Toward a Language-Agnostic, Syntactic Representation for Preprocessed Code

Jeffrey L. Overbey Matthew D. Michelotti Ralph E. Johnson
University of Illinois at Urbana-Champaign
{overbey2,mmichel3,johnson}@cs.uiuc.edu

Abstract

Preprocessors complicate refactoring because refactoring tools must manipulate *unpreprocessed* code. Working forward from Garrido's work on refactoring preprocessed C [1], we propose a general means by which preprocessor directives can be represented in a syntax tree for an *arbitrary language.* The representation is simple but general, based on the notions of *token-whitetext affixes* and *substitution tokens.* Moreover, we suggest that a preprocessing infrastructure can be *completely independent* of the language being preprocessed, allowing a single preprocessor to be reused among several refactoring tools and allowing numerous preprocessors to be plugged into a single refactoring tool.

1. Introduction

Preprocessing is common and even essential for many languages. The C preprocessor is perhaps the most popular, although many tools—including sed, m4, and even Perl—can serve as preprocessors. The need to parse, represent, analyze, and transform *unpreprocessed* code poses a tremendous challenge to refactoring tools.

First, many of these tools are *stream* preprocessors, oblivious to any lexical or syntactic structure.[1] The substitutions they make do not necessarily occur on token boundaries, nor can they be represented nicely as syntactic units in an abstract syntax tree.

Furthermore, *conditional compilation* (the C preprocessor's `#ifdef` directive) poses an exceptional challenge. Consider the following example.

```
if (
 #if defined(A) || defined(B)
 variable
 #else
 function() < 1 && variable
 #endif
 < 2) x = 3;
```

The usual interpretation—keep one branch of the conditional, and ignore the rest—is insufficient for a refactoring tool (consider renaming *variable).* However, if every branch needs to be included, it is not even obvious how to parse this—much less represent it in an AST.

Garrido [1] addressed refactoring C preprocessed code at length; her seminal work defined many of the problems in the area and proposed solutions in the context of refactoring C. However, some of her solutions are C-specific. This paper generalizes her work, providing an overview of how C preprocessor directives can be handled and eventually represented in an abstract syntax tree, *regardless of the language being preprocessed.* In other words, our model handles C-preprocessed Fortran, Yacc, or Lua as well as C.

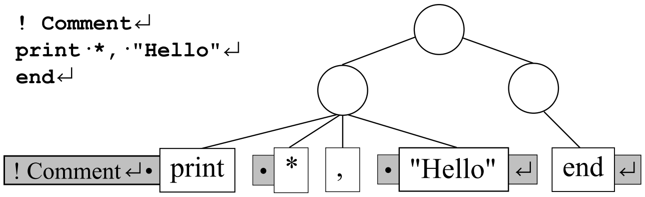
Although we will focus on the C preprocessor [2]—indeed, it is widely-used and its capabilities are stereotypical—the conceptual model we develop is applicable for other preprocessors as well, including the handling of INCLUDE lines in Fortran as well as simple `sed` substitutions. Note also that the focus will be limited to syntactic issues; the preconditions and mechanics involved in *transforming* preprocessed code are a topic in their own right.

2. Pseudo-preprocessing

The need to transform *unpreprocessed* source code requires a *pseudo-preprocessor* [1], a customized preprocessor which (1) provides a way to map preprocessed source code back to unpreprocessed text (e.g., to discover which tokens were in the original source code and which tokens originated from a macro expansion) and (2) makes substitutions (like expanding trigraphs, macros, and file inclusions) but otherwise does *not execute* directives (like `#error` or `#line`), instead passing them on to the lexer/parser for inclusion in the AST.

[1] Technically, the C preprocessor is a *lexical* preprocessor: The text is tokenized as C/C++, and the substitutions are made in terms of these tokens. But when the C preprocessor is applied to other languages (like Fortran), it effectively becomes a stream preprocessor.

3rd Workshop on Refactoring Tools '09 Oct. 26, 2009, Orlando, FL
Copyright © 2009 ACM 978-1-60558-909-1. . . $10.00



```
! Comment↵
print·*,·"Hello"↵
end↵
```

Figure 1. Fortran program and whitetext-affixed parse tree.

3. Representation

Our representation will be constructed assuming that a pseudo-preprocessor feeds a lexical analyzer, which feeds a parser. In practice, the parser would usually build an abstract syntax tree, but our discussion will assume a *concrete* syntax tree (parse tree), which includes every token returned by the lexical analyzer.[2]

Furthermore, it will assume that all of the *whitetext* in a program—comments, whitespace, etc.—is associated with either the preceding or following token. Thus, a token has three fields:[3]

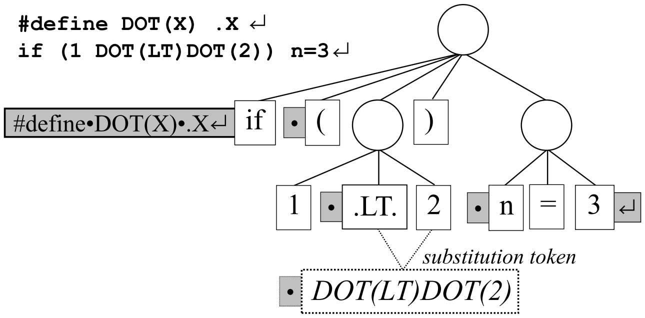
- preceding whitetext,
- token text, and
- trailing whitetext.

If every token is in the parse tree and the whitetext is partitioned among the tokens, then in the absence of preprocessing, the original source code can be reproduced exactly through a simple tree traversal, printing each token's text along with the adjacent whitetext (see Figure 1).

Such a parse tree can be augmented with preprocessing information in such a way that it is possible to reproduce *unpreprocessed* text, with character-for-character accuracy, through a tree traversal. This is intended to demonstrate that preprocessing information is stored at a fine enough granularity to be useful for source code manipulation in a refactoring tool. This preprocessing information will be attached to *tokens* in the parse tree as follows.

- The leading and trailing whitetext may be a sequence of "ordinary" whitetext elements (spaces, etc.) *as well as preprocessor directives.*
- Each token has a fourth field, a *substitution token,* which has the same fields as an ordinary token. (Thus, a substitution token may have its own substitution token.)

The strategy for reproducing source code from a tree traversal will be adjusted accordingly.



```
#define DOT(X) .X ↵
if (1 DOT(LT)DOT(2)) n=3↵
```

Figure 2. Fortran program and parse tree augmented with whitetext and preprocessor directives.

1. If a token's substitution token is non-null, the substitution token is printed *instead of* the original token. This is applied recursively, so if a substitution token has its own substitution token, that will be printed instead.
2. If several adjacent tokens have the same substitution token, it is only printed once.

Now, we will explain how this model is sufficient to represent C preprocessed code. The C preprocessor has three types of directives to consider: substitutions, control lines, and conditional compilation.

3.1 Substitutions

During the first phases of translation, the C preprocessor makes some simple textual substitutions to the input text: removing comments, splicing lines adjoined with backslash-newline sequences, and replacing trigraphs.[4] Similarly, in a later phase, adjacent string literals are concatenated. In these cases, the logical text for some tokens may differ from the physical text in the unpreprocessed file. In these cases, the physical text can be stored in a substitution token.

File inclusion directives and macro replacements can similarly be handled by representing the `#include` directive or macro invocation in a substitution token. This requires taking advantage of rule 2: Since a single `#include` directive or macro invocation will likely expand into several tokens, all of these tokens must have the *same* (pointer-wise) substitution token, so that it will only be printed once. (This is illustrated in Figure 2. The two adjacent macro invocations have been combined into a single substitution token for reasons that will be described in §4.)

Particularly in the case of macros, it may be advantageous for the substitution token's "token text" to be a more complicated representation than a string—for example, a small AST representing the macro invocation. In refactoring, it may be necessary to bind macro invocations to macro definition and substituted tokens to the macro parameters they are

[2] Assuming a CST simplifies the discussion since ASTs generally exclude "uninteresting" tokens, but these may be preprocessed as well; consider the macro `#define LPAREN (`. The "best" way to handle this would depend on implementation details of the AST.

[3] Again, this need not be taken literally. Many tools do not store whitetext in tokens but rather store offset/length information in AST nodes and "expand" this region to include whitetext when manipulating source code.

[4] A trigraph is a sequence of three characters that takes the place of another character. For example, `??(` is equivalent to `{`.

replacing; in these cases, a more structured representation of directives may be desirable.

If a macro invocation expands to nothing, this must be treated as a special case. If such a macro precedes a token, it can be stored in the token's preceding whitetext. If such an invocation occurs in the middle of a token, it must be recorded in a substitution guard for the token.

3.2 Control Lines

Control lines include macro definitions (#define and #undef), the line control directive (#line), #error, #pragma, and the null directive (# followed by a newline).[5] Inserting a directive in the middle of a statement

```
int
#define MACRO
some_function() {}
```

is perfectly legal; directives do not necessarily appear at statement boundaries. So, in general, no assumptions can be made about where directives will or will not need to be placed in an AST.

We will effectively treat these as whitetext. When they precede a token, they are stored in its preceding whitetext (as in Figure 2). If they occur in the middle of a token, a substitution token must be used.

3.3 Single-Configuration Conditional Compilation

As noted in §1, handling only a single configuration of a conditional compilation directive is generally insufficient for a totally-correct refactoring tool. Nevertheless, this is what is done in Apple Xcode, Eclipse CDT, and Microsoft Visual Studio. And it can be handled easily in our existing representation: The #ifdef and all inactive branches can be stored in either the leading or trailing whitetext of an appropriate token; the tokens from only one branch (the "active branch") are passed on to the lexical analyzer. As alluded to earlier, handling multiple configurations is more complicated. We will return to this topic in §6.1.

4. Construction

Most lexical analyzers—particularly generated ones—expect to read from a character stream. They tokenize the character stream and feed the resulting stream of tokens to the parser.

The previously-described representation can be constructed by integrating a pseudo-preprocessor with an existing lexical analyzer as follows. Note that this solution *does not* require modifying the lexer.

1. The pseudo-preprocessor reads the input file, makes substitutions, and writes the result to a character stream.
2. The existing lexical analyzer uses this character stream as its input, tokenizing it as usual.
3. A *directive recovery* module is placed between the lexer and the parser. It reads tokens from the lexer, calls back to the preprocessor to set their substitution tokens and leading/trailing whitetext fields, and finally passes these tokens on to the parser.

Most lexers can associate an offset and length with each token they produce. When a lexer is reading from a pseudo-preprocessed stream, this will be called the token's *stream offset.* This is distinct from the token's *file offset,* its offset in the unpreprocessed file.

When the directive recovery module calls back to the pseudo-preprocessor, it can use the tokens' *stream offsets and lengths* to determine their leading/trailing whitetext and substitution tokens (as well as their file offsets). A complete description of the algorithm is outside the scope of this paper, but the essential ideas are as follows. The pseudo-preprocessor maintains a mapping between its input stream—i.e., the unpreprocessed source file—and its output stream, as illustrated in Figure 3. When the directive recovery module calls back to the pseudo-preprocessor, it passes the stream offset and length of a token returned by the lexer. The pseudo-preprocessor maps that range of characters back to the unpreprocessed stream. If it consists entirely of characters that *did not* result from a preprocessor directive (i.e., those indicated by a dotted line in Figure 3), then the token has no substitution token. Otherwise, it does. Determining the substitution token is often straightforward, but it can become complicated when preprocessor substitutions do not occur on token boundaries. For example, in Figure 3 .LT. is a single token in Fortran, but .LT is covered by one macro invocation, while .2 is covered by the next. In this case, the pseudo-preprocessor must combine these two macro invocations into a single substitution token, and that substitution token must be shared by both the .LT. token and the 2 token in the resulting Fortran parse tree. Again, the complete algorithm will be presented in a future paper.

5. Implementation

Following this model, a pseudo-preprocessor has been implemented by modifying the C Preprocessor in the Eclipse C/C++ Development Tools (CDT 5.0) and has been integrated with Photran, an IDE and refactoring tool for Fortran [3]. The preprocessor was modified to collect whitetext and to act as a stream preprocessor (mimicking gcc -E) rather than as a lexical preprocessor. The directive recovery module was implemented by subclassing Photran's JFlex-generated tokenizer in order to affix whitetext and set substitution tokens. Photran uses a *concretized abstract syntax tree* [4], which internally retains every token returned by the lexical analyzer; it reproduces source code from the AST using the tree traversal strategy described in §3.

[5] The #include directive is also a control line but is treated uniquely for our purposes.

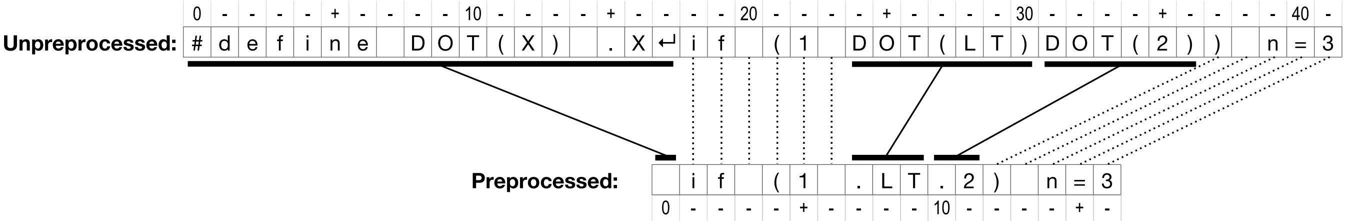


Figure 3. Example of mapping between an unpreprocessed and a preprocessed character stream. Characters that do not result from preprocessor substitutions map directly between the two streams; these are indicated by a dotted line. Solid lines indicate a mapping between a directive (a character range in the input stream) and a character range in the output stream.

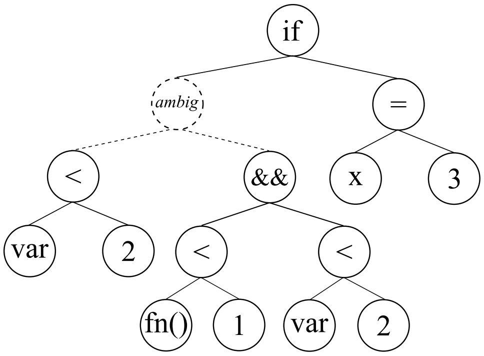


Figure 4. A multiple-configuration AST for the example from §1. The dotted node represents an "ambiguous" expression whose children are each guarded by a different preprocessor configuration.

6. Future Work

6.1 Multiple-Configuration Conditional Compilation

Although multiple-configuration conditional compilation has not yet been implemented, the essential ideas were sketched in a prior paper [4], and details will be published in the future. The fundamental problem is that the branches of an `#ifdef` do not necessarily enclose entire syntactic units (as in the example in §1). Each branch of the conditional must be "expanded" to include some of the tokens preceding and/or following the conditional, until each branch forms a complete syntactic unit. Garrido calls this the *conditional completion problem* [1]. Afterwards, the "completed" branches can be placed under an "ambiguous" node in the AST, as illustrated in Figure 4.

This representation can be constructed by modifying an LALR(1) parser. Essentially, when the parser reaches an `#ifdef`, it makes several clones of itself; a different clone handles each branch of the conditional. Each token following the conditional is fed to all of the clones; clones are merged (and an "ambiguous" node created) as soon as they reach equivalent configurations. In the worst case, this will happen at the end of the input, and the root of the AST will be an "ambiguous" node.

6.2 Preprocessor Composition

Refactoring preprocessed code is further complicated when preprocessors are *composed.* A common example occurs in Fortran programs: C preprocessor directives are handled, then Fortran INCLUDE lines are processed. Adding several sed scripts into this chain is not unheard of. Ideally, every pseudo-preprocessor should be oblivious to what other pseudo-preprocessors, if any, precede or follow it. And, similarly, the lexer should be oblivious to what preprocessor(s) are feeding it. Whether this can be implemented—and how—is a direction for future work.

7. Conclusions

This paper outlines how preprocessor directives can be affixed to tokens in an syntax tree. Whitetext affixes and *substitution tokens* allow syntactic information about preprocessor directives to be stored at a fine enough granularity to reproduce unpreprocessed source code from the tree. This program representation can be constructed by a pseudo-preprocessor, which precedes the lexer, and a *directive recovery module,* which sits between the lexer and parser. Both of these components can be language-independent. Future work will describe the directive recovery algorithm in detail, as well as discuss language-independent handling of multiple-configuration preprocessed code and the composition of arbitrary preprocessors.

Portions of this work were supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and are part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation.

References

[1] Garrido, A.: Program refactoring in the presence of preprocessor directives. PhD thesis, University of Illinois at Urbana-Champaign (2005)

[2] ISO/IEC 9899:1999: Programming Languages – C.

[3] Photran. `http://www.eclipse.org/photran`

[4] Overbey, J., Johnson, R.: Generating rewritable abstract syntax trees. SLE 2008. LNCS 5452, 114–133 (2009)