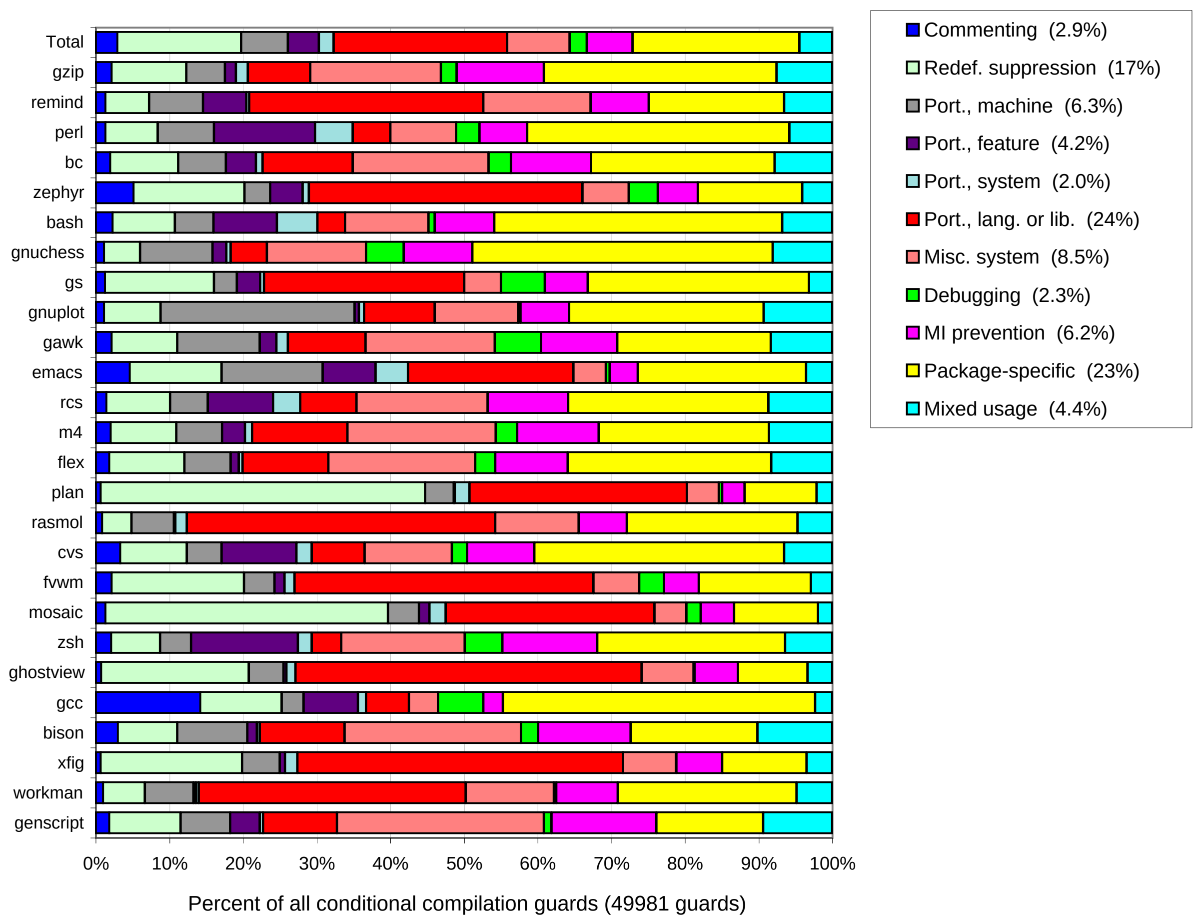

Figure 12: Purposes for conditional compilation directives. The legend indicates what percentage of all Cpp conditionals fall into each category, numerically presenting the information in the top row of the chart.

same purpose, then the conditional is given that purpose; otherwise, the conditional is classified as "mixed usage". The macro purposes, which are determined from the macro's name rather than an examination of its definitions (which are often either unavailable or trivial, such as the constant `1`), are as follows:

Portability, machine These symbols name the operating system or machine hardware (e.g., `sun386` or `MACINTOSH`).

Portability, feature These symbols describe specific parameters or capabilities of the target machine or operating system (e.g., `BYTEORDER`, `BROKEN_TIOCGWINSZ`).

Portability, system These symbols are commonly defined constants or pseudo-inline functions in system or language libraries (e.g., `O_CREATE`, `isalnum`, `S_IRWXUSR`).

Portability, language or library These symbols are predefined by a compiler, defined by a standard library, or defined by the package as part of the build process to indicate existence of compiler, language, or library features (e.g., `GNUC`, `STDC`, `HAS_BOOL`).

Miscellaneous system These symbols are reserved (they begin with two underscores) and do not fit any other purpose.

Debugging These symbols control inclusion of debugging or tracing code. The macro names contain `DEBUG` or `TRACE` as substrings.

Multiple inclusion prevention These guards encompass an entire file to ensure that the enclosed code is seen only once per translation unit by the compiler. Such guards are indicated by convention with a trailing `_H` or `_INCLUDED` in the macro name they check.

Package-specific These symbols are specific to the given package. They do not fit any of the other purposes.

Mixed usage These guards test multiple symbols that have different purposes (e.g., `#if defined(STDIO_H) || SYSV_SIGNALS`).

There is significant variation among packages, and no clear pattern of use emerges. Portability accounts for 37% of conditional compilation directives. Redefinition warning suppression, at 17%, is surprisingly high; it is essentially a macro definition mechanism, not a conditional inclusion technique. Mixed usage is relatively rare. This suggests both that the conventions for macro names are fairly standardized and that programmers rarely write conditional tests that combine entirely different concerns in a single expression.

These data suggest that 23% of conditional compilation directives would be unnecessary if the C preprocessor had two simple language features: a "define only if not already defined" directive and an Objective-C-like `#import` facility which automatically avoids multiple inclusions. Another 37% of conditional compilation directives involve variances in target operating systems. Tools such as the GNU project's `autoconf` may account for the prevalence of these guards by making it easier to maintain code bases with sprinkled `#ifdef`s managing multiple target operating system variants. It would be interesting to compare these data to those collected for software that targets only a single specific platform.

7 Dependences

Macros control the program that results from running Cpp via inclusion dependences and expansion dependences. This section reports the incidence of these dependences, both by macro and by line.

Inclusion dependence results from Cpp conditionals that test macros to determine which lines of the Cpp input appear in the output. A line is inclusion-dependent on a macro name if and only if the macro's definedness or its expansion can affect whether the line appears in the preprocessor output. In other words, there is a set of values for all other macros such that the line appears in the output for one value of the macro (or for the case of the macro being undefined) and does not appear in the output for the other value of the macro (or for the case of the macro being undefined). This notion is related to control dependence in program analysis.

Expansion dependence results from replacement of macros outside Cpp conditionals by their definition bodies, which controls the content of the lines on which the macros appear. A line is expansion-dependent on a macro name if the macro's definedness or value affects the text of the line after preprocessing. In other words, for some set of values for all other macros, setting the macro to one value (or undefining it) results in a different final text for the line than setting the macro to a different value (or undefining it). This notion is related to data dependence in program analysis.

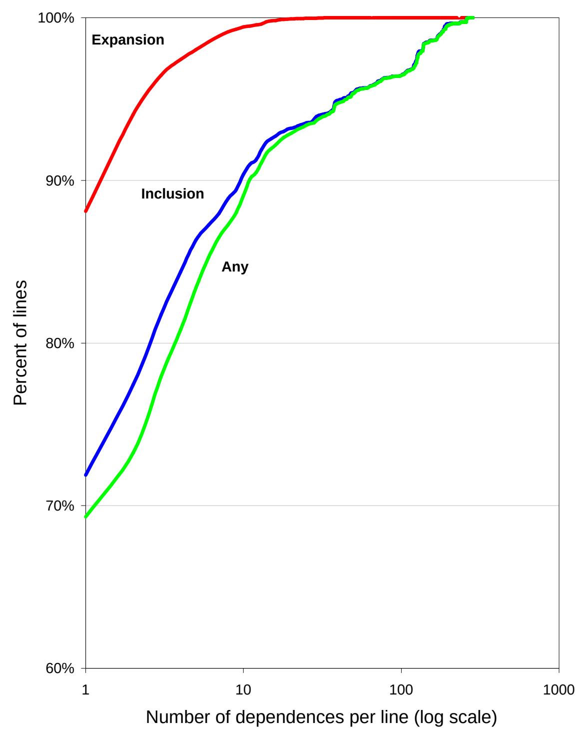

Figure 13: Percentage of lines dependent on a particular number of macros (or fewer). For instance, 94% of all lines are expansion-dependent on 2 or fewer macros, and 90% of all lines are inclusion-dependent on 10 or fewer macros. Lines higher in the graph indicate dependence on fewer macros. The values for 0 macros (which does not fall on the log-scale x axis) are as follows: 75% of lines expand no macros, 63% of lines are unconditionally included (or excluded), and 60% of lines are not dependent on any macros at all.

We report both direct and indirect dependences. A line directly depends upon macros that appear in the line or in a `#if` condition whose scope contains the line. It indirectly depends on macros that control the definitions of directly controlling macros. After `#define S_ISBLK(m) ((m) & S_IFBLK)`, the final text of a line that uses `S_ISBLK` depends not just on its definition but also on that of `S_IFBLK`. An indirect dependence is an expansion dependence if every dependence in the chain is an expansion dependence; otherwise, the indirect dependence is an inclusion dependence.

We distinguish *must* from *may* dependences. A must dependence links a use to the macro's known single definition site; a may dependence links a use to multiple definition sites, when it is not known which definition is in effect at the point of use. When a macro is defined on both branches of a `#if` conditional, the macro's definedness does not depend on the values tested in the conditional, though its value might. We do track dependences across file boundaries: if a macro controls whether a file is `#include`d, then the macro also controls every line of that file.

The statistics reported in this section are underestimates because they omit emacs, which aggressively uses macros. Its full dependence information exceeded 512 MB of virtual memory, in part due to its optional use of Motif, a complex external library with extensive header files. (While this paper reports only on macros defined in each package, we computed dependences and other information for all macros, including those defined or used in libraries. Additionally, our implementation is not optimized for space.) We did generate dependence information for mosaic and plan, which also use Motif.

7.1 Dependences by line

Figure 13 graphs the percentage of lines dependent on a given number of macros. On average, each line in the 25 packages we did dependency analysis on is expansion-dependent on 0.59 macros, inclusion-dependent

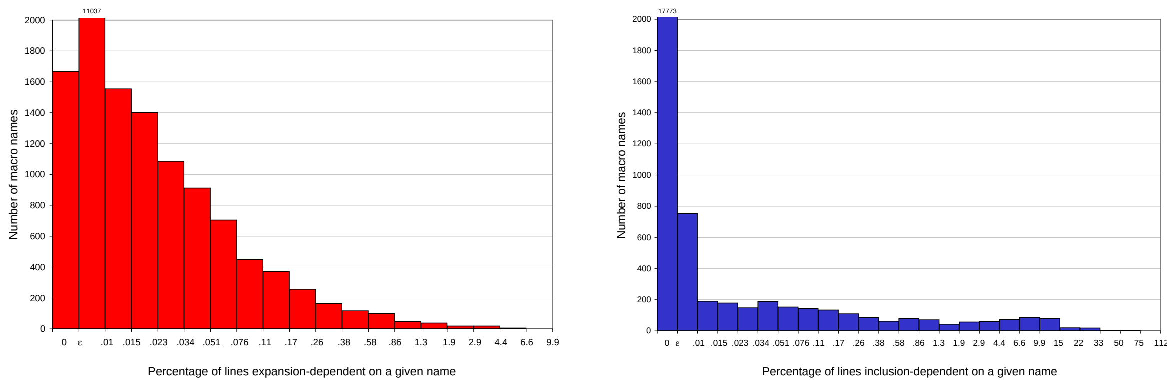

Figure 14: Dependences by macro name for 19945 macro names in 25 packages. The first bar represents all macros that are completely unused in a package. Each other bar represents all macros that control a specified percentage range of the lines in a package. For instance, the 0.17–0.26 bar in the expansion dependence chart indicates that 257 macros each control between 0.17% and 0.26% of the entire package that contains that macro. The maximum falls in the penultimate bucket (i.e., the rightmost bucket is the first empty one). The second bar of each graph represents all macros controlling at least one line (ε is a very small but nonzero value), but no more than .01for the x axis.

on 8.2 macros, and has some dependence on 8.7 macros. Some lines that are inclusion-controlled by macros do appear unconditionally in the package source but are inside a guard to avoid multiple inclusion — this is the case for many header files.

Expansion dependence on multiple macros is not prevalent — only 3.6% of lines are expansion-dependent on more than 3 macros, and only 1.1% are expansion-dependent on more than 7 macros. However, one line of `gcc` — `LEGITIMIZE_ADDRESS (x, oldx, mode, win);` — is expansion-dependent on 41 different macros. (When we examined all source code, not just one architecture/operating system configuration, it was expansion-dependent on 187 macros!) Macro `LEGITIMIZE_ADDRESS`, which creates a valid memory address for a memory operand of a given mode, is defined 30 times in `gcc`, with many of the definitions dozens of lines long and themselves studded with macro invocations. Inclusion dependences have a much wider distribution. Overall, 10% of lines are inclusion-dependent on at least 10 macros, and 1% of lines are dependent on over 176 macros.

7.2 Dependences by macro

Figure 14 graphs how many lines are dependent on each macro (Figure 13 gave the same information by line rather than by macro). Since the 25 packages vary in size, the graphs of Figure 14 aggregate them by reporting percentages of a package rather than absolute numbers.

The expansion dependence chart illustrates that most macros control few lines, a few macros control many lines, and the transition between the two varieties is gradual. (The values follow a variant of Zipf's law [Zip49]: the product of the number of macros and the percentage of lines dependent on those macros is nearly constant. Excluding the first four and last two buckets, which represent extremal values, the product's average is 40, with a standard deviation of 8.7.) Most `#if` directives (which account for 2.0% of all lines) expand at least one macro; the rare exceptions include testing the compiling machine's character set.

Each of the 20 most frequently used macros is expanded on at least 3.0% of lines in its package. Four of these (in xfig and perl) rename variables to manage dynamic binding or linking. Two are user-defined types (`rtx` in gcc and `SvANY` in perl), two are user-defined type modifiers (`private_` in gs and `local` in gzip), and one creates either ANSI or K&R function prototypes (`P` in rcs). One is a user-defined constant (`AFM_ENC_NONE` in genscript), another is an expression (`SvFLAGS` in perl), and another is incompatibly defined sometimes as a constant taking no arguments and sometimes as an expression taking arguments (`ISFUNC` in bash). The other ten redefine built-in quantities: `fprintf` in gnuplot to substitute a custom version, `__far` in rasmol because that is meaningful only in x86 dialects of C, and `void` in perl, `const` in gs and rcs, and `NULL` in cvs, fvwm, gawk, m4, and mosaic. Because we determined which symbols are macros by running a modified version of the preprocessor, we report such redefinitions only when the package may override the built-in version. Generally, only a few such built-in symbols are overridden.

The inclusion dependence graph is bimodal. While most macros control inclusion of zero or few lines, quite a few control over 5% of the package, and there are not a lot of macros in between. The graphs for the individual packages exhibit far higher peaks than the aggregate inclusion dependence graph of Figure 14; summing the graphs tended to average them. The heaviest dependences tend to be on macros controlling header file inclusion.

7.3 Cpp partial evaluation

Support for multiple dialects of a language, such as ANSI C and K&R C, is a common use of the preprocessor: 3 of the 26 packages support only ANSI C, 4 support only K&R C, 13 use the preprocessor to fully support both dialects, and 6 prefer one dialect but partially support another (for instance, part of the package supports both dialects or a substantial number of functions have both varieties of prototype). Such support is possible only in the preprocessor, not in the language, and leads to unstructured macros (partial declarations and other difficult-to-handle constructs). Furthermore, these uses are highly visible and distracting to programmers and tools, because they may change the syntax of every function declaration and definition.

We performed an experiment to determine whether eliminating these macros would simplify understanding or analysis by reducing the complexity of macro definitions and usage as measured in this article. We built a Cpp partial evaluator called Cppp. Given Cpp-style command-line arguments specifying which macros are known to be defined or undefined (and, optionally, their expansions), Cppp discharges Cpp conditionals, including nested conditionals, that depend on those macros. Other conditionals, and macro uses, remain in the output. Cppp does not expand macros inline or use macro definitions found in its input files. Cppp is similar to the `unifdef` program distributed with some Unix variants, except that `unifdef` does not permit specifying a value for a defined symbol and only operates on `#ifdef` tests.

We used Cppp to preprocess all of our test suite (and all library header files) with definitions for all the macros that can be depended on if using ANSI standard C or C++ (including prototypes and booleans) and POSIX-compliant libraries. This reduced the size of the codebase by 1.2% overall; individual package reductions ranged from 3.2% for emacs to none at all for workman. We then reran all of our experiments, but to our surprise, the results were little changed from the full versions of the packages. The decline in multiple definitions of macros, "other" categorizations, dependences on macros, and other metrics we report were single-digit percentages of their original values. We conclude that extra-linguistic macro usage in our test programs presents no obvious single point of attack: even eliminating one highly prevalent and visible use — which was also the largest identifiable source of Cpp conditionals (see Figure 12) — did not significantly reduce the complexity introduced by preprocessor.

Discharging selected conditionals can sometimes be worthwhile, however. The developers of the Scwm window manager [SB99] used our Cppp tool to eliminate numerous compile-time options inherited from Scwm's predecessor, fvwm2. This transformation resulted in a cleaner code base that the developers found easier to understand and modify.

8 Related work

We know of no other empirical study of the use of the C preprocessor nor any other macro processor. However, the literature does contain guidance on using C macros effectively and on writing portable code, tools for checking macro usage, and techniques for understanding and exploring C source code that uses the preprocessor.

8.1 Style guidelines

Several coding style guides make recommendations on preprocessor use and on ways to reduce unexpected behavior resulting from poorly designed constructs [CEK+90]. Our empirical data help refine these sets of suggestions, both by extending their taxonomies and recommendations and by indicating which problems occur in practice.

The GNU C preprocessor manual [GNU] discusses a set of techniques including simple macros, argument macros, predefined macros, stringization macros, concatenation macros, and undefining and redefining macros. Its list of "pitfalls and subtleties of macros" include unmatched parentheses, unparenthesized formals and bodies, dangling semicolons, multiple formal uses, and self-referential macros. It also discusses three issues not addressed in this paper: argument prescan (macro arguments are actually scanned for macro expansions twice), cascaded macros (macro bodies use the current definition of other macro, not the ones in effect at the definition site), and newlines in macro invocations (which can throw off error line reporting).

The GNU coding standards [Sta97] mention Cpp only to recommend upper case for macros and use of macros to provide meanings for standard symbols on platforms that don't support ANSI C. In any event, most GNU programs do not appear to have been written with an eye to careful adherence to the standards. The Ellemtel coding standards [Ell92] recommend that inline definitions appear in separate files of their own (not in `.h` header files) that never include other files, that inline functions and `const` or `enum` constants be used in place of `#define`, and that preprocessor macro names be prefixed to avoid name clashes.

An adaptation of the Indian Hill C Style and Coding Standards [CEK+90] makes numerous suggestions. Programmers should use all capital letters for macro names except for macros that behave like function calls, which are acceptable only for short functions. Statement macros should be wrapped in `do { ... } while (0)`, particularly when they contain keywords. Conditional bodies should be fully bracketed to avoid dangling semicolons. Macros should evaluate their parameters exactly once and arguments should not have side effects. Side effects to macro parameters should be documented. Macros should avoid using global variables, since the global name may be hidden by a local declaration. Macros with bodies that are long or that reference variables should take an empty parameter list. Programmers should use the ANSI specification rather than preprocessor tricks such as using `/**/` for token pasting and macros that rely on argument string expansion. Syntax should not be changed by macro substitution (except for the `PROTO` macro). While `#ifdef` is to be avoided, it should appear in header files (which should not be nested) rather than source code. It can be used to protect `#pragma` and to define macros that are used uniformly in the code, without need for further `#ifdef`. If a machine is not specified, compilation should fail rather than using a default; but conditional compilation should generally test features, not machines or operating systems. The text inside an `#ifdef`-bracketed section should be parsable code, not arbitrary text.

Stroustrup [Str94] lists 14 tasks supported by the C preprocessor and notes C++ alternatives for 5 of the 8 uses of `#define`: constants, inline subroutines, parameterized types and functions, and renaming. While a principle of C++'s design was elimination of the preprocessor, C++ continues to rely on it for the other uses Stroustrup lists: `#define` for string concatenation, new syntax, and macro processing, `#ifdef` for version control and commenting, `#pragma` for layout and control flow hints (though pragmas are disparaged), and `#include` for exporting declarations and composing source text. Stroustrup proposes moving `#include` into the language, which could eliminate some of its quirks and simplify the task of programmers and tool writers, as the `#import` directive does for Objective-C [CN91]. Stroustrup remarks that "In retrospect, perhaps the worst aspect of Cpp is that it has stifled the development of programming environments for C."

Carroll and Ellis [CE95] list eight varieties of non-`#include` macro usage, culled in part from Stroustrup's list [Str94]. They say that C++ can replace these uses of the preprocessor other than declaration macros (such as declaring a class constructor and destructor via a macro call) and code versioning (such as debugging versions). They also recommend the use of corporation- and project-specific prefixes to avoid macro name and header file name conflicts.

8.2 Portability

Two other style guidelines focus on portability concerns. Dolenc et al. [DKR90] warn of implementation limits permitted by the C language standard such as 32 nesting levels of parenthesized expressions, 1024 macro identifiers, and 509 characters in a logical source line. They also note incompatibilities among preprocessors that usually result from failure of implementations to conform to the specification. For instance, some nonstandard preprocessors do not support the `defined` operator or the `#pragma` or `#elif` directives, ignore text after the `#else`, `#elif`, and `#endif` directives, or perform concatenation and stringization in nonstandard orders during macro substitution. The authors recommend using some of these incompatibilities — such as accepting only directives starting in the first column, without leading whitespace — along with `#ifdef` to hide modern Cpp features from older preprocessors. The paper also treats in detail specific header files that may cause portability problems, showing how to overcome these difficulties, generally by using the macro preprocessor to define some symbols (more) correctly.

Spencer and Collyer [SC92] provide a set of techniques for achieving portability without using `#ifdef`, which they recommend only for providing default values for macros and preventing multiple inclusion of header files. The paper is as much about good software engineering as it is about the preprocessor per se, but does contain some preprocessor-specific recommendations. The authors suggest using standard interfaces, then providing multiple implementations if necessary. These implementations should appear in separate files rather than sharing code via `#ifdef`, and the build or configure script should select among them; thus, a choice of files replaces Cpp conditionals. An override directory early in the search path permits bad include files to be replaced rather than selectively overridden. More uses of `#ifdef` can be eliminated by moving system-specific tests and operations into shell scripts or by using standard programs (such as ls or df) instead of accessing system services from C code. (These strategies can complicate porting to non-Unix systems, and even standard programs may have different behavior in different places.) Use of the preprocessor to establish numeric constants should be viewed with suspicion; dynamically-sized objects are a better approach. Uses of `#ifdef` should test for features or characteristics, not machines, and an error is desirable to selecting a default machine. Spencer and Collier recommend that `#ifdef` be restricted to declarations and macro definitions, never used at call sites, and that `#include` never appear inside `#ifdef`. They also break down the uses of `#ifdef` in the 20,717 lines of their C News program. Of the 166 uses, 36% protected a default value for a preprocessor symbol, 34% were used for configuration, 15% commented out code, and 3% prevented multiple inclusion of header files.

8.3 Error checking

Krone and Snelting use mathematical concept analysis to determine the conditional compilation structure of code [KS94]. They determine the preprocessor macros each line depends upon (in our terminology, they only compute inclusion dependence, not expansion dependence) and display that information in a concept lattice. They do not determine macro relationships directly, but only by their nesting in `#if`, and the information conveyed is about the program as a whole. Each point in the lattice stands for a set of lines dependent on exactly the same preprocessor symbols, though not necessarily in exactly the same way. The lattice can reveal that one macro is only tested within another one's influence, for example. When the lattice does not have a regular grid-like structure, it is possible that the code does not properly separate concerns. The most closely related part of our paper is Section 6.3, which analyzed single compilation directives that tested multiple incompatible macros using a fixed set of macro purposes.

A number of tools check whether specific C programs satisfy particular constraints. Various lint [Joh77] source-code analyzers check for potentially problematic uses of C, often including the C preprocessor. Macro errors are usually discovered as a byproduct of macro expansion — for instance, by generating an empty statement that causes lint to issue a warning — rather than in their own right. A survey of nine C++ static checkers [MK97] mentions the macro preprocessor only in terms of whether such warnings can be turned off in the tools; however, that paper focuses on coding style likely to lead to errors rather than on lexical issues.

LCLint [EGHT94, Eva96] allows the programmer to add annotations that enable more sophisticated checks than many other lint programs. LCLint optionally checks function-like macros — that is, those that take arguments — for macro arguments on the left hand side of assignments, for statements playing the role of expressions, and for consistent return types. LCLint's approach is prescriptive: programmers are encouraged not to use constructs that might be dangerous, or to change code that contains such constructs. For full checking, LCLint also requires users to add fine-grained annotations to macro definitions. We tried to run LCLint version 2.3i on our set of packages with its macro diagnostics enabled, but LCLint reported either a parse error or an internal bug on 92% of the files in our test suite.

PC-Lint/FlexeLint [Sof99] is a commercial program checker. Among the macro errors and problematic uses identified by our tool, PC-Lint/FlexeLint warns about unparenthesized bodies, multiple macro definitions (classifying them as either "identical" or "inconsistent"), unreferenced defined macros, and macros that could probably be replaced by a const variable. It also warns of illegal arguments to the `#include` directive, header files none of whose declarations are used in a module, and names defined as both C variables and macros. At macro definitions, it warns of multiple formals with the same name, use of `defined` as a macro name, and formal names that appear in strings (because some non-compliant preprocessors perform substitution even in strings). At call sites, it warns of incorrect number of arguments, unparenthesized macro arguments ("when the actual macro argument sufficiently resembles an expression and the expression involves binary operators," but ignoring operator precedence), and arguments with side effects (when the formal parameter is used multiple times in the expression body). Its warnings can be individually suppressed to accommodate intentional uses of these paradigms.

Check [SS92] is a C macro checker that detects some instances of multiple statements, swallowed `#else`, side effects, and precedence errors using largely lexical checks. Precedence error checks are performed on macro uses rather than reporting error-prone definitions. The authors do not report on the effectiveness of the tool in practice, nor do they justify their tradeoffs between techniques that perform parsing and those that do not, although such tradeoffs are a major theme of their paper.

We found no mention in the literature of software complexity metrics that specifically address the macro preprocessor (except for a study of program comprehensibility in the dBaseIII and Lotus 1-2-3 macro languages [DDL90]). As anecdotal confirmation of this neglect of the preprocessor, we examined 12 publicly-available systems for computing metrics over C programs [Lot98]. metre requires preprocessed source, spr and metrics filter out Cpp directives, c_lines, cyclo and lc ignore Cpp, metre requires preprocessed source, csize performs preprocessing itself and counts Cpp directives, and c_count, clc, and hp_mas count preprocessor directives; c_count counts statements but ignores those in expanded macros. cccc permits users to specify specific macros to ignore; it skips preprocessor lines and assumes macros contain no unbalanced braces. c-count (a different program than c_count, mentioned above) operates on unpreprocessed source, but its parser is stymied by Cpp comments and by macros not expanding to statements, expressions or certain types, so users may specify the expansions of certain macros (so that only one side of Cpp conditionals is examined) and which macros to ignore.

8.4 Understanding Cpp

A limited number of tools assist software engineers to understand code with containing Cpp directives. Emacs's `hide-ifdef-mode` [Sta94] enables the programmer to specify preprocessor variables as explicitly defined or not defined; the mode then presents a view of the source code corresponding to that configuration, hiding code that is conditionally unincluded, much like the Cppp and unifdef tools. A similar tool is the VE editor, which supports both selective viewing of source code and automatic insertion of Cpp conditionals around edited code. Based on evidence from version control logs, this tool was found to be effective in making programmers more productive [ABGM99]. Various language construct "tagging" mechanisms (e.g., `etags` and `ctags`) recognize macro definitions and permit tag-aware editors to move easily from a macro expansion to the various definitions of that macro name. One of the sample analyses of the PCp^3 analysis framework provides an Emacs-based tool for editing the unprocessed source while dynamically updating the current macro's expansion in a separate window [BN98].

Favre suggests that Cpp be expressed in an abstract syntax similar to that of other programming languages [Fav96]. After a simple semantics (free of loops and other complicating constructs) is assigned to Cpp, traditional analyses such as call graph construction, control and data flow analysis, slicing, and specialization can be performed on it. The Champollion/APP environment implements this approach but does not support "the full lexical conventions of the C language" nor macros that take parameters, which make up 30% of macros in our study. The Ghinsu slicing tool [LS94] takes a similar approach, mapping Cpp constructs — particularly macro substitution — to an internal representation that supports slicing.

TAWK [GAM96], which permits searches over a program's abstract syntax tree, handles some uses of Cpp macros. Macros whose bodies are expressions or statements are left unexpanded and entered into the symbol table (Scruple [PP94] takes a similar approach); otherwise, the body is reparsed assuming that each macro argument (in turn) is a typedef; otherwise, the macro is expanded inline, which the authors find necessary for 4% of non-header-file uses.

9 Making C programs easier to understand

The combination of C and Cpp often makes a source text unnecessarily difficult to understand. A good first step is to eliminate Cpp uses where an equivalent C or C++ construct exists, then to apply tools to explicate the remaining uses. Here we discuss a few approaches to reducing the need for the preprocessor by better changing the state of the art in C programming, rather than applying tools to a specific source code artifact. We do not seriously consider simply eliminating the preprocessor, for it provides convenience and functionality not present in the base language.

Since many of the most problematic uses of Cpp provide portability across different language dialects or different operating environments, standardization can obviate many such uses, either in legacy code or, more easily, in new code. Canonicalizing library function names and calling conventions makes conditional compilation less necessary (37% of conditionals test operating system variances) and incidentally makes all programs more portable, even those that have not gone to special effort to achieve portability. This proposal moves the responsibility for portability (really, conformance to a specification) from the application program into the library or operating system.

Likewise, one of the most common uses of Cpp macros could be eliminated if the C language and its dialects had only a single declaration syntax. Declarations are particularly important to tools and humans, who examine them more often than they examine implementations, and declaration macros are particularly cluttering. Because most C compilers, and all C++ compilers, accept ANSI-style declarations, support for multiple declaration styles may have outlived its usefulness. The ansi2knr tool [Deu90] translates a C program using ANSI style function declarations into one using classical function declarations. This tool frees authors from maintaining two commonly required configurations.

Some Cpp directives can be moved into the language proper or be replaced by more specialized directives. For instance, Objective-C [CN91] uses `#import` in place of `#include`. Stroustrup [Str94] also proposes putting `include` in the C++ language; either approach eliminates the need for the 6.2% of Cpp conditionals (and the related definitions) that prevent multiple inclusion of header files. A `#default` or `#ifndefdef` directive would obviate another 17% of conditionals. Compilers that do a good job of constant-folding (42% of macros are defined as constants) and dead code elimination (eliminating many uses of `#if`, for debugging and other purposes) can encourage programmers to use language constructs rather than relying on the guarantees of an extra-linguistic tool like Cpp. It is not sufficient for the compiler to perform appropriate optimizations — the programmer must also have confidence that the compiler will apply those optimizations; skeptical programmers will instead use Cpp to hide computations from the compiler to guarantee code is not generated.

Common Cpp constructs could be replaced by a special-purpose syntax. For instance, declarations or partial declarations could be made explicit objects, such as by making type modifiers first-class at compile (or, more likely, preprocess) time; the language or the preprocessor could include a `dynamic` declaration modifier for dynamic binding (like the `special` declaration for dynamic variables in Lisp [Ste90]); and similar support could be provided for repetitive constructs. Manipulations of these objects would then be performed through a clearly-specified interface rather than via string and token concatenation, easing the understanding burden on the programmer or tool. Such uses would also be visible to the compiler and to program checkers such as lint. The downside of this approach is the introduction of a new syntax or new library functions that may not simplify the program text and that cannot cover every possible case.

The Safer_C language [Sal96] uses partial evaluation as a replacement for preprocessing and for source-code templates, in order to support portability and reuse without run-time cost. Evaluation of expressions occurs at compile time or at run time, controlled by programmer annotations. Maintainability is enhanced due to the elimination of a separate preprocessing step and the language's simpler syntax. Since Safer_C does not support all the features of the preprocessor, the author recommends recoding in a totally different style for uses of the preprocessor that extend beyond definition of preprocessor constants, inclusion of header files, and straightforward conditional compilation. The paper remarks that `gcc` version 2.5.8 contains only "several instances" of such preprocessor use, but our results contradict that finding.

Some macro systems have been designed that avoid particular pitfalls of Cpp. A hygienic macro system [KFFD86] never unintentionally captures variables, though it can do so via a special construct. Other macro systems require that their output be a syntactic AST rather than merely a token stream [WC93].

An alternative approach that avoids the clumsiness of a separate language of limited expressiveness is to make the macro language more powerful — perhaps even using the language itself via constructs evaluated at compile time rather than run time. (The macro systems of Common Lisp [Ste90] and Scheme [KCR98], and their descendants [WC93], take this approach.) In the limit, a language can provide a full-fledged reflection capability [KdRB91]. Such an approach is highly general, powerful, and theoretically clean. The added generality, however, degrades a tool's ability to reason about the source code. In practice, such systems are used in fairly restricted ways, perhaps because other uses would be too complicated. A dialogue among users, compiler writers, tool writers, and language theorists is necessary when introducing a feature in order to prevent unforeseen consequences from turning it into a burden.

10 Future work

Our data and results suggest a wide variety of future avenues for research, both in terms of expanding our understanding of uses of the preprocessor in practice and in addressing the issues identified by this study.

Comparing how Cpp use in libraries differs from its use in application code may yield insights into the language needs of library authors. Other comparisons, such as Unix vs. Microsoft Windows packages, packages with different application domains or user-interface styles, different versions of a single package, and the like, may also prove valuable.

We did not formally analyze any C++ source code. Preliminary results indicate that many C++ packages rely heavily on Cpp, even for uses where C++ supports a nearly identical language construct. This unfortunate situation probably stems from a combination of trivial translations from C to C++ and of C programmers becoming C++ programmers without changing their habits. A useful analysis of C++ packages would consider the code in the context of both the history of the package and the background of its authors.

Further analysis of the macros with free variables is needed to see which of the roughly 84% of expression macros and 63% of statement macros that lack free variables should be easy to convert to inline functions.

Our framework currently does not benefit from analyzing the context of macro expansions in determining a macro's category. For example, a macro used where a type should appear can be inferred to expand to a type; a macro used before a function body is probably expanding to a declarator.

11 Conclusions

We analyzed 26 comprising 1.4 million of real-world C code to determine how the C preprocessor is used in practice. This paper reported data, too extensive and wide-ranging to be simply summarized here, regarding the prevalence of preprocessor directives, macro body categorizations, use of Cpp to achieve features impossible in the underlying language, inconsistencies and errors in macro definitions and uses, and dependences of code upon macros.

As the first empirical study of the C preprocessor, this article serves to confirm or contradict intuitions about use of the C preprocessor. It indicates the difficulty of eliminating use of the preprocessor through translation to C++, shows the way to development of preprocessor-aware tools, and provides tools including an extensible preprocessor, a Cpp partial evaluator, and a lint-like Cpp checker; we anticipate that the data presented here may be useful for other Cpp-related investigations, as well. In particular, the data and analyses in this paper can provide value to language designers, tool writers, programmers, and software engineers.

Language designers can examine how programmers use the macro system's extra-linguistic capabilities. Future language specifications can directly support (or prevent) such practices — for instance, along the lines suggested in Section 9 — thus imposing greater discipline and structure.

Programming tool writers can choose to cope only with common uses of the preprocessor. By partial preprocessing (or embedded understanding of some Cpp constructs), a parser can maintain the programmer-oriented abstractions provided by preprocessor directives and macro names without getting confused by programs containing syntax errors.

Programmers who wish to make their code cleaner and more portable can choose to limit their use of the preprocessor to the most widely-used and easiest to understand Cpp idioms. Similarly, they can choose to avoid constructs that cause tools (such as test frameworks and program understanding tools) to give incomplete or incorrect results.

Finally, our results are of interest to software engineering researchers for all of the above reasons and more. Since this is the first study of Cpp usage of which we are aware, it was worth performing simply to determine whether the results were predictable *a priori*. Each aspect of our analysis has been surprising and interesting to some individuals. Pursuing more highly focused and deeper analyses along some of these directions could be worthwhile.

Acknowledgments

This work was supported by an IBM Cooperative Fellowship (Ernst) and a National Science Foundation Graduate Fellowship (Badros). We thank Robert Bowdidge, Netta Shani, and the anonymous referees for their comments, which substantially improved the presentation.

References

[ABGM99] David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, CA, May 19-21, 1999.

[BN98] Greg Badros and David Notkin. A framework for preprocessor-aware C source code analyses. Technical Report UW-CSE-98-08-04, University of Washington, Seattle, Washington, August 1998.

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

[CEK+90] L.W. Cannon, R.A. Elliott, L.W. Kirchoff, J.H. Miller, R.W. Mitze, E.P. Schan, N.O. Whittington, Henry Spencer, David Keppel, and Mark Brader. *Recommended C Style and Coding Standards*, 6.0 edition, June 25, 1990.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1991.

[DDL90] J.S. Davis, M.J. Davis, and M.M. Law. *Empirical Foundations of Information and Software Science*, chapter Comparison of subjective entropy and user estimates of software complexity. V. Plenum, New York, NY, USA, 1990.

[Deu90] Peter Deutsch. ansi2knr. Freely available software program, December 1990. Included in the ghostscript distribution from Aladdin Enterprises, ftp://ftp.cs.wisc.edu/ghost/.

[DKR90] A. Dolenc, D. Keppel, and G. V. Reilly. *Notes on Writing Portable Programs in C*, 8th revision edition, November 1990. http://www.apocalypse.org/pub/u/paul/docs/cport/cport.htm.

[EGHT94] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of SIGSOFT '94 Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[Ell92] Ellemtel Telecommunication Systems Laboratory. Programming in C++: Rules and recommendations. Technical report, Ellemtel Telecommunications, Sweden, 1992.

[Eva96] David Evans. *LCLint User's Guide*, August 1996. http://www.sds.lcs.mit.edu/lclint/guide/.

[Fav96] Jean-Marie Favre. Preprocessors from an abstract point of view. In *International Conference on Software Maintenance (ICSM'96)*, Monterey, California, November 4–8, 1996. IEEE Computer Society Press.

[GAM96] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE 1996 Workshop on Program Comprehension*, March 1996.

[GNU] GNU Project. *GNU C Preprocessor Manual*, v2.7.2 edition.

[HS95] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, fourth edition, 1995.

[Joh77] S. C. Johnson. Lint, a C program checker. Computing Science TR 65, Bell Labs, Murray Hill, NJ, September 1977.

[KCR98] Richard Kelsey, William Clinger, and Jonathan A. Rees. The revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Richard P. Gabriel, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–181, Cambridge, MA, August 1986.

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

[Lot98] Christopher Lott. Metrics collection tools for C and C++ source code. http://www.cs.umd.edu/users/cml/cmetrics/, 1998.

[LS94] Panos E. Livadas and David T. Small. Understanding code containing preprocessor constructs. In *IEEE Third Workshop on Program Comprehension*, pages 89–97, Washington, DC, USA, November 14-15 1994.

[MK97] Scott Meyers and Martin Klaus. Examining C++ program analyzers. *Dr. Dobb's Journal*, 22(2):68,70–2,74–5,87, February 1997.

[MNL96] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, March 1996.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[Sal96] Daniel J. Salomon. Using partial evaluation in support of portability, reusability, and maintainability. In Tibor Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 208–222, Linköping, Sweden, 24–26 April 1996. Springer.

[SB99] Maciej Stachowiak and Greg J. Badros. *Scwm Reference Manual: The Authoritative Guide to the Emacs of Window Managers*, 1999. http://vicarious-existence.mit.edu/scwm/.

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Usenix Summer 1992 Technical Conference*, pages 185–197, San Antonio, Texas, June 8-12, 1992.

[Sof99] Gimpel Software. Pc-lint/flexelint. http://www.gimpel.com/lintinfo.htm, 1999.

[SR96] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Proceedings of SIGSOFT '96 Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, October 1996.

[SS92] David A. Spuler and A. Sayed Muhammed Sajeev. Static detection of preprocessor macro errors in C. Technical Report 92/7, James Cook University, Townsville, Australia, 1992.

[Sta94] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, tenth edition, July 1994. ISBN 1-882114-03-5.

[Sta96] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, MA, USA, 2.7.2 edition, June 29, 1996.

[Sta97] Richard Stallman. *GNU Coding Standards*. GNU Project, July 1997. ftp://prep.ai.mit.edu/pub/gnu/standards/standards.texi.

[Ste90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1990. Second edition.

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

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, Albuquerque, NM, June 1993.

[Zel83] Polle T. Zellweger. An interactive high-level debugger for control-flow optimized programs. Technical Report CSL-83-1, Xerox Palo Alto Research Center, Palo Alto, California, January 1983.

[Zip49] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, MA, 1949.