PROGRAM REFACTORING IN THE PRESENCE OF PREPROCESSOR DIRECTIVES

BY

ALEJANDRA GARRIDO

Licenciada, Universidad Nacional de La Plata, 1997
M.S., University of Illinois at Urbana-Champaign, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

PROGRAM REFACTORING IN THE PRESENCE OF PREPROCESSOR DIRECTIVES

Alejandra Garrido, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2005
Ralph Johnson, Advisor

The C preprocessor is heavily used in C programs because it provides useful and even necessary additions to the C language. Since preprocessor directives are not part of C, they are removed before parsing and program analysis take place, during the phase called preprocessing. In the context of refactoring, it is inappropriate to remove preprocessor directives: if changes are applied on the preprocessed version of a program, it may not be possible to recover the un-preprocessed version. This means that after refactoring, all the source code would be contained in a single unit, targeted to a single configuration and without preprocessor macros.

This thesis describes a novel approach to preserve preprocessor directives during parsing and program analysis, and integrate them in the program representations. Furthermore, it illustrates how the program representations are used during refactoring and how transformations preserve preprocessor directives.

Additionally, the semantics of the C preprocessor are formally specified, and the results of implementing this approach in a refactoring tool for C, CRefactory, are presented.

To Federico, Juan and Manuel,

who teach me what life is about.

Acknowledgements

This project would not have been possible without the support of many people. I would like to sincerely thank my advisor, Ralph Johnson, for his constant guidance and support, and invaluable help during my entire career. Also thanks to my committee members, José Meseguer, Samuel Kamin and Mehdi Harandi, who guided me and encouraged me.

Many thanks to Ira Baxter, who wisely pointed me in the direction of the hardest problem I should concentrate on, and reviewed several papers of this project.

Thanks to John Brant, for helping me in several opportunities with the problems of the C preprocessor, refactoring and his invaluable parser generator, SmaCC. Also thanks to the Software Architecture Group, specially Brian Foote who advertised my research in several opportunities. Thanks to Mark Hills for helping me with Maude.

I would like to thank all my friends, those back in Argentina and all the wonderful people I met in Urbana, who supported me and shared good and bad times.

I thank my parents for always being there for me. Their love, their encouragement and their belief in my abilities through graduate school and maternity were essential to my survival.

My husband deserves at least as much credit as I do for this project. He made it all possible. From his constant encouragement and feedback in all aspects of this project, to his absolute dedication towards our children that turned him into the amazing father that he is. Our children, Juan and Manuel, are the most extraordinary gifts that God could give us. Thanks also to them for their patience during stressful

times and for brightening every day of my life.

Finally, I thank God for granting me the skills and opportunities that made this possible, and blessing my life with the best family I could think of.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Refactoring has become a well-known technique for transforming code in a way that preserves behavior [1; 2]. Refactorings may be applied manually, although manual code manipulation is error prone and cumbersome. Developers and maintainers need tools to make automatic refactorings. There is currently extensive literature on refactoring object-oriented programs and some very good tools for refactoring Smalltalk and Java code [1–7]. However, refactoring tools for C with full support for preprocessor directives have not yet appeared, even when there is a vast amount of systems written in C for which a refactoring tool would be tremendously beneficial [8].

One of the main obstacles to building a refactoring tool for C programs is that these programs have a mix of two languages: pure C and preprocessor directives. Preprocessor directives obey the syntax of Cpp: the C preprocessor. Since the syntax of Cpp is different than the syntax of C, it is necessary to evaluate and remove preprocessor directives, i.e., preprocess a program, before it can be compiled. However, it is inappropriate for a refactoring tool to preprocess a program: if refactorings are applied on the preprocessed version, it may not be possible to recover the un-preprocessed source code with preprocessor directives and macro calls. The preprocessed program becomes unmaintainable [8; 9].

Refactoring C with preprocessor directives poses two main research challenges.

On the one hand, a specialized infrastructure is needed to parse, represent, analyze and transform source code that has a mix of C and Cpp. On the other hand, new precondition and execution rules must be defined for refactorings to preserve behavior in the presence of Cpp directives. We have addressed these challenges by constructing CRefactory, a refactoring tool for C with full support for preprocessor directives.

This chapter is organized as follows: firstly, it presents a background on software evolution and refactoring. Secondly, it outlines Cpp directives and the problems they bring to refactoring. Thirdly, it describes the motivation for this work. Fourthly, it gives an example of a program that has been refactored with CRefactory, so the reader can have a glimpse of the outcome of this work before embarking on the details. Finally, it lists the contributions of this research.

1.1 Background

1.1.1 Software evolution

Software systems must always adapt to changing environments. This is a rule of software evolution, proposed back in the seventies by Manny Lehman as the law of "Continuing Change" [10]. System requirements are transformed or new ones are elicited. Designers are challenged to create and maintain highly reusable components that accommodate to evolving requirements. Code is often rewritten to reflect new functionality or new designs. Moreover, systems experience "Continuing Growth" to maintain user satisfaction [11].

Unless changes are carefully incorporated, code that is constantly modified is in danger of becoming unmaintainable and buggy. This is supported by Lehman's laws of "Increasing complexity" ("As a program is evolved its complexity increases unless work is done to maintain or reduce it" [10]) and "Declining Quality" ("Programs will be perceived as of declining quality unless rigorously maintained and adapted

to a changing operational environment" [12]). Code is also degraded when different people work on it, coding under pressure and without elegance or documentation, without proper planning, design, impact analysis or regression testing. Many systems eventually reach a point where it is easier to throw them away and rewrite them than to make a single change in requirements. As a result, companies are wasting money, time and resources and cannot keep up with good software quality.

It is imperative that before incorporating changes in functionality, the code is reshaped and reorganized in ways that make it more open to changes, easier to read and easier to maintain, in short, more reusable.

1.1.2 Refactoring

The concept of program transformation has evolved from that of "software restructuring" [13], in which local changes are automatically applied by some rules usually in batch mode, to that of "refactoring" [2; 7], where users expect an interactive tool that can apply changes in small steps while they browse or even edit their code.

Refactoring allows improving the design of the code, making it more reusable and flexible to subsequent semantic changes. Refactoring is a disciplined process, so that changes do not affect program behavior and consequently, tests are not violated. A refactoring does not affect program behavior, or in other words, a refactoring preserves program behavior, when the versions of the program before and after refactoring are semantically equivalent. That is, the mapping of input to output values remains the same [2]. A typical example of a refactoring is the renaming of a variable. A complex example of refactoring is the application of a design pattern [14].

The Software Architecture Group at the University of Illinois at Urbana-Champaign has been researching on refactoring for a long time ([2; 6; 7; 15–19]). William Opdyke and Ralph Johnson were the first to coin the term "refactoring" [17]. Turning research into practice, Don Roberts and John Brant built the first successful refactoring tool

for the Smalltalk language [7].

Research on refactoring has spread, and well-known refactoring techniques have been catalogued as step-by-step recipes to help maintainers with a manual process [1]. However, most of the literature about refactoring is concentrated on object-oriented languages like Smalltalk or Java and transformations in the inheritance hierarchy ([1; 14; 20]).

A non object-oriented language like C calls for a different set of refactorings meaningful to its imperative, pointer-and-struct based nature. In an earlier work [16], we have proposed a catalog of refactorings for the C language that were implemented in a prototype tool. Since then we have aimed at adding full support for Cpp directives in refactoring.

This thesis and related publications ([8; 9; 21]) add refactorings for Cpp directives and specify necessary changes for existing C refactorings to account for the presence of directives. We believe that our results will not only foster the development of future refactoring tools for C, but also the analysis of code with Cpp directives.

1.2 The C Preprocessor and the Problems it Brings to Refactoring

C is a relatively "low level" language and as such, it allows the programmer to do low level manipulation of objects like characters, numbers and addresses [22]. Moreover, C provides several language facilities by means of a preprocessor, often known as `Cpp`. The C preprocessor enhances C in many ways, as it allows the definition of constants, the abbreviation of repetitive or complicated constructs, the manipulation of types as first class objects, program configuration with conditional compilation and partition of programs in multiple files, among plenty of other uses.

Ernst, Badros and Notkin analyze and categorize different uses of Cpp directives

from several publicly-available C software packages [23]. Their paper emphasizes that Cpp is heavily used in C programs since it provides for significant flexibility, but its ability to perform arbitrary source code manipulations complicate the understanding of C programs by programmers and tools. Stroustrup also recognizes that "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" [24].

Preprocessing occurs before compilation, transforming a program by a series of textual replacements. These replacements include removing comments, converting the input file into a sequence of tokens (tokenization), executing directives and expanding macros [25]. Preprocessor directives start with '#' and their syntax is completely independent of the syntax of the C language [26].

Cpp provides the following directives:

#include Allows the inclusion of header files. It causes Cpp to scan the specified file as input before continuing with the rest of the current file.

#define Allows the definition of macros. A macro is a text fragment which has been given a name. This directive causes Cpp to substitute all occurrences of the macro name by its replacement text.

#if, #ifdef, #ifndef, #elif, #else, #endif Allows for conditional inclusion based on configuration settings. It instructs the preprocessor to evaluate a condition during compilation and depending on that value, select whether or not to include a piece of code in the output.

#line Modifies the preprocessor's value of the current file name and line number that it provides to the compiler.

#error Causes Cpp to report a fatal error. The text following the directive name is used in the error message. This directive usually appears inside conditional

compilation directives to detect a combination of configuration parameters that are inconsistent.

#pragma This directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The action specified is implementation-dependent.

The heavy use of Cpp directives in C programs and the complexity that they add to program understanding call for tools that correctly deal with Cpp as well as C. Moreover, a refactoring tool cannot apply transformations on the preprocessed version of a program or the un-preprocessed version may be irrecoverable, as will be described in Chapter 4.

In order to handle Cpp directives in a refactoring tool, there are two main problems that need to be solved. Firstly, the syntax of Cpp must be integrated with the syntax of C to be able to parse C programs and create representations of them. As Chapter 4 will show, seamless integration of both languages is not possible, so a special preprocessing is needed that does not remove Cpp directives but transforms conditional directives and expands macro calls to allow for syntax integration. This is called *pseudo-preprocessing*.

Secondly, file inclusion, macros and conditional compilation directives introduce additional problems during the application of refactorings. For example, file inclusion extends the scope of global entities while conditional directives restrict scope to some part of the program; the same macro may refer to different program entities depending on the context of each individual macro call; macros may use concatenation to produce the name of an entity; conditional directives usually introduce different definitions for the same name. These problems and others are described in detail in Chapter 6, and their solution is specified for some refactorings in the form of new preconditions and transformation rules.

Currently, there are no refactoring tools, not even analysis tools for C that handle the C preprocessor completely and correctly. Languages like Smalltalk and Java with mature refactoring support do not have to deal with this problem because these languages do not require a preprocessor, so the same source code is the input to the parser and the refactoring engine. Chapter 2 reviews existing program analysis tools for the C language and refactoring tools for Smalltalk, C and C++.

1.3 Motivation

We are specially motivated by the lack of refactoring tools for C that handle preprocessor directives completely and correctly.

A few years ago we were involved in a project with a flight management system written in C that had new requirements. The system was so poorly maintained that they were about to start from scratch. The man-machine interface subsystem had three major problems: code duplication, poor data structures and global access to variables. Refactoring was crucial before any change in functionality could be applied. Unfortunately, there was no refactoring tool that could help us in the process. We wrote some Emacs scripts to do the simplest renamings and manipulated the code by hand.

From this experience our goal became to develop a refactoring tool for C. The first step towards that goal was to create a catalog of refactorings for the C language and to implement some of them, as renaming of different symbols and grouping of variables in a structure [16]. Then we realized it was critical to support preprocessor directives, for the reasons previously described.

While trying to add support for Cpp directives we found that they were not only an obstacle for refactoring tools, but for analysis or program understanding tools, which have been ignoring the problem and provide information on the preprocessed version

of a program. This information is therefore only partial, applicable for example to a single program configuration.

Our goal was as such extended to come up with a a general methodology to deal with C and Cpp simultaneously, and to construct representations of C programs with preprocessor directives.

1.4 CRefactory: a Refactoring Tool for C

CRefactory has been implemented in VisualWorks Smalltalk^{TM}. The refactoring engine mimics the design of the Smalltalk Refactoring Browser [7]. To load a program, CRefactory needs to know the source files, include directories, read-only directories (those that contain non-modifiable files, like standard library headers), command line macros, false conditions and incompatible conditions. CRefactory processes all possible program configurations simultaneously by considering all Cpp conditional branches to be potentially true, except for those that the user specifies in the input as having "*false conditions*". Moreover, the space of all possible configurations is calculated by producing all combinations of configuration variables. The user may force some combinations to be discarded by specifying "*incompatible conditions*" in the input. This is described in detail in Chapter 4.

This section provides some examples on the kind of refactorings that can be applied in the current version of CRefactory. The case study used is rm, the "remove file" utility in Unix platforms. The input to CRefactory required a single source file: rm.c, although loading the program included 94 header files. Only one of these headers is changed by the refactorings listed below: remove.h, which is in the same directory as rm.c. Appendix D lists the source code of files rm.c and remove.h in their original versions and their final versions after refactoring.

The list of refactorings applied were the following:

1. Rename '`x`' in function '`rm_option_init`' to '`opts`'.

   This refactoring is local to function '`rm_option_init`', of which '`x`' is a parameter. All preconditions pass so the refactoring executes successfully. Figure 1.1 shows the original version of the function on the left and the refactored version on the right.

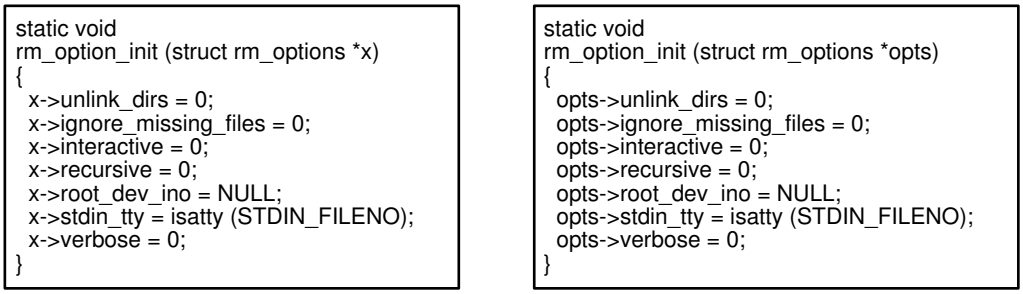

Figure 1.1: Original and refactored versions of function "rm_option_init"

2. Rename '`x`' in function `main` to '`opts`'.

   This refactoring is local to function '`main`' and has no interference with the previous refactoring. All preconditions pass and the refactoring executes successfully. See the refactored version of rm.c in Appendix D.

3. Rename macro '`LC_ALL`' to '`LOCALE_ALL`'.

   In this case, the user selects the call to macro '`LC_ALL`' in function '`main`' and chooses to rename it. This refactoring is not valid because '`LC_ALL`' is defined in file /usr/include/locale.h, a library header whose directory has been set as read-only. For that reason, the preconditions of this refactoring fail and return the appropriate error without changing the code.

4. Rename structure field '`unlink_dirs`' to '`unlink_directories`'.

   The user selects a reference to '`unlink_dirs`' in function '`main`'. This is a field of the structure with type '`struct rm_options`' defined in file 'remove.h'. The refactoring is valid and its scope is global. It changes the two uses of

'unlink_dirs' in file rm.c (one in function 'rm_option_init' and the other in function 'main)' and the structure definition in file remove.h.

5. Add variable 'preserve_root' as a field of structure 'rm_options'.

   This refactoring has the same scope as the previous one. It adds 'preserve_root' as a field in the definition of 'struct rm_options' in file remove.h. It removes the definition of 'preserve_root' from function 'main' in rm.c and warns the user that its initialization is lost in this step. The user needs to take care of adding the initialization for the new field in function 'rm_option_init'. Also, the refactoring replaces the uses of the variable by uses of the structure field.

Appendix D shows the source code of rm.c and remove.h after the previous refactorings have been applied and the initialization of the new field 'preserve_root' has been added to function 'rm_option_init'.

Although not used in this example, other refactorings currently implemented in CRefactory are: 'Rename Function', 'Delete Unreferenced Variable', 'Change to Pointer' (adding a level of indirection) and 'Change from Pointer' (removing a level of indirection). A discussion of these and other refactorings applicable on C programs appears in Chapter 6.

1.5 Contributions

The contributions of this research are:

- A formal specification of the C preprocessor (Chapter 3).
- A new method for preprocessing that does not remove preprocessor directives but prepares the source code for parsing (Chapter 4).
- Design of program representations that integrate C and Cpp (Chapter 5).

- The identification of problems posed by Cpp in refactoring (Chapter 6, Sections 6.1 to 6.3).
- Specification of the new preconditions and transformation rules for some refactorings, which solve the problems identified (Chapter 6, Section 6.4).
- A catalog of refactorings for Cpp directives (Chapter 6, Section 6.5).
- Measurements on the program representations when loading a few case studies (Chapter 7).
- A refactoring tool for C: CRefactory.

Chapter 2

Related Work

The ability to represent and analyze preprocessor directives as part of a C program is not only necessary for a C refactoring tool but it is also a highly desirable feature for a program analysis or program understanding tool. There is a considerable number of program analysis and program understanding tools for C, although few of them recognize the importance of preprocessor directives and provide some support for them. The first section of this chapter reviews some of these tools.

As mentioned in Chapter 1, there is no refactoring tool for C with full support for preprocessor directives. Nevertheless, there is a transformation tool and a refactoring tool that handle directives to some degree. They are presented in the second section of this chapter. That section also describes the Refactoring Browser for Smalltalk, since its design had a substantial influence on CRefactory.

Last but not least, the third section reviews refactoring approaches for C++.

2.1 Program Analysis and Understanding Tools

Program analysis tools extract information from the source code and construct useful representations or reports to improve program understanding. They also support metrics, testing and re-engineering.

There is a considerable number of tools that perform static program analysis on preprocessed C code. Except for a few cases, these tools do not include any information of preprocessor directives. Examples of program analysis tools for pure C language are: Cscope [27], the Combined C Graph [28], GENOA [29], Rigi [30], CStar [31].

The GENOA framework [29] implements a specification language to generate code analyzers. Genoa reuses an existing parser for a given programming language that outputs a parse tree. The scripting language of GENOA allows analysis functions by traversing parse tree data structures built by such parsers. Genoa was used to create GEN++, a meta-case tool for creating C++ source analysis tools [32]. As existing C parsers do not save information about preprocessor directives, no analysis can be performed on them.

Rigi [30] uses existing parsers to extract software artifacts and store them as graph structures. Subsystems are semi-automatically identified in the graph to build multiple layered hierarchies of abstraction.

Program slicing is a different analysis technique based on the control and data flow information. Weiser defined a "program slice" as the set of statements that do not affect or influence a given variable [33]. Weiser discovered that computing program slices can be very useful when debugging. However, experiments showed that slices are often too large to be generally useful. Recent studies aimed at improving the calculation of pointer analyses to reduce the slice size [34; 35]. Program slicing is supported by various tools like the Star Diagram [31; 36], which provides a hierarchical representation of references to chosen variables or data structures, and the Wisconsin Program-Slicing Tool [37].

Let us now turn into some tools and approaches that allow a partial analysis of Cpp directives. We briefly describe the following: CIA [38], "Understand for C++" [39], PCp^3 [40], LCLint [41], Parsing Minimization technique [42], Ghinsu [43], XML

tagging technique [44] and GUPRO [45].

CIA [38] stores information on macros such as the lines where they are defined, and the functions or global variables that refer to them. CIA stores all the information in a database, which allows users to define their own queries and reports. The disadvantage of CIA is that it is not customizable, since the type of information it looks for is fixed.

The commercial tool "Understand for C++" [39] provides some analysis data such as calls/caller cross-referencing and metrics for C++ code. It may show includes/ included-by trees. It can also report information on macros, such as where they are declared and where they are used. However, the user must specify which macros should be used in the analysis and their definition. Parses appear to be based on a single-preprocessor pass that the user must configure.

The framework PCp^3 [40] allows the analysis of C source code with preprocessor directives by providing "hooks" in the preprocessor or in the parser. That is, the code is preprocessed but the user can define callbacks in the Perl scripting language, making use of those hooks in the preprocessor. PCp^3 can provide useful information about Cpp directives. However, the program representations that PCp^3 can produce would still be based on a single configuration and so be inappropriate for refactoring.

Tools like LCLint [41] operate on unprocessed source code to detect bugs in typical C programs including preprocessor directives. LCLint uses approximate parsing, i.e., it constructs an approximate model of the source code's appearance to a compiler. These tools are therefore inappropriate for program analysis that requires exact or conservative information [40].

Somé and Lethbridge argue that program understanding tools should provide information about each configuration in which an entity can be considered [42]. They propose some heuristics to detect the configurations that can be parsed in the same pass, therefore minimizing the number of passes needed. This approach may still be

exponential for large programs. Our approach, instead, is to come out with a *single* representation of the program's source code, which includes the code for all possible configurations but still can be parsed in a *single pass*.

Livadas and Small [43] argue that preprocessor directives, especially macros, can complicate maintenance because program analysis results cannot be reflected back in the un-preprocessed source code. Their paper describes a preprocessor developed for the Ghinsu environment that captures several mappings between preprocessed tokens and their counterparts in the original source code. These mappings are then maintained in Ghinsu's internal representations and allow the tool to highlight the results of program analysis in the original source code.

Cox and Clarke [44] present a fine-grained approach where they markup every character in the preprocessor output with the history of preprocessor replacements from which it was derived. This provides an exact mapping of preprocessed characters to the un-preprocessed source code, by way of XML tags. The drawbacks of this approach are that it works on a single configuration at a time and that it takes up considerable space. We found out that tagging at the level of each individual character is not necessary. The paper does not address the issues of representing the tagged output in other program representations like the abstract syntax tree.

Kullbach and Riediger [45] present a similar approach where the preprocessor generates, besides the usual Cpp output, a representation that maintains original source coordinates, conditionals and macro replacements. Their program understanding tool called GUPRO can then display the source code with unexpanded macros (folded macros) or expand them one level at a time (macro unfolding).

2.2 Related Refactoring and Transformation Tools

Although catalogs of refactorings are helpful, manual code manipulation is error prone and cumbersome. A program of more than ten thousand lines of code may even make manual manipulation unfeasible. Therefore, maintainers need tools to make automatic refactorings. Many people now think that refactoring tools are going to make a big impact in the software engineering process [46].

Tools for refactoring Java code are improving. Good examples are jFactor [5], IDEA [47] and Eclipse [3]. Yet the model for refactoring tools is the Refactoring Browser (RB) for Smalltalk [7], which is described next. We found a refactoring tool: Xrefactory [48; 49] and a transformation tool: DMS [50], which handle C and provide some level of preprocessor awareness. They are described in subsequent sections.

2.2.1 Refactoring Browser

The Refactoring Browser is integrated in the Smalltalk development environment and allows for powerful and fast refactorings that guarantee to preserve behavior [4]. As Martin Fowler says, not many people use Smalltalk, but "the Refactoring Browser makes one thing very clear. Tool support for refactoring is both possible and valuable." [46].

The Refactoring Browser and the philosophy behind it have been major influences for CRefactory. As described in our previous work [16], we share the same goals of having an interactive, fast and usable tool, that users can trust to perform their refactorings. Moreover, CRefactory's transformation engine has the same design as its counterpart in RB. In both tools, refactorings are executed in two steps: check the preconditions and apply the transformation. Preconditions are checked by querying a "program model" [51], or as we call it, a "program repository", which is populated during compilation and updated during refactoring. Preconditions are represented by

"Condition objects" that can be composed and know how to check their truth value.

Applying the transformation consists of another two steps: first, rewrite the abstract syntax tree by way of a parse tree rewriter and second, create change objects that update the "program model". These change objects have the capability of implementing "undo". The program model, however, looks very different in both cases, mainly because Smalltalk and C are so dissimilar. Moreover, the program model comes for free in any Smalltalk environment, while we had to implement the program model for C.

2.2.2 Xrefactory

The tool Xrefactory from Xref-Tech comes as a plug-in for Emacs and XEmacs [48]. The advantage of Xrefactory is that it allows the user to apply refactorings on the un-preprocessed code, and provides some support for macros. Current refactorings include renaming, function extraction, insertion, deletion and moving of parameters. Renaming is also possible on macros and macro parameters. When renaming a symbol used inside a macro's body, the macro's body is changed accordingly. Moreover, if the macro refers to more than one definition of the symbol depending on the context, the tool warns the user of the possible problem. However, the tool does not seem to handle macro concatenation during renaming, since it returns an internal error during the refactoring. Furthermore, when we tried renaming a macro that was undefined later on, the tool changed the name in the #define directive, but it did not change the name in the #undef.

Xrefactory does not work correctly during Extract Function, in the case where there are different definitions for a macro in the place where the new function goes. Figure 2.1 shows a sample code on the left, on which we applied extract function of the selected piece. The code on the right is the result of the refactoring. Behavior was not preserved by the refactoring, since the code on the left initialized `x` to 1 and

printed ‘1’, whereas the refactored code on the right initializes `x` to 2 and prints ‘2’.

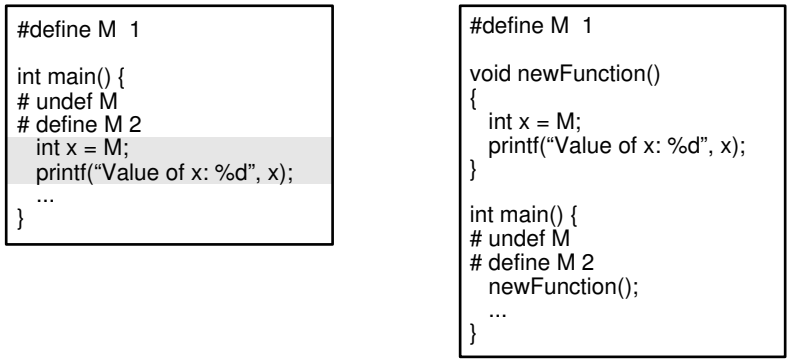

Figure 2.1: Error in Extract Function Refactoring

Xrefactory also provides some support for file inclusion. For example, renaming a global symbol worked correctly when the symbol was used in a source file but defined in a header file. However, the tool reported an internal error in the case of two source files using a symbol defined in a header file. Moreover, when we tried renaming a library function (printf, defined in “stdio.h”), the tool changed the source file without warning that it was breaking the code because it could not change the header file.

Xrefactory can handle more than one configuration at a time. The user must specify the possible configurations, after which the program is processed multiple times and multiple representations result, one for each specified configuration [49]. Although this is a significant improvement over refactoring tools that work on a single configuration, this approach still cannot scale up to any medium size program with 10 or more configuration variables.

2.2.3 DMS

The tool DMS, an acronym for “Design Maintenance System”, is being constructed at Semantics Designs, Inc. with the vision of supporting the whole development and maintenance cycle [52]. With that vision, DMS provides a very general infrastructure that includes a generalized compiler for context-free grammars, program representa-

tions that include ASTs and symbol tables, an attribute evaluator to define analysis functions on the ASTs and a transformation engine. This transformation engine uses a database of pre-defined rules, which can also be augmented, and applies classic compiler optimizations [50]. Figure 2.2 shows an example of a DMS rewrite rule [52]. The transformation engine can also remove dead code branches from preprocessor conditionals tied to a discontinued system configuration [53].

```
default domain C.
rule auto_inc(v:lvalue):
   statement->statement =
   "\v = \v+1;" rewrites to "\v++;"
   if no_side_effects(v) .
```

Figure 2.2: Example of a DMS transformation rule

The advantage of DMS over Xrefactory is that it can handle multiple configurations processing the program a single time. DMS is able to parse conditional directives by allowing them at certain, predefined places in the grammar. With this approach, DMS can parse 85% of un-preprocessed C files [53]. DMS can also parse macro calls, without expanding them, when they appear in predefined places and contain complete syntactical units.

DMS also provides a type checking tool based on a symbol table in which the type of each symbol is conditioned by an expression [54]. This is similar to CRefactory's symbol tables.

When the CRefactory project was in the early stages, we considered using DMS as its infrastructure. The idea was appealing since we only had to build the refactoring engine on top of it. We decided against it for two main reasons:

- We envisioned a tool without restrictions on the content or the placement of macros and other preprocessor directives;
- DMS can apply transformations on batch mode. It does not include a graphical interface. A refactoring tool needs to be interactive, allowing the user to choose

which refactoring to apply on which piece of code. Our goal is to support interactive code manipulation through smaller refactorings that are selected by the user with a drop-down menu while visualizing the code.

2.3 Refactoring for C++

Since C++ is an extension of C, both languages are highly related. Moreover, C++ still uses the C preprocessor. Therefore, the contributions of this thesis should be directly applicable to C++.

The first work on refactoring for C++ comes from William Opdyke [2]. His dissertation describes several refactorings in terms of the preconditions that they should satisfy to preserve behavior. Except for the refactorings on classes and inheritance hierarchies, the others provide the basis for our refactorings. Our work enhances the preconditions of Opdyke's refactorings to support preprocessor directives. His thesis also list the analysis functions necessary to check for preconditions, so they become the starting point for CRefactory's analysis functions.

The other work on refactoring of C++ comes from Tokuda and Batory [20]. The refactorings they propose, however, apply to the class diagrams of an application rather than to source code, and do not provide support for preprocessor directives. Their refactorings are more high-level, like the application of a design pattern.

There are two commercial tools that advertise refactoring support but do not appear to include the preprocessor. Ref++ is a refactoring plug-in for Visual Studio.net [55], and it is "only for native C++ source code". SlickEdit is a C++ editor that includes some refactorings, including Extract Method [56]. The user supplies configuration values so it works for a single configuration.

Yet another related area of research is that of transforming C code into C++. Fanta and Rajlich propose a tool-set to find potential classes in a C program and

restructure the code into classes [57]. However, their tool preprocesses the C code first, so transformations are applied on a single configuration at a time. They propose to discover all configuration variables by examining the code, apply the transformations on each configuration separately and recombine the results into a single output [58]. This solution is not only very expensive but is also impractical for a large system. The Linux kernel (version 2.6.7) has about 1,672 binary-valued configuration variables. The number of possible configurations is huge. Even a small program like Flex, with less than 20K lines of code among 21 files, has 5 configuration variables with binary value, which make up a space of 2^5 possible configurations.

Chapter 3

The C Preprocessor

The C language depends on its preprocessor for many useful features. The C preprocessor (`Cpp`) lets programmers divide the program into manageable parts, customize the code for different platforms or C dialects and define constants and constructs that can be used on any data type. Cpp is controlled by special commands called *preprocessor directives*. Preprocessor directives start with '#' and their syntax is completely independent of the syntax of the C language [26]. In fact, Cpp is often used with other languages, such as Fortran or C++.

Preprocessing occurs as a separate phase before compilation. During preprocessing, the program is transformed as a series of textual replacements and tokenized, as described in the following steps.

1. *The source code of the input file is read into memory and broken into lines. Comments are removed.*

   A line is the unit of processing for Cpp. The only structure that Cpp assumes for a C program is that it is a sequence of lines. A line can be continued by placing a backslash character at the end. Cpp removes backslash characters and merges continued lines into one.

   Comments in standard C begin with '/*' and end with the first subsequent occurrence of '*/'.

2. *The source code is converted into a sequence of preprocessing tokens (tokenization).*

   In this step, Cpp works as a scanner, separating the source code into tokens [25]. Preprocessing tokens are the same as C language tokens with a few exceptions: they include file names and Cpp directives, they can be concatenated by the '##' operator, white spaces separate tokens and C keywords are treated as plain identifiers (see [25] for a complete list of exceptions).

3. *Directives are executed and macros are expanded.*

   A Cpp directive is a line that starts with '#' followed by the directive name. The execution of directives causes the inclusion of header files, the definition of macros, the exclusion of tokens depending on configuration conditions, warnings or error reporting and changes in line numbering for the compiler. This results in a new sequence of tokens, which is the output of Cpp to the compiler.

This chapter defines the syntax and semantics of Cpp, using the Maude specification language [59]. This specification will help to understand how the syntax of Cpp will be integrated into the syntax of C and how Cpp's behavior can be mimicked without removing its directives. Appendix A contains the complete listing of Cpp's specification. This specification is divided into a series of Maude modules, each giving semantics to a particular language feature, similar to the approach taken by Roşu in his Programming Language Design class [60].

3.1 An Introduction to Maude

This section presents the specification of the basic elements of Cpp's syntax: *tokens* and *lines*. As described in the introduction of this chapter, Cpp structures a C program as a sequence of lines. A line is either a Cpp directive or a sequence of tokens ending in carriage return. The specification of tokens and lines is used to

introduce the syntax and key concepts of Maude, although we recommend the official Maude documentation [61–63] for a detailed description of Maude and its logical foundations.

Maude is an executable specification language that models systems and the actions within those systems [60; 63]. The key concept of Maude is a *module*. A module is a set of definitions that represent an algebra, i.e., a collection of sorts and the operations on these sorts. Cpp's specification uses *functional modules*, which add equations for the algebra to satisfy, yielding an equational theory [61].

Figure 3.1 shows a Maude functional module called TOKEN. This module specifies *tokens* and *token sequences*. The parenthesized numbers at the left of the figure are not part of the module specification but provide line numbering for easy reference in the detailed explanation that follows.

```
(1) fmod TOKEN is
(2)   protecting IDENTIFIER .
(3)   sorts Token TokenSequence .
(4)   subsorts Identifier < Token < TokenSequence .
(5)   op nil : -> TokenSequence .
(6)   op __ : TokenSequence TokenSequence -> TokenSequence [assoc id: nil] .
(7) endfm
```

Figure 3.1: Specification of the functional module TOKEN

A functional module starts with the keyword `fmod` and ends with `endfm`. Inside the module, statements are separated by periods, which should have white spaces before and after.

Module TOKEN expands on a previously defined module called IDENTIFIER. For this reason, line 2 in Fig. 3.1 imports module IDENTIFIER by using the keyword `protecting`, which can also be abbreviated with `pr`. Another way of importing a module is by using the keyword `extending` or `ex`, which is used to change the meaning of the imported module [63].

Sorts are declared with the keywords `sort` or `sorts`, depending on whether there is one sort, or more than one sort, being declared. In line 3 of Fig. 3.1, two sorts are

being declared: Token and TokenSequence.

The set of declared sorts can be partially ordered by a *subsort* relationship. The keywords `subsort` or `subsorts` and the "<" character are used for this purpose. The subsort relationship $s \leq s'$ is interpreted semantically by the subset inclusion $A_s \subseteq A_{s'}$ between the sets A_s and $A_{s'}$ of data elements associated to s and s' in an algebra A [61]. In line 4 of Fig. 3.1, the expression `Identifier < Token` means that an Identifier *is a* Token and a Token can be an Identifier or something else (the sort Identifier is defined in the module IDENTIFIER; see Appendix A). Moreover, the expression `Token < TokenSequence` says that a Token is also a TokenSequence (a token sequence with just one element). This is the way to express in Maude that a TokenSequence is composed of Tokens.

Operations are declared using the keyword `op` followed by the name of the operation, then a colon, then the sorts of the arguments, then an arrow, and then the sort of the result. Maude understands both prefix and mixfix notation for operations. When declaring an operation with mixfix notation, the underscore character is used to specify the places for the arguments. Lines 5 and 6 in Fig. 3.1 define two operations, which are actually constructors of a TokenSequence. Line 5 defines nil as the empty TokenSequence and line 6 defines the concatenation of TokenSequences, by juxtaposition. This means that TokenSequences are concatenated by placing a white space in between.

The declaration of the operation in line 6 has something else after the sort of the result: *equational attributes*. A binary operation in Maude can be declared to satisfy some equational axioms like associativity (with the keyword `assoc`), commutativity (with the keyword `comm`), identity with respect to an identity element (keyword `id`), etc. In line 6, the concatenation of TokenSequences is declared to be associative and to have nil as its identity element.

Let us now turn into the specification of Lines. Figure 3.2 shows two more

functional modules. The module CPP-DIR-SYNTAX declares the sort CppDirective. Module LINE-SEQ-SYNTAX extends CPP-DIR-SYNTAX and TOKEN and declares the sorts Line and LineSeq (a line sequence). A detailed description follows the figure.

```
fmod CPP-DIR-SYNTAX is
  sort CppDirective .
endfm

fmod LINE-SEQ-SYNTAX is
  pr CPP-DIR-SYNTAX . pr TOKEN .
  sorts Line LineSeq .
  subsorts CppDirective < Line < LineSeq .
  op nilLS : -> LineSeq .
  op __ : LineSeq LineSeq -> LineSeq [assoc id: nilLS] .
  op _cr : TokenSequence -> Line .
  op _\`cr_ : TokenSequence Line -> Line .

  vars TS1 TS2 : TokenSequence .
  eq TS1 \ cr TS2 cr = (TS1 TS2) cr .
endfm
```

Figure 3.2: Basic elements of Cpp syntax. A Cpp directive, a line and a line sequence

The purpose of having module CPP-DIR-SYNTAX is to define the super-sort of all Cpp directives once, and then import it in the modules for each individual directive, where each such module declares a subsort of CppDirective.

Inside LINE-SEQ-SYNTAX, the subsort relations specify that a Line may be a CppDirective and that a LineSeq is composed of Lines. The operation nilLS constructs an empty LineSeq. The next operation defines the concatenation of LineSeqs in the same way as the concatenation of TokenSequences, with empty juxtaposition syntax. The operation cr (representing a carriage return) applied to a TokenSequence also constructs a Line as seen by Cpp. The last operation in the module defines the concatenation of two subsequent lines with the backslash character.

The new features in module LINE-SEQ-SYNTAX are variable declarations and equations. Variables are declared with the keywords `var` or `vars`, followed by the variable name(s), a colon and the sort to which the variable(s) belong. Equations define the properties that the operations should satisfy. Equations start with the

keyword `eq` followed by two expressions separated by an "=" character. The equation in the figure shows how the operation \ works to concatenate two lines.

3.2 Behavior of Cpp

This section gives a high-level view of the behavior of Cpp over Lines and LineSeqs (line sequences). The specific behavior of Cpp with file inclusion, macro definition and conditional compilation directives will be described in subsequent sections. The other directives are not relevant in this research, since they do not cause problems in refactoring.

In order to describe the behavior of Cpp, it is first necessary to define its *state*. The state of Cpp is a data structure that contains information about each directive that has been processed, (e.g., macro definitions, truth value of current Cpp conditional branch, etc.) and where the output is incrementally built. The state will change as Cpp consumes input tokens and produces output tokens. Fig. 3.3 shows a preliminary version of module CPP-STATE; other elements will be added when specific Cpp directives are described. For example, include directories will be added to support `#include`, a macro table will be added for `#define` and the final version in Fig. 3.16 will add a few more elements to support conditionals.

```
fmod CPP-STATE is
  pr TOKEN .
  sorts CppState CppStateAttribute .
  subsort CppStateAttribute < CppState .
  op empty : -> CppState .
  op _,_ : CppState CppState  -> CppState [assoc comm id: empty] .
  op outputStream : TokenSequence -> CppStateAttribute .
endfm
```

Figure 3.3: State of Cpp during preprocessing

The sort CppState describes the state of Cpp during preprocessing, as a set of attributes, where attributes are defined as operations on the data they store. The sort of attributes is CppStateAttribute. An empty CppState is constructed by the operation

empty. A comma concatenates CppStateAttributes to form a CppState. Specifying ',' as an associative and commutative operation, with empty as the identity element, defines CppState as a *multiset* [61]. Defining the state as a multiset of attributes allows us to abstract away from the concrete order and number of attributes when defining operations on the state. The only attribute described in Fig. 3.3 is outputStream. It stores the TokenSequence that Cpp builds as it preprocesses the input and that it returns at the end of preprocessing.

Cpp's behavior is defined in terms of the operation state, which given some input and a CppState, modifies the state accordingly. The operation state on a LineSeq and a CppState applies the operation state to each line in the sequence subsequently, modifying the CppState each time. The operation state when the first argument is a single Line processes each token on the line. Moreover, the modules defining the semantics of each Cpp directive will add their own behavior for the state operation.

The external interface of Cpp is defined with an operation called preprocess, which given a file name, reads the line sequence of the file, constructs the initial CppState and applies the operation state on them. Figure 3.4 presents the high-level view of Cpp's semantics, with definitions for the operations preprocess and state.

Module CPP-DIR-SEMANTICS defines the signature of the operation state when the input is a CppDirective. Modules defining the behavior of specific directives will provide equations to define this operation.

Module LINE-SEQ-SEMANTICS defines the state operation on a LineSeq. If the input is a line with no tokens, or if it is an empty LineSeq, the state is unmodified. If the input is a single Line (second equation), the first token is consumed and is appended to the outputStream. Note that in this equation, S represents the subset of all other attributes in the state apart from outputStream. The semantics on a non-empty LineSeq is to apply the state operation to each Line in turn.

Module CPP-SEMANTICS defines two operations: preprocess and returnOutput.

```
fmod CPP-DIR-SEMANTICS is pr CPP-DIR-SYNTAX .
  pr CPP-STATE .
  op state : CppDirective CppState -> CppState .
endfm

fmod LINE-SEQ-SEMANTICS is pr LINE-SEQ-SYNTAX .
  pr CPP-DIR-SEMANTICS .
  op state : LineSeq CppState -> CppState .
  var L : Line .  var LS : LineSeq .  var S : CppState .
  eq state(nil cr, S) = S .
  eq state((T TS) cr, (outputStream(O), S)) = state(TS cr, (outputStream(O T), S)) .
  eq state(nilLS, S) = S .
  eq state(L LS, S) = state(LS, state(L, S)) .
endfm

fmod CPP-SEMANTICS is pr CPP-SYNTAX .
  pr HELPING-OPS . ex LINE-SEQ-SEMANTICS .
  op preprocess : String -> TokenSequence .
  op returnOutput : CppState -> TokenSequence .
  var Name : String . var S : CppState . var O : TokenSequence .
  eq preprocess(Name) = returnOutput(state(readFile(Name), outputStream(nil))) .
  eq returnOutput(outputStream(O), S) = O .
endfm
```

Figure 3.4: High-level view of Cpp's behavior

The preprocess operation is the external interface of Cpp, as described above. It returns the outputStream at the end of processing by applying the operation returnOutput to the final state. The operation readFile is defined in module HELPING-OPS in Appendix A.

3.3 File Inclusion Directive

The `#include` directive allows programmers to divide the program into smaller, more manageable parts, and tie common declarations together [22]. Included files are usually called header files. The name of the file to be included is denoted after the `#include` keyword with one of three possible forms: "filename" (which usually denotes a header file in the same package or subsystem), <filename> (to refer to library or standard implementation files) or a macro call that expands to one of the previous forms (the latter are called "computed includes" [25]). Figure 3.5 shows the Maude specification of the syntax of `#include`. The module MACRO-CALL-SYNTAX can be found in Appendix A.

```
fmod INCLUDE-SYNTAX is
  ex CPP-DIR-SYNTAX .
  pr IDENTIFIER . pr MACRO-CALL-SYNTAX .
  sorts IncludeDir FileName .
  subsort IncludeDir < CppDirective .
  op <_> : Identifier -> FileName .
  op #include_cr : FileName -> IncludeDir .
  op #include_cr : String -> IncludeDir .
  op #include_cr : MacroCall -> IncludeDir .
endfm
```

Figure 3.5: Syntax of the #include directive

The `#include` directive causes Cpp to process the contents of the specified file before continuing with the rest of the current file, as if those contents had appeared in place of the `#include` directive. Cpp appends the output resulting from the included file to the output already generated and then appends the output that comes from the text after this directive [25; 26].

Cpp also accepts as input the directories where it should search for included files. These are called "include directories". This requires us to change the external interface of the preprocess operation to accept a list of include directories as input (in module CPP-SEMANTICS), and it also requires a new CppStateAttribute to store these directories (in module CPP-STATE). The new versions of these modules appear in Figure 3.6.

```
fmod CPP-STATE is
  ...
  pr STRINGS .  --- defines StringSet
  op includeDirs : StringSet -> CppStateAttribute .
endfm

fmod CPP-SEMANTICS is pr CPP-SYNTAX .
  pr HELPING-OPS . ex LINE-SEQ-SEMANTICS .
  op preprocess : String StringSet -> TokenSequence .
  op returnOutput : CppState -> TokenSequence .
  var Name : String . var S : CppState . var IDirs : StringSet . var O : TokenSequence .
  eq preprocess(Name, IDirs)
       = returnOutput(state(readFile(Name), (includeDirs(IDirs), outputStream(nil)))) .
  eq returnOutput(outputStream(O), S) = O .
endfm
```

Figure 3.6: Addition of include directories to CPP-STATE and CPP-SEMANTICS

Figure 3.7 shows the semantics of the `#include` directive in the case that the file

name is specified as a String. The other cases are very similar (see Appendix A). The behavior in this case is to leave the outputStream unmodified (which is not even shown in the equation but is part of the state S), read in the lines of the included file using the operation readFile and apply the operation state on those lines and on the current state. The operation readFile is defined in module HELPING-OPS (see Appendix A) and simulates reading a file into memory. Module HELPING-OPS actually adds a couple of equations that feed some example lines to the operation readFile for a simple test case.

```
fmod INCLUDE-SEMANTICS is pr INCLUDE-SYNTAX .
  ex CPP-DIR-SEMANTICS . pr HELPING-OPS .
  var FN : String . var S : CppState . var SS : StringSet .
  eq state(#include FN cr, (includeDirs(SS), S))
       = state(readFile(FN, SS), (includeDirs(SS), S)) .
endfm
```

Figure 3.7: Semantics of the #include directive

If a given file is included more than once in a compilation unit, Cpp processes the file completely each time. The reason is that macros may have changed from the previous inclusion of the file.

3.4 Macro Definition Directive

The `#define` directive is used to define macros. A macro associates a name with an arbitrary fragment of C code [25]. The name of a macro may be any valid identifier. The replacement text ends with an end of line that has not been preceded by a backslash character, and cannot contain another Cpp directive.

The scope of a macro definition starts right after its `#define` and ends with the compilation unit. A macro may be undefined through the `#undef` directive followed by the macro name. This directive reduces the scope of the macro.

Macros with parameters are called *function-like macros*. A function-like macro is invoked by writing its name, a left parenthesis, the actual argument list as comma-

separated sequences of expressions and a right parenthesis. Figure 3.8 shows the syntax of the `#define` and `#undef` directives.

```
fmod DEFINE-SYNTAX is
  ex CPP-DIR-SYNTAX . pr TOKEN .
  sorts MacroDefDir MacroUndefDir .
  subsort MacroDefDir MacroUndefDir < CppDirective .
  op #define__cr : Identifier TokenSequence -> MacroDefDir .
  op #define_‘(_‘)_cr : Identifier IdentifierList TokenSequence -> MacroDefDir .
  op #undef_cr : Identifier -> MacroUndefDir .
endfm
```

Figure 3.8: Syntax of the #define directive

When Cpp encounters a `#define` directive, it creates an entry in a macro table that associates the given name with its replacement text. Figure 3.9 shows the specification for macro definitions (module MACRO-DEF) and for the macro table (module MACRO-TABLE). The figure only presents the signatures of these modules but the complete specification can be found in Appendix A.

Module MACRO-DEF defines sort MacroDef to represent macro definitions, with two constructors for them, with or without parameters. This module will be extended when the semantics of macro calls is described, with operations that return the macro expansion. Module MACRO-TABLE defines sort MacroTable as a set of [Identifier : MacroDef], which represents a pair [macro name, macro definition].

The macro table has to be added to the CppState, since it will get built and used during preprocessing. The CppStateAttribute added for that purpose is called macroTbl. Since Cpp also accepts command line macros as input, the macroTbl attribute is initialized with those macros (see Appendix A).

Upon a `#define` directive, Cpp creates a MacroDef and appends it to the macro table. Upon a `#undef` directive, Cpp removes the MacroDef for it from the macro table. Figure 3.10 presents the module DEFINE-SEMANTICS, with the behavior of Cpp upon a macro definition without parameters (first equation), with parameters (second equation) and upon the un-definition of a macro (third and fourth equations). The module has one conditional equation, specified with the keyword ceq. Conditional

```
fmod MACRO-DEF is
  pr TOKEN .
  sort MacroDef .
  op name_replText_ : Identifier TokenSequence -> MacroDef .
  op name_params_replText_ : Identifier IdentifierListp TokenSequence -> MacroDef .
  op name : MacroDef -> Identifier .
  op hasArgs : MacroDef -> Bool .
endfm

fmod MACRO-TABLE is
  pr MACRO-DEF .
  sort MacroTable .
  op empty : -> MacroTable .
  op [_:_] : Identifier MacroDef -> MacroTable .
  op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
  op _[_] : MacroTable Identifier -> MacroDef .
  op _[_<-_] : MacroTable Identifier MacroDef -> MacroTable .
  op isMacro : Identifier MacroTable -> Bool .
  op isMacroWithArgs : Identifier MacroTable -> Bool .
  op isMacroWithoutArgs : Identifier MacroTable -> Bool .
  op remove : MacroDef MacroTable -> MacroTable .
endfm
```

Figure 3.9: Specification of macro definitions and the macro table

equations are only applied if the condition they specify is true. In this module, a symbol is removed from the macro table with `#undef` only if the symbol was previously defined as a macro. If the condition of the conditional equation is not true, the following equation in the module will be applied, since it has the attribute `[owise]`, meaning "otherwise", i.e., if the condition of the previous equation is false, apply the current equation.

```
fmod DEFINE-SEMANTICS is pr DEFINE-SYNTAX .
  ex CPP-DIR-SEMANTICS .
  var I : Identifier . var TS : TokenSequence . var MT : MacroTable .
  var S : CppState . var IdL : IdentifierList .
  eq state(#define I TS cr, (macroTbl(MT), S))
    = macroTbl([I : (name I replText TS)] MT), S .
  eq state(#define I ( IdL ) TS cr, (macroTbl(MT), S))
    = macroTbl([I : (name I params (IdL) replText TS)] MT), S .
  ceq state(#undef I cr, (macroTbl(MT), S))
    = macroTbl(remove(MT[I], MT)), S if isMacro(I, MT) .
  eq state(#undef I cr, (macroTbl(MT), S)) = macroTbl(MT), S [owise] .
endfm
```

Figure 3.10: Semantics of the #define directive

While Cpp tokenizes each input line and upon each identifier, it searches its macro table for a macro with that name. If it finds the name in the macro table, and if the macro does not reference itself directly or indirectly, Cpp replaces the identifier

(and arguments if it has) by the replacement text after the appropriate argument substitution. Figure 3.11 shows the additional equations of operation state in module LINE-SEQ-SEMANTICS, which handle macro calls in the TokenSequence of a line. There are three CppStateAttributes involved in these equations. Two of them have been already introduced: the current macro table (macroTbl), to check if the next token is the name of a macro, and the outputStream. A new state attribute, curMacroCalls, is needed to check if a macro references itself directly or indirectly, in which case the inner reference is not macro expanded. The attribute curMacroCalls is a list of macro names (an IdentifierList) whose expansion is currently being processed. Moreover, some way is needed to remove a macro name from curMacroCalls when Cpp finishes processing its expansion. For this purpose the specification uses the token `'##` (which can never occur in this context) to mark the end of a macro expansion. Therefore, when the next token in the input is a `'##` (first and second equations), the behavior is to remove the top of curMacroCalls. When a macro call is expanded (fourth and fifth equations), a `'##` token is placed at the end of the expansion and the macro name is inserted at the beginning of curMacroCalls.

```
fmod LINE-SEQ-SEMANTICS is ...
  vars TS O AS : TokenSequence . var I : Identifier . var IL : IdentifierList . var T : Token .
  var S : CppState . var MT : MacroTable . var MC : MacroCall .

  eq state(('## TS) cr, (curMacroCalls( (I, IL) ), S))
    = state(TS cr, (curMacroCalls( (IL) ), S)) .
  eq state(('## TS) cr, (curMacroCalls( (I) ), S)) = state(TS cr, (curMacroCalls( () ), S)) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), outputStream(O), S))
    = state(TS cr, (macroTbl(MT), curMacroCalls(ILP), outputStream(O T), S))
    if not(isMacro(T, MT)) or (T in ILP) .
  ceq state((T '`( AS '`) TS) cr, (macroTbl(MT), curMacroCalls(ILP), S))
    = state((expandWithTSArgs(MT[T], toTokenSeqList(AS)) '## TS) cr,
        (macroTbl(MT), curMacroCalls(cons(T, ILP)), S))
    if isMacroWithArgs(T, MT) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), S))
    = state((expand(MT[T]) '## TS) cr, (macroTbl(MT), curMacroCalls(cons(T, ILP)), S))
    if isMacroWithoutArgs(T, MT) .
endfm
```

Figure 3.11: Extension of module LINE-SEQ-SEMANTICS to handle macro calls

When a call to a function-like macro is encountered, the token sequence that represents the arguments (`AS` in Fig. 3.11) needs to be separated into individual

arguments. For this purpose, the operation toTokenSeqList is applied to convert AS into a TokenSeqList, which is a list of ';' separated TokenSequences, one for each argument. Module TOKEN-TO-ARG (see Appendix A) defines sort TokenSeqList and the operation toTokenSeqList.

While it is easier to read module LINE-SEQ-SEMANTICS from the bottom up, the current order of equations dictates the correct precedence with which equations should be matched by Maude's matching algorithm.

The replacement text of a macro may call other macros and for this reason Cpp needs to repeatedly re-scan the macro expansion for more instances of macros to expand. That is the reason why the macro expansion is inserted at the front of the input and not directly appended to the output stream. Similarly, macro arguments may also contain macro calls, but these calls are expanded only after the containing macro has been expanded. In this sense, the semantics of macro expansion can be compared to "call-by-name" parameter passing style. Note that this semantics follows the ANSI-C convention for macro expansion [26]. In contrast with this, GCC's version of Cpp expands macro calls in arguments before they are substituted into the macro body [25].

The following describes the semantics of macro expansion, to define the operations expand and expandWithTSArgs in module MACRO-DEF. Cpp macro substitution is quite complex and may lead to unexpected results. See [26] and [25] for a complete list of rules on macro substitution and common pitfalls.

The replacement text of macros may use two special operators: the stringification operator '#' and the concatenation operator '##'. When a macro parameter is immediately preceded by '#', Cpp converts the parameter name into a string constant. When a macro body contains a '##' operator, Cpp pastes together or concatenates the tokens surrounding the '##' token. Figure 3.12 shows the equations in module MACRO-DEF that implement macro expansion.

```
fmod MACRO-DEF is
  ...
  op expand : MacroDef -> TokenSequence .
  op expandWithTSArgs : MacroDef TokenSeqList -> TokenSequence .
  op ex-recTS : IdentifierListP TokenSequence TokenSeqList -> TokenSequence .
  op dquote : -> Qid .

  var N : Identifier . var TS : TokenSequence . vars T T2 : Token .
  var PL : IdentifierList . var TSL : TokenSeqList .

  ceq expandWithTSArgs(name N params PL replText TS, TSL) = nil if (size(PL)  =/= size(TSL)) .
  eq expandWithTSArgs(name N params PL replText TS, TSL) = ex-recTS(PL, TS, TSL) [owise] .
  eq ex-recTS(PL, nil, TSL) = nil .
  eq ex-recTS(PL, '# T TS, TSL) = dquote elemAtTS(pos(T, PL), TSL) dquote
                        ex-recTS(PL, TS, TSL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(T) + string(T2)) ex-recTS(PL, TS, TSL)
    if not(T in PL) and not(T2 in PL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(elemAtTS(pos(T, PL), TSL)) + string(T2))
        ex-recTS(PL, TS, TSL)
    if (T in PL) and not(T2 in PL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(T) + string(elemAtTS(pos(T2, PL), TSL)))
        ex-recTS(PL, TS, TSL)
    if not(T in PL) and (T2 in PL) .
  eq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(elemAtTS(pos(T, PL), TSL)) +
        string(elemAtTS(pos(T2, PL), TSL))) ex-recTS(PL, TS, TSL) [owise] .
  ceq ex-recTS(PL, T TS, TSL) = T ex-recTS(PL, TS, TSL) if not(T in PL) .
  ceq ex-recTS(PL, T TS, TSL) = elemAtTS(pos(T, PL), TSL) ex-recTS(PL, TS, TSL) if (T in PL) .
endfm
```

Figure 3.12: Semantics of macro expansion

The operation expandWithTSArgs uses the auxiliary operation ex-recTS, a recursive operation that traverses all tokens in the body of the macro, replacing formal parameters by arguments. If the macro body is empty (nil) the expansion is the nil token sequence (first equation of ex-recTS). A '# in front of a token "stringifies" the token, using the constructor dquote that represents a double quote (second equation of ex-recTS). The next four equations deal with token concatenation, for every combination where the involved tokens are or are not parameters of the macro. These equations use the concatenation operator provided for Strings (+), so the involved tokens are first transformed into Strings (using the operation string), concatenated with + and then converted into tokens again (by using the operation qid). When stringification or concatenation are not involved and the first token is not a parameter (previous to last equation), the token is just returned followed by the expansion of the rest of the tokens in the macro body. When the next token is a parameter (last equation), it returns the corresponding argument, followed by the rest of the expan-

sion. To obtain the corresponding argument, the operation pos (defined in module IDENTIFIER) is used to get the position of the parameter in the formal parameter list, while the operation elemAtTS (defined in module TOKEN-TO-ARG) returns the argument at that position from the list of arguments.

3.4.1 Differences between Cpp macros and C functions

It is useful to analyze the differences between a function-like macro and a C function. Although this may not be a complete list, we have found the following interesting differences that should be considered during refactoring. The function definition `f1` and its macro counterpart `M1`, which appear in Figure 3.13, will help as running examples in this section.

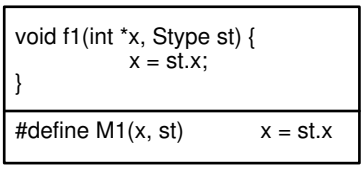

Figure 3.13: A function definition and its macro counterpart

Semantics of parentheses. If a function name appears without being followed by parentheses, like in '`a = f1;`', the semantics are those of taking the address of the function. If a reference to a function-like macro appears without subsequent parentheses, it does not represent a call to the macro nor its address, it is just left alone as an identifier.

Inlining. Macros are inlined at the place of the call at compile time, which does not occur with functions.

Scoping Rules. All C functions are global to the compilation unit in which they are defined. Macros are also global to the compilation unit but its scope may be restricted by the presence of `#undef` directives in their scope. Moreover, a

function definition may be exported to a different compilation unit through the linker by using the C language keyword `extern`. Macros cannot be exported in this fashion because Cpp does not recognize the `extern` keyword.

Parameter typing. In a function definition, all parameters need to be typed, as we see in Figure 3.13, and typing rules are strictly enforced at compile time. Therefore, if the same functionality is needed for different types of arguments, different functions must be defined. In the case of macros, parameters are not typed, so the same macro definition may be used for different types. In this context, macros are similar to C++ templates.

Parameter replacement. When a macro is expanded, every occurrence of a parameter name in the macro body is replaced by its actual argument, no matter whether the name has a different use. In the example of Figure 3.13, the two occurrences of '`x`' in the body of `M1` are replaced by the first argument of calls to `M1`, even when in the syntax of C, the second '`x`' refers to a structure field. Both occurrences of '`x`' are the same for the preprocessor. This does not happen in the case of functions, which are parsed according to C syntax. In the example, the second occurrence of '`x`' in the body of `f1` will take the value of the structure field instead of the value of parameter '`x`'.

Argument evaluation. The C language supports *call-by-value* and *call-by-reference* parameter passing styles. In the case of call-by-value, function arguments are evaluated right before the actual call takes place, and the resulting value of evaluating each argument is passed to the function call. Instead, macro arguments are first replaced literally in the macro body at compile time and their evaluation is delayed until they are used at run-time. Consequently, macro arguments are evaluated each time they occur in the macro body instead of just one time as in function calls. This semantics can be compared to *call-by-name*

parameter passing, in which argument evaluation is delayed until their values are needed [64]. For this reason, parentheses surrounding macro parameters in macro bodies become very important, as each parameter may be replaced by a complicated sequence of tokens instead of by a single value as with functions.

Self-reference. If a function has a reference to itself in its definition, the function is considered recursive and it will call itself every time the self-reference is encountered. In the case of a macro, a self-reference in the macro body is not recursively expanded, i.e., the self-reference does not constitute a macro call.

3.5 Conditional Compilation Directives

Conditional compilation directives define separate code branches, which are included or excluded from the final compilation unit depending on the value of conditions evaluated by `Cpp` [22].

Conditional directives are mainly used to configure a program for different platforms. A *configuration* can be defined as the initial value of macros (a.k.a. configuration variables) that Cpp receives as input to preprocess a program. With this, a configuration determines which *single* branch of each preprocessor conditional will be present in the output of Cpp. Others consider this output of Cpp, the preprocessed code, to be a configuration. The definitions are isomorphic but we generally refer to the first.

Most projects written in C are highly configurable. For example, Flex [65] has less than 20K lines of code among 21 files, but has 5 configuration variables that make up a space of 2^5 possible configurations. The Linux kernel (version 2.6.7) has about 1,672 configuration variables with binary value. The number of possible configurations is huge.

Conditional directives are also used to have a separate version of the program

with debugging or testing code, for commenting, if the condition is always false and they are also used in conjunction with macros as "include guards", i.e., to prevent multiple inclusions of the same file.

A conditional directive is one of the following: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` or `#endif`. The `#if`, `#ifdef` and `#ifndef` directives start a *Cpp conditional* construct, creating its first branch. The `#elif` and `#else` directives create additional branches on the Cpp conditional and the `#endif` ends the construct. The source text inside a branch may include other preprocessor directives. Consequently, Cpp conditionals can also be nested. Figure 3.14 shows an example taken from the file "`compiler.h`" in the Linux kernel (version 2.6.7).

```
#ifndef __ASSEMBLY__
#if __GNUC__ > 3
# include <linux/compiler-gcc+.h>
#elif __GNUC__ == 3
# include <linux/compiler-gcc3.h>
#elif __GNUC__ == 2
# include <linux/compiler-gcc2.h>
#else
# error Sorry, your compiler is too old/not recognized.
#endif
#endif
```

Figure 3.14: Example of the use of conditional directives

The `#if` and `#elif` tokens are followed by a constant expression that can only be composed of the following:

- integer or character constants;
- arithmetic and logical operators, except for assignments, sizeof, comma, increment or decrement operators or casts expressions;
- macros;
- "`defined` *id*" or "`defined` (*id*)" expressions, which evaluate to 1 if *id* has been defined as a macro or 0 otherwise;

- identifiers that are not macros, which are replaced by 0 after macro expansion and cannot be function calls.

The lines "`#ifdef` *id*" and "`#ifndef` *id*" are abbreviations of "`#if defined` *id*" and "`#if !(defined` *id*`)`" respectively.

Figure 3.15 shows the specification of the syntax of conditional directives. Module COND-EXP-SYNTAX defines sort CondExp, which represents the condition that follows `#if` and `#elif`. Note that the operation `e` had to be defined as a wrapper over integers so we could modify the attributes of the operations defined on CondExp values. The complete list of operations allowed for sort CondExp can be found in Appendix A. Module COND-DIR-SYNTAX defines sort CondDir to represent conditional directives.

```
fmod COND-EXP-SYNTAX is
  pr IDENTIFIER . pr INT .
  sort CondExp .
  subsort Identifier < CondExp .
  op e : Int -> CondExp .
endfm

fmod COND-DIR-SYNTAX is ex CPP-DIR-SYNTAX .
  pr ALL-COND-EXP-SYNTAX .
  sort CondDir .
  subsort CondDir < CppDirective .
  op #if_cr : CondExp -> CondDir .
  op #ifdef_cr : Identifier -> CondDir .
  op #ifndef_cr : Identifier -> CondDir .
  op #elif_cr : CondExp -> CondDir .
  op #else‘cr : -> CondDir .
  op #endif‘cr : -> CondDir .
endfm
```

Figure 3.15: Syntax of conditional directives

Upon a Cpp conditional, Cpp evaluates the expressions after the `#if`, `#ifdef`, `#ifndef` and `#elif` directives until the first one is found to be true. It then passes on the text inside that branch to the output and discards the previous and the remaining conditional branches. If the value of all expressions is false and there is a `#else` line, the text inside the `#else` branch is the one passed on to the output.

To specify this semantics, three new state attributes must be added: skip is a boolean value to distinguish when Cpp is scanning the true branch of a conditional

from the state in which Cpp is *skipping* the tokens in a false branch; nestLevelOf-Skipped is the depth of nesting of conditionals that are being "skipped"; branchTaken is another boolean value that tells whether a branch from the current conditional has already been taken, in which case all the other branches will be skipped disregarding their conditions. Figure 3.16 shows the final version of module CPP-STATE will all necessary CppStateAttributes.

```
fmod CPP-STATE is
  pr MACRO-TABLE . pr TOKEN . pr STRINGS .
  sorts CppState CppStateAttribute .
  subsort CppStateAttribute < CppState .
  op empty : -> CppState .
  op _,_ : CppState CppState  -> CppState [assoc comm id: empty] .

  op includeDirs : StringSet -> CppStateAttribute .
  op macroTbl : MacroTable -> CppStateAttribute .
  op curMacroCalls : IdentifierList -> CppStateAttribute .
  op skip : Bool -> CppStateAttribute .
  op nestLevelOfSkipped : Nat -> CppStateAttribute .
  op branchTaken : Bool -> CppStateAttribute .
  op outputStream : TokenSequence -> CppStateAttribute .
endfm
```

Figure 3.16: Final version of CPP-STATE

Module LINE-SEQ-SEMANTICS has to be updated once again to process lines and tokens only when the state attribute skip is not true. See Appendix A for the final version.

Figure 3.17 shows module COND-DIR-SEMANTICS with the behavior of Cpp on the conditional directives `#if`, `#else` and `#endif` (the behavior of `#elif` is similar to `#else` and can be found in the extended version of this module in Appendix A). There are three cases of the operation state for each conditional directive. For example, in the first equation, when Cpp is "not skipping" the tokens in a false branch, and a `#if` directive is found with a condition that evaluates to true, Cpp continues in "not skipping" mode (i.e., the value of skip continues to be false) but the value of the state attribute branchTaken turns true, meaning, a branch in the current conditional (the one starting with this directive) has been taken. As another case, in the third equation, when Cpp is "skipping", meaning, is in the false branch of a conditional,

the appearance of a `#if` maintains Cpp in skipping mode, no matter what condition follows the `#if`. The value of the attribute nestLevelOfSkipped is incremented because the depth of nesting of conditionals being "skipped" turns one more. The operation evalB is used to evaluate the truth value of the expression in a conditional directive, given the current macro table. This operation is defined in module COND-EXP-SEMANTICS (see Appendix A).

```
fmod COND-DIR-SEMANTICS is pr COND-DIR-SYNTAX .
  ex CPP-DIR-SEMANTICS .
  pr ALL-COND-EXP-SEMANTICS .
  var CE : CondExp . var N : Nat . var B : Bool . var AMT : MacroTable . var S : CppState .

  --- Case 1 of #if: Not skipping -> Not skipping
  ceq state(#if CE cr, (macroTbl(AMT), skip(false), branchTaken(false), S))
    = macroTbl(AMT), skip(false), branchTaken(true), S  if evalB(CE, AMT) = true .
  --- Case 2 of #if: Not skipping -> Skipping
  ceq state(#if CE cr, (macroTbl(AMT), skip(false), nestLevelOfSkipped(0), branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1), branchTaken(false), S
    if evalB(CE, AMT) = false .
  --- Case 3 of #if: Skipping -> Skipping
  eq state(#if CE cr, (skip(true), nestLevelOfSkipped(N), branchTaken(B), S))
    = skip(true), nestLevelOfSkipped(N + 1), branchTaken(false), S .

  --- Case 1 of #else: Not skipping -> Skipping
  eq state(#else‘cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S .
  --- Case 2 of #else: Skipping -> Skipping
  eq state(#else‘cr, (skip(true), nestLevelOfSkipped(N), branchTaken(true), S))
    = skip(true), nestLevelOfSkipped(N), branchTaken(true), S .
  --- Case 3 of #else: Skipping -> Not skipping
  eq state(#else‘cr, (skip(true), nestLevelOfSkipped(1), branchTaken(false), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(true), S .

  --- Case 1 of #endif: Not skipping -> Not skipping
  eq state(#endif‘cr, (skip(false), branchTaken(true), S))
    = skip(false), branchTaken(false), S .
  --- Case 2 of #endif: Skipping -> Skipping
  ceq state(#endif‘cr, (skip(true), nestLevelOfSkipped(N), S))
    = skip(true), nestLevelOfSkipped(N - 1), S   if N > 1 .
  --- Case 3 of #endif: Skipping -> Not Skipping
  eq state(#endif‘cr, (skip(true), nestLevelOfSkipped(1), branchTaken(true), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(false), S .
endfm
```

Figure 3.17: Semantics of conditional directives

Chapter 4

Pseudo-Preprocessing in CRefactory

The previous chapter described how Cpp evaluates and removes preprocessor directives. The preprocessing step is necessary to be able to compile a program and execute it, since Cpp directives are not part of the C language. However, preprocessing a program makes the results be specific to a single configuration and loses information about macros and file dependencies. Although this problem has been recognized and various approaches have been explored, none of the approaches are a complete solution.

Refactoring requires a complete solution to this problem. A refactoring tool cannot use Cpp for two main reasons:

1. The preprocessed version of the code may be unmanageable for a user to visualize and change. Imagine a program conveniently divided into 100 files, which Cpp would merge into a single piece, with only the code for a single configuration and all macros expanded.

2. If changes are applied to the preprocessed version of a program, it may be impossible to recover the corresponding un-preprocessed version (which would

mean the developer would be stuck with a single file instead of having the 100 pieces, working for a single configuration, with no macros in it) [8].

Fig. 4.1 shows an example of un-preprocessed source code on the left and the result of preprocessing it on the right. If the variable `errStatus` is renamed to `error`, it is not possible to translate the right-hand side back into the un-preprocessed source code.

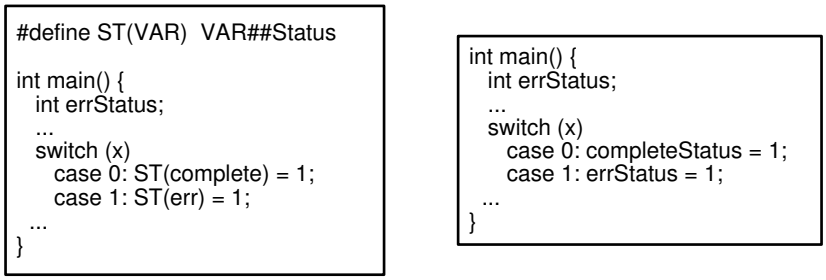

Figure 4.1: A source code and its preprocessed version. The macro 'ST' receives one parameter and applies concatenation of its parameter with the string 'Status'

Moreover, changing the code once it has been targeted to a specific configuration isolates that code from all the rest: if the changed code is merged back, the source code for other configurations may not compile anymore, or the behavior may be altered.

Therefore, it is not acceptable to have the refactoring tool work on a single configuration of a program, nor to lose any Cpp directives. A refactoring tool must ensure that program behavior is preserved for all possible configurations.

4.1 The Need for Pseudo-Preprocessing

One solution would be to avoid preprocessing at all and have the parser and semantic analyzer deal directly with Cpp directives. However, it is not possible to apply this solution without introducing ambiguities in the grammar or being too restrictive. Conditional compilation directives and macro calls often "break" statements.

A conditional compilation directive or macro call breaks a statement when it produces a fragment of C code that is not a complete syntactic unit. Figure 4.2 shows a case where the preprocessor conditional produces the start of an if-statement and its condition, but not its compound statement. In another example the preprocessor conditional could produce half the condition, or the last half of the condition and the first half of the compound statement, or only the compound statement. In theory, the C grammar could be extended to handle all these situations, but the resulting grammar would be very large and ambiguous.

```
  dep->changed = !dir_file_exists_p (name, "");
#ifdef VMS
  if (dep->changed && strchr (name, ':') != 0)
#else
  if (dep->changed && *name == '/')
#endif
  {
      freerule (rule, lastrule);
      ...
  }
```

Figure 4.2: Conditional directives breaking a statement. This piece of code was extracted from file "rule.c" in the source code of make-3.80

Macros cause a similar problem and are harder to fix by changing the grammar, because a macro may represent any arbitrary fragment of C code. Fig. 4.3 shows an example where parsing cannot proceed without making various assumptions. `ERROR_EXIT` is a function-like macro. Without expanding `ERROR_EXIT` at its calls, the parser cannot answer: is the third `if`-statement inside the second `if`? Is there an `else` branch for the second `if` which is not closed and so includes the third `if`? Is the last '`}`' token closing something that was opened inside `ERROR_EXIT`?

Since we do not want CRefactory to be too restrictive in the kind of macros it accepts or the placement of directives, CRefactory cannot parse the input directly. Instead, CRefactory needs to "partially" preprocess the input, in a way that does not remove directives but makes the input parseable. We call this "**pseudo-preprocessing**". We have developed a pseudo-preprocessor, called **P-Cpp**, which

```
if ((fp_target = fopen(ptarget, "r")) != NULL)
{
    fgets(old_line, buffer_size, fp_target);
    if (fclose(fp_target) != 0)
            ERROR_EXIT(ptarget);
    if (!strcmp(line, old_line))
            is_same = 1;
}
```

Figure 4.3: A macro call that may prevent correct parsing. This piece of code was extracted from file "split-include.c" in linux-2.6.7.

tokenizes the input and executes directives as Cpp does, but does not remove directives from the tokenized output. Instead, it constructs tokens and other representations of Cpp directives. P-Cpp does not merge included files but tokenizes them separately and creates a representation of file dependencies. It expands macro calls but labels the tokens in the expansion so that the call can be traced back and reconstructed. Moreover, P-Cpp places conditional directives in the tokenized output so that they do not break syntactical units of the C grammar, labelling the tokens so the original form can be reconstructed.

The next section describes the input of P-Cpp. The following sections explain the behavior of P-Cpp with each of the directives: `#include`, `#define` and conditional directives. We use Maude in this chapter not to specify the full behavior of P-Cpp, but to precisely describe some of the important differences between the semantics of P-Cpp and Cpp, given the same Cpp syntax specified in Chapter 3. Appendix B has the Maude snippets in this chapter plus a few more details.

4.2 The Input of P-Cpp

P-Cpp receives as input a CRConfiguration, an object composed of the following fields:

- The names of all source files that compose the program. This differs from Cpp, which receives a single source file as input.

- Include directories (same as Cpp).
- Read-only directories, which contain files that cannot be modified.
- Command line macros (same as Cpp).
- A list of conditions that should be considered always false, which will be described in Section 4.5.
- A list of incompatible conditions, again to be described in Section 4.5.

Figure 4.4 shows the Maude specification of sort CRConfiguration and its components. The directories that are read-only are specified with a '*' as the first character of the directory name in the includeDirs set.

```
fmod CONFIG is
  pr STRINGS .
  sort CRConfiguration .
  op fileNames_includeDirs_commandLineMacros_falseConds_incompatConds_ :
        StringSet StringSet StringSet StringSet StringSet -> CRConfiguration .
endfm
```

Figure 4.4: Specification of sort CRConfiguration. It represents the input of P-Cpp

With respect to the source code that P-Cpp expects as input, P-Cpp does not make any assumptions about the placement of conditional directives in the input lines. However, P-Cpp does assume that `#include` directives appear in between statements or declarations and that `#define` directives only occur in between five possible syntactic constructs, which are listed in Table 4.1. Note that a `#include` or a `#define` may appear inside a compound statement, as long as it does not break any statement inside the compound statement. These are the only places we have found these directives in open source code packages, so we believe this assumption about the kind of input that CRefactory accepts is not too restrictive.

Let *CppInput* be the set of programs that Cpp accepts as input. Then, the programs accepted by P-Cpp are a subset of well-formed programs of *CppInput* called

Table 4.1: Syntactic constructs of the C grammar that macro definitions should not break

Statement
Declaration
Structure field
Enumerator value
Array initializer value

$WFCppInput$. The set $WFCppInput$ excludes C programs that have `#include` breaking statements or declarations and `#define` directives breaking the syntactic construct of Table 4.1. Additionally, $WFCppInput$ excludes C programs with files that start or end with half a statement or have only a part of a Cpp conditional construct, i.e., each file must be parseable on its own.

Moreover, Section 4.5 will show how P-Cpp manipulates conditional directives so that its output conforms to the format of the CRefactory parser (CRParser). The input that CRParser expects is a set $WF_0CppInput$ such that

$$CppInput \supset WFCppInput \supset WF_0CppInput$$

The set $WF_0CppInput$ excludes C programs that have any Cpp directives breaking the syntactic constructs of Table 4.1. Moreover, the source code of programs in $WF_0CppInput$ do not have macro calls, so it is also the job of P-Cpp to expand macros. Appendix C shows the complete extended grammar accepted by CRParser. Note that this grammar also supports some GCC extensions like assembler instructions and statement expressions [66].

4.3 Handling File Inclusion

The behavior of Cpp with `#include` directives is to merge all files into a single compilation unit, i.e., Cpp produces a single stream of tokens, no matter how many files compose the given program. This behavior is not appropriate for a refactoring

tool. Programmers divide programs into several files to increase readability, ease maintenance and increase code reuse. Merging all files into a single unit defeats all those good reasons, which are also the goals of refactoring.

P-Cpp produces a *separate* output stream of tokens for each file. Representing each file separately eases analysis, transformation and pretty-printing. The representation of a file is called ProgramFile. Since the source code that appears after the `#include` line depends on the declarations of the file being included, it is still necessary to process the included file before continuing with the rest of the code in the current file, just as Cpp would. For this purpose, P-Cpp maintains a *stack* of ProgramFiles being processed, and pushes a new ProgramFile on the stack as it finds `#include` directives. The output tokens that P-Cpp creates are appended to the output stream of the ProgramFile at the top of the stack.

P-Cpp also maintains a stack of the source code of each file whose processing has been paused by a file inclusion directive. The elements in the input stack correspond one-to-one with the elements in the stack of ProgramFiles, except for the ProgramFile at the top whose line sequence is currently being processed (and so has not been stacked).

Since each file is represented separately, it is necessary to model the relationships or dependencies among files. The best representation of file dependencies is a graph, which we call **Include Dependencies Graph** (**IDG**). The nodes in this graph are ProgramFiles and the edges represent `#include` dependencies. Specifically, there is an edge in the graph from Program File 'A' to Program File 'B' if 'B' includes 'A'.

The exact position at which the `#include` directive occurs is important when calculating the definitions reaching a certain line of code. Moreover, file inclusion may be conditional, if the `#include` directive occurs inside a Cpp conditional. Therefore, edges in the IDG are labelled with the position at which the file is included and the condition under which it is included. Fig. 4.5 shows a simple example of an IDG.

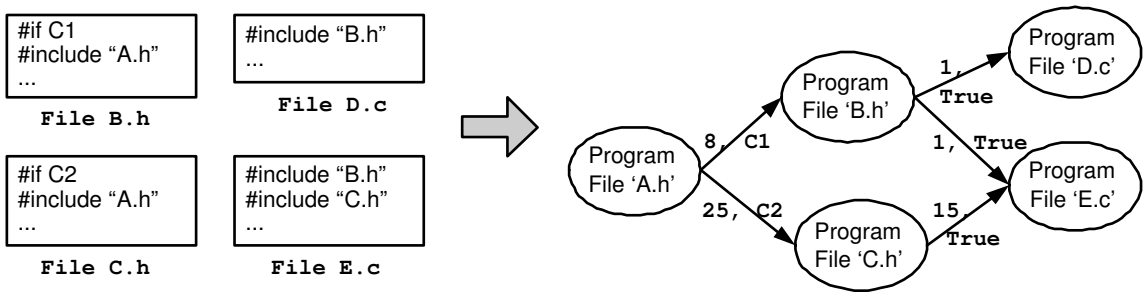

Figure 4.5: Example of Include Dependencies Graph

The example in Figure 4.5 shows that file "E.c" indirectly includes two copies of file "A.h". This is common, and Chapter 3 explained that the behavior of Cpp is to preprocess the file every time it is included. On the contrary, P-Cpp does not need to process a file more than once. Upon a `#include` directive, P-Cpp checks if the file is already in the IDG. If it is not, P-Cpp adds a node in the IDG for the file and the corresponding edge, processes the file and builds a representation of that file that is stored in the file's node. If the file has been previously included, it is not necessary to process it again but P-Cpp can reuse the representations already generated. Specifically, it will gather all the macros that have been defined in that file and the files it includes (which are its predecessors in the IDG). Therefore, in the case of file inclusion, P-Cpp works more efficiently than Cpp reusing previously generated representations. Section 4.6 will show how P-Cpp reuses representations, after macros and conditional directives are discussed.

The formal specification of how P-Cpp handles file inclusion starts in Figure 4.6. The figure shows module LOC specifying Locations, module CR-TOKEN and module INCLUDE-DEP-GRAPH. P-Cpp represents Locations in the source code as a pair: the file name and the character offset in that file. Module CR-TOKEN defines CRToken and CRTokenStream. Output tokens created by P-Cpp are of sort CRToken and streams of CRTokens are of sort CRTokenStream. Subsequent sections will show how CRTokens get labelled with macro calls and conditions. The module

CONDITIONS will be described in detail in Section 4.5. It defines the sort CppCondition that represents a condition associated with enclosing Cpp conditionals. Module INCLUDE-DEP-GRAPH defines the sorts: ProgramFile, ProgramFileStack, IncludeDepGraph, IdgEdge and IdgEdgeList. A ProgramFile is represented by the name of the file, the output token stream that P-Cpp creates to represent it, the edges to the files that include it directly (successor edges) and the edges to the files it includes (predecessor edges). Subsequent sections in this chapter will add other elements to the representation of a ProgramFile.

```
fmod LOC is
  pr STRING . pr NAT .
  sort Location .
  op nilLoc : -> Location .
  op file_offset_ : String Nat -> Location .
endfm

fmod CR-TOKEN is pr TOKEN .
  pr MACRO-DEF . pr CONDITIONS .
  sort CRToken . sort CRTokenStream .
  subsort CRToken < CRTokenStream .
  op value_ : Token -> CRToken .
  op value_macroCalls_cond_ : Token MacroCallStack CppCondition -> CRToken .
  op empty : -> CRTokenStream .
  op __ : CRTokenStream CRTokenStream -> CRTokenStream [assoc id: empty] .
endfm

fmod INCLUDE-DEP-GRAPH is
  pr CR-TOKEN . pr CONDITIONS . pr LOC .
  sorts ProgramFile ProgramFileStack IncludeDepGraph IdgEdge IdgEdgeList .
  subsort ProgramFile < IncludeDepGraph .
  subsort ProgramFile < ProgramFileStack .
  subsort IdgEdge < IdgEdgeList .
  op empty : -> IncludeDepGraph .
  op __ : IncludeDepGraph IncludeDepGraph -> IncludeDepGraph [assoc comm id: empty] .
  op nil : -> ProgramFileStack .
  op _;_ : ProgramFileStack ProgramFileStack -> ProgramFileStack [assoc id: nil] .
  op dest_pos_under_ : ProgramFile Location CppCondition -> IdgEdge .
  op nil : -> IdgEdgeList .
  op _,_ : IdgEdgeList IdgEdgeList -> IdgEdgeList [assoc id: nil] .
  op name_tokenStream_includingFiles_includedFiles_ :
      String CRTokenStream IdgEdgeList IdgEdgeList -> ProgramFile [ctor] .
  op programFile : String -> ProgramFile [ctor] .
  op includes : IncludeDepGraph String -> Bool .
  op addEdgeFrom_to_at_under_ : ProgramFile ProgramFile Location CppCondition -> ProgramFile .
  op name : ProgramFile -> String .
  op appendOutputToken : ProgramFile CRToken -> ProgramFile .
endfm
```

Figure 4.6: Specification of Locations and the Include Dependency Graph.

Like Cpp, P-Cpp's behavior is defined in terms of an operation state, which takes some input and the state of P-Cpp and modifies the state accordingly. The state

of P-Cpp is represented with sort PcppState, which is also a multiset of state attributes of sort PcppStateAttribute. Figure 4.7 shows a preliminary version of module PCPP-STATE with the definition of sorts PcppState and PcppStateAttribute and the operations that represent each attribute. The state attribute represented by inputStack is the stack of source code lines of each file whose processing has been paused by a file inclusion directive. The sort for this stack (LineSeqStack) is defined in module LINE-SEQ-STACK (see Appendix B). The attribute given by fileNames has the names of all source files that compose the program, received as input in the initial CRConfiguration (Fig. 4.4). The attribute represented with includeDirs is the same as in Cpp, stores the include directories received as input parameter. The attribute given by macroTbl is also similar to Cpp. The other new attributes are the Include Dependency Graph (idg), the stack of ProgramFiles being processed (curPF), the stack of current conditions to label IDG edges (curCond) and the current location (curLoc). The specification of P-Cpp does not show how curLoc is updated but it is trivial: the file name is always the name of the ProgramFile at the top of curPF and the offset is incremented as characters are consumed from the input. The sort for attribute curCond will be described in Section 4.5.

```
fmod PCPP-STATE is
  pr LINE-SEQ-STACK . pr INCLUDE-DEP-GRAPH . pr MACRO-TABLE . pr STRINGS .
  pr CONDITIONS .
  sorts PcppState PcppStateAttribute .
  subsort PcppStateAttribute < PcppState .
  op empty : -> PcppState .
  op _,_ : PcppState PcppState  -> PcppState [assoc comm id: empty] .

  op inputStack : LineSeqStack -> PcppStateAttribute .
  op fileNames : StringSet -> PcppStateAttribute .
  op includeDirs : StringSet -> PcppStateAttribute .
  op idg : IncludeDepGraph -> PcppStateAttribute .
  op curPF : ProgramFileStack -> PcppStateAttribute .
  op macroTbl : MacroTable -> PcppStateAttribute .
  op curCond : CondStackStack -> PcppStateAttribute .
  op curLoc : Location -> PcppStateAttribute .
endfm
```

Figure 4.7: State of P-Cpp during preprocessing

Finally, Figure 4.8 shows the semantics of P-Cpp with the file inclusion directive.

Module PCPP-DIR-SEMANTICS shows the signature of the operation state, and module INCLUDE-SEMANTICS has two definitions of that operation. The first is applied when the file to be included has not been processed yet (it is not in the IDG) and the second is for the case that it has been processed.

```
fmod PCPP-DIR-SEMANTICS is pr CPP-DIR-SYNTAX .
  pr PCPP-STATE .
  op state : CppDirective PcppState -> PcppState .
endfm

fmod INCLUDE-SEMANTICS is pr INCLUDE-SYNTAX .
  ex PCPP-DIR-SEMANTICS . pr HELPING-OPS .
  var FN : String . var LS : LineSeq . var LSS : LineSeqStack . var Dirs : StringSet .
  vars PF PF' : ProgramFile . var IDG : IncludeDepGraph . var PFS : ProgramFileStack .
  var LO : Location . var S : PcppState . var MT : MacroTable .
  var CS : CondStack . var CSS : CondStackStack .

  ceq state(#include FN cr LS, (inputStack(LSS), includeDirs(Dirs), idg(PF IDG),
        curPF(PF ; PFS), curCond(CS ; CSS), curLoc(LO), S))
    = state(readFile(FN, Dirs), (inputStack(LS ; LSS), includeDirs(Dirs),
        idg((addEdgeFrom programFile(FN) to PF at LO under condFromStack(CS)) PF IDG),
        curPF(programFile(FN) ; appendOutputToken(PF, value qid("#include" + FN)) ; PFS),
        curCond(nil ; CS ; CSS), curLoc(update(LO)), S))
    if includes(IDG, FN) == false .

  ceq state(#include FN cr, (idg(PF' PF IDG), curPF(PF ; PFS), macroTbl(MT),
        curCond(CS ; CSS), curLoc(LO), S))
    = idg((addEdgeFrom PF' to PF at LO under condFromStack(CS)) PF IDG),
        curPF(appendOutputToken(PF, value qid("#include" + FN)) ; PFS),
        macroTbl(MT macrosDefInPredsOf(PF')), curCond(CS ; CSS), curLoc(LO), S
    if name(PF') == FN .
endfm
```

Figure 4.8: Semantics of P-Cpp with file inclusion

If the file to be included (FN) has not been processed, then its source code is read using readFile, which returns the LineSeq representing the file. The LineSeq of FN will be preprocessed next and the LineSeq LS that followed the #include directive is pushed on top of the inputStack, to be processed after FN. A new node for FN and a new edge from it to the current ProgramFile PF are added to the idg. Although we still cannot describe the state attribute curCond, the reader can infer from Fig. 4.8 that it is implemented as a stack, and each element of this stack corresponds to each ProgramFile in curPF. The element at the top of the stack (CS), which represents the current condition in file PF, is used to label the edge from the node for FN to PF. The program file for FN is pushed on top of the current program file stack (curPF). Finally,

a new token is appended to the output stream of `PF` with the token resulting from the string ("`#include`" + `FN`) as value.

The second state operation in INCLUDE-SEMANTICS takes care of the case that the file to be included is already in the IDG (node `PF'`). The source code of the file is not pushed on the input stack because there is no need to process the source code again (more on this will come in Section 4.6, where this operation is reviewed and refined). A new edge is added from the included file `PF'` to the including file `PF`. The token representing the file inclusion line is appended to the output stream of `PF`. Finally, the macros defined in `PF'` and all the files it includes (its predecessors in the IDG) will be added to the current macro table (macroTbl) by invoking the function macrosDefInPredsOf, again, to be described in Section 4.6.

4.4 Handling Macros

Handling macros has two parts: macro definitions and macro calls. P-Cpp handles a `#define` like Cpp, by creating an entry for the macro definition in its macro table. A *Macro Definition* entry in P-Cpp's macro table contains data about the location of the definition, the location of its `#undef` if it has one, and references to all calls to the macro. The exact location where a macro is defined and undefined is used to calculate its scope during analysis. Unlike Cpp, P-Cpp creates a token for the `#define` line in the tokenized output. The token that represents the `#define` line has a reference to the Macro Definition object (see Fig. 4.9).

Macro calls may appear anywhere in the code and may represent any part of a statement or sequence of statements, so they need to be expanded for parsing to work. Like Cpp, P-Cpp expands macro calls, replacing them in the output by the tokens in the expansion. The difference is that P-Cpp represents a macro call with a *Macro Call* object and labels each token in the macro expansion with that object. Figure

4.9 shows an example of the tokens generated by P-Cpp. The top level of the figure shows a piece of source code with a macro definition and a macro call. The middle level of the figure has the stream of output tokens, with the first token representing the macro definition and the last three tokens representing the expansion of the macro call. The token for the macro definition has a reference to the Macro Definition entry in the bottom layer and the tokens in the macro expansion have a reference to the Macro Call object. Moreover, the Macro Definition and the Macro Call reference each other.

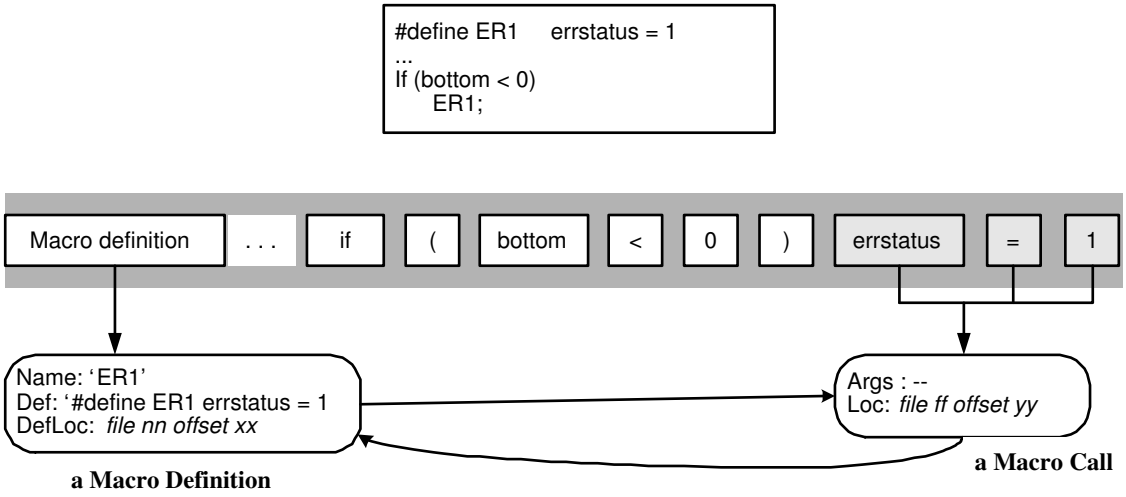

Figure 4.9: Macro expansion and token labelling

If a token comes from the expansion of nested macro calls, the token is labelled with all macro call objects in the sequence that created it, starting with the innermost macro call and ending with the outermost. This introduces *layers* in the representation of token positions. The offset of a token that comes from the expansion of nested macro calls is actually a collection of relative offsets inside the nested macro expansions. Figure 4.10 shows an example of the position of a token '`b`' that is in the expansion of a macro '`M1`', which is in turn called from a macro '`M2`'. The call to '`M2`' is assumed to be at offset 59 in the file.

Another issue with macro definitions results from the interaction with conditional

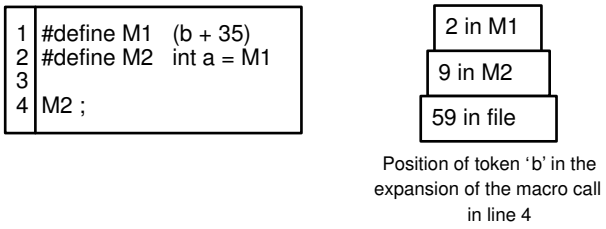

Figure 4.10: Layers in the representation of a token's position

directives. Macro definitions often occur inside the branches of a Cpp conditional, with a different definition of the same macro name in each branch. Figure 4.11 shows an example of a macro `BO_EXBITS` with two definitions.

```
#ifdef __LITTLE_ENDIAN
#define BO_EXBITS       0x18UL
#elif defined(__BIG_ENDIAN)
#define BO_EXBITS       0x00UL
#endif
```

Figure 4.11: A macro with multiple definitions. Each definition is under a different branch of a Cpp conditional

P-Cpp considers all branches of Cpp conditionals simultaneously. Therefore, the macro table created by P-Cpp may have more than one definition for the same macro name. Macro definitions with the same name are distinguished by the condition associated with the enclosing conditional branch. Figure 4.12 shows the entry for macro `BO_EXBITS` in the macro table.

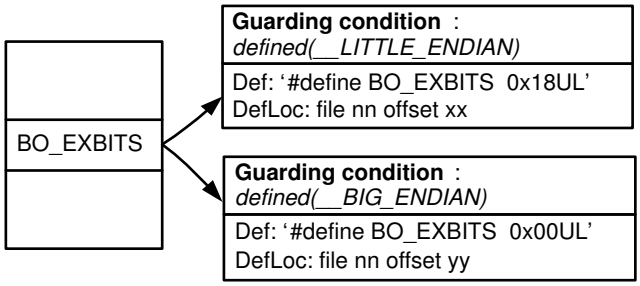

Figure 4.12: Macro table entry for a macro with two definitions

Since macros can have multiple definitions, a macro call may bind to more than

one definition. Specifically, a macro call MC that occurs under a condition C_{MC}, binds to the macro definition under the *same* condition C_{MC}, or, if there is no such definition, it binds to a list of macro definitions MDL such that

$$\forall\ MD \in MDL : compatible(guardingCondition(MD), C_{MC})$$

Two conditions are *compatible* when one is not the logic negation of the other one and they do not appear in the list of incompatible conditions provided by the initial configuration (Section 4.5 describes incompatible conditions).

When a macro call binds to multiple macro definitions, it will have more than one possible expansion. P-Cpp solves this problem by expanding the macro call to a Cpp conditional, with one branch for each possible macro expansion [9]. Figure 4.13 shows an example of the expansion of a call to `BO_EXBITS`.

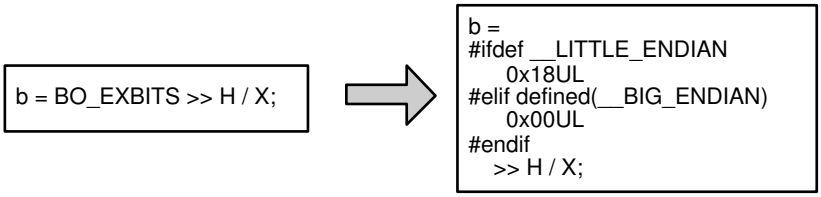

Figure 4.13: Expansion of a macro call that binds to multiple definitions

A harder problem with the interaction of macros and conditionals is that a given symbol may be a macro under a particular conditional branch and a C language element in another branch. Figure 4.14 shows an example that has been extracted from file "dep.h" in the source code of make-3.80. The symbol `dep_name` is defined as a macro in one branch of the Cpp conditional and as a function in the other branch.

```
#ifndef       iAPX286
#define dep_name(d) ((d)->name == 0 ? (d)->file->name : (d)->name)
#else
extern char *dep_name ();
#endif
```

Figure 4.14: A macro and a C symbol defined with the same name

As with the case of multiple macro expansions, when P-Cpp finds a use of a symbol like `dep_name`, it introduces a Cpp conditional with a branch for the macro expansion and a branch for the symbol alone, as shown in Figure 4.15. The problem is, how does P-Cpp know that a symbol is defined in a way other than a macro, if it only stores macros? The answer is that P-Cpp keeps a set of symbols that have been defined as C language elements. Knowing that a symbol is both a language element and a macro allows P-Cpp to expand it with a Cpp conditional for both possibilities.

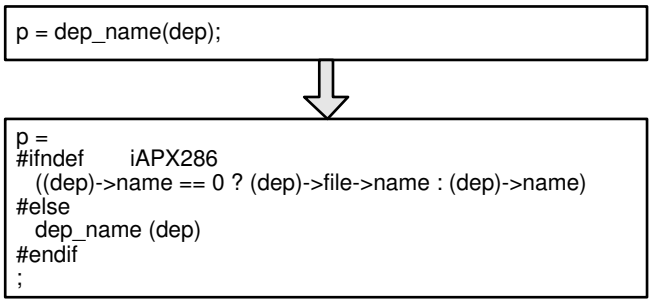

Figure 4.15: Expansion of a symbol defined as a macro and a C language element. The box at the top shows a statement taken from the source code of make-3.80. The box at the bottom shows the expanded form of the statement

Similarly to the specification of Cpp, a macro definition in P-Cpp's formal specification is represented with the sort MacroDef and the macro table with the sort MacroTable. However, a MacroDef in P-Cpp's specification has more information than its counterpart in Cpp: besides the name it also saves the whole text of the definition, its location, the condition under which this definition occurs, the list of calls to this macro and the location of its `#undef` if it has one.

Figure 4.16 shows the specification of macro definitions (module MACRO-DEF) and the macro table (module MACRO-TABLE). Except for two operations, the figure shows the signature of operations but their equations can be found in Appendix B.

Note that in MACRO-TABLE, the map is from an Identifier to a MacroDefList, i.e., there may be more than one macro with the same name, which are gathered in the same MacroDefList. To retrieve a specific MacroDef from the macro table, not only the

```
fmod MACRO-DEF is
  pr STRINGS . pr DEFINE-SYNTAX . pr MACRO-CALL-SYNTAX . pr LOC . pr CONDITIONS .
  sorts MacroDef MacroDefList MacroCallDescr MacroCallDescrList MacroCallStack .
  subsort MacroDef < MacroDefList .
  subsort MacroCallDescr < MacroCallDescrList .
  subsort MacroCallDescr < MacroCallStack .
  op nil : -> MacroDefList .
  op _,_ : MacroDefList MacroDefList -> MacroDefList [assoc comm id: nil] .
  op nil : -> MacroCallDescrList .
  op _;_ : MacroCallDescrList MacroCallDescrList -> MacroCallDescrList [assoc comm id: nil] .
  op nil : -> MacroCallStack .
  op __ : MacroCallStack MacroCallStack -> MacroCallStack [assoc id: nil] .

  op name_def_defLoc_condition_calls_undefLoc_ :
    Identifier MacroDefDir Location CppCondition MacroCallDescrList Location -> MacroDef [ctor] .
  op macroDefs_args_loc_ : MacroDefList StringList Location -> MacroCallDescr [ctor] .
  op name : MacroDef -> Identifier .
  op hasArgs : MacroDef -> Bool .
  op guardCond : MacroDef -> CppCondition .
  op expand : MacroDef -> TokenSequence .                         --- idem Cpp
  op expandWithArgs : MacroDef ArgList -> TokenSequence .         --- idem Cpp
  op expandWithTSArgs : MacroDef TokenSeqList -> TokenSequence . --- idem Cpp
  op undef : MacroDef Location -> MacroDef .
  op addCall : MacroDef MacroCallDescr -> MacroDef .
  op findWithGuardCond : MacroDefList CppCondition -> MacroDef .
  op addCallToAll : MacroDefList MacroCallDescr -> MacroDefList .
  op expandMacroCall : MacroCallDescr -> TokenSequence .
  op expMC-rec : MacroDefList -> TokenSequence .

  var M : MacroDef . var L : Location . var MDL : MacroDefList .
  eq expandMacroCall(macroDefs M args nil loc L) = expand(M) .
  eq expandMacroCall(macroDefs (M , MDL) args nil loc L)
    =  '#if tokenize(guardCond(M)) 'cr  expand(M) 'cr
       expMC-rec(MDL)
       '#endif 'cr .
  eq expMC-rec(nil) = nil .
  eq expMC-rec(M , MDL) = '#elif tokenize(guardCond(M)) 'cr  expand(M) 'cr
                  expMC-rec(MDL) .
endfm

fmod MACRO-TABLE is pr MACRO-DEF .
  sort MacroTable .
  op empty : -> MacroTable .
  op [_:_] : Identifier MacroDefList -> MacroTable .
  op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
  op _[_] : MacroTable Identifier -> MacroDefList .
  op _[_under_] : MacroTable Identifier CppCondition -> MacroDef .
  op _[_<-_] : MacroTable Identifier MacroDefList -> MacroTable .
  op isMacro : Identifier MacroTable -> Bool .
  op remove : MacroDef MacroTable -> MacroTable .
endfm
```

Figure 4.16: Specification of a macro definition and a macro table

name is needed but also the condition under which it applies. This is achieved with the operation _[_under_]. Consequently, a Macro Call object (represented with sort MacroCallDescr) may be associated with a MacroDefList, such that the condition under which the call occurs is compatible with the conditions of all associated MacroDefs.

Some of the operations on a MacroDef are: guarCond, returning the condition that guards the macro definition; expand and expandWithArgs, which behave exactly like their counterparts in Cpp (see Chapter 3); findWithGuardCond, which is used by MacroTable to implement the operation _[_under_] and addCallToAll, which adds the same MacroCallDescr to all MacroDefs in the first argument.

The figure shows the equations for the operation expandMacroCall for the case when the call has no arguments. The first equation is for the case when there is only one associated macro definition, so the operation expand is invoked on that macro definition. The second equation applies when there is more than one macro definition associated with the macro call. In that case, and as explained before, the macro call is expanded to a conditional directive with one branch for each possible expansion. The operation expMC-rec is used to create the intermediate `#elif` branches.

Macro tables are used inside each ProgramFile, to describe the macros defined in a given file, and in PCppState, to maintain the active macro definitions. The state also maintains a stack of current macro calls, to label the tokens that come from macro expansion. This PcppStateAttribute is called currMacroStack and its value is of sort MacroCallStack, which is also defined in module MACRO-DEF.

Finally, Figure 4.17 shows the semantics that P-Cpp gives to a `#define` directive, adding the new macro definition to the macro table.

```
fmod DEFINE-SEMANTICS is pr DEFINE-SYNTAX .
  ex PCPP-DIR-SEMANTICS .
  var I : Identifier . var TS : TokenSequence . var PF : ProgramFile .
  var PFS : ProgramFileStack . var MT : MacroTable . var CSS : CondStackStack .
  var S : PcppState . var L : Location .

  eq state(#define I TS cr, (curPF(PF ; PFS), macroTbl(MT), curCond(CSS), curLoc(L), S))
    = curPF(appendOutputToken(PF, value qid("#define")) ; PFS),
      macroTbl([I : (name I def (#define I TS cr) defLoc L condition
                                                  condFromStackStack(CS ; CSS))] MT),
      curCond(CSS), curLoc(update(L)), S .
  --- similarly for a macro with arguments
endfm
```

Figure 4.17: Semantics of P-Cpp with macro definitions

4.5 Handling Conditional Directives

Chapter 3 described how Cpp selects a single branch of each Cpp conditional construct, discarding conditional directives and the code in the other branches. This behavior of Cpp is not appropriate for a refactoring tool. First, the preprocessed version of a Cpp conditional looks quite different from the source code, since only the code under a single branch remains. Second, applying complex refactorings like "Extract Function", which moves code around, can make it too difficult to recognize the original placement of Cpp conditionals.

Lastly but most importantly, changing the code once it has been targeted to a specific configuration isolates that code from all the rest: if the changed code is merged back, the source code for other configurations may not compile anymore or the behavior may be altered. As an example, consider Figure 4.18, which shows a piece of code from file "`alloca.c`" in the source code of GNU make (version 3.80), where the type pointer has two alternative declarations.

```
#if __STDC__
typedef void *pointer;
#else
typedef char *pointer;
#endif
```

Figure 4.18: Multiple definitions for type pointer

Suppose the code in Figure 4.18 is preprocessed in a configuration where __STDC__ evaluates to *true*, so the second declaration of pointer is discarded. On the preprocessed code the user chooses to rename pointer to ptr, and changes are merged back in the un-preprocessed source code. There would be one declaration for ptr and one for pointer. If now the program is compiled with a configuration in which __STDC__ is *false*, all the references to ptr will cause a compiler error because the declaration will be for type pointer, not ptr.

Therefore, we claim that:

> *A refactoring in the presence of Cpp conditionals is correct (i.e., preserves behavior) if and only if it is correct for all possible system configurations.*

This means that all outputs generated by Cpp for each possible system configuration should be analyzed and refactored together.

One way to represent all possible configurations of a program would be to compute all possible combinations of configuration variables, apply Cpp on the program multiple times, one for each combination, and recombine the results of analyzing each configuration. This is the approach used by Xrefactory [49] and by Fanta and Rajlich [58]. However, this approach is very expensive even for a small-sized program. On the one hand, computing all possible configurations is very difficult [42]. Somé and Lethbridge studied the alternative of computing the possible configurations by trial and error, i.e., try to parse until it fails, applying heuristics to minimize the number of parses needed [42]. On the other hand, even if all possible configurations can be computed, recombining the result of $\simeq 10$ configurations would be very complex and expensive, two things that interactive refactoring tools must avoid [7]. In the case of a large system like the Linux Kernel, with 1,672 configuration variables, this approach is completely unfeasible.

Our approach, instead, is to process all configurations *simultaneously*, producing a *single* tokenized representation for each file, and from there, a single AST inte-

grating all possible configurations for each file [9]. This solution creates a compact representation and in principle it could handle programs as large as the Linux kernel (although we have not tested the whole kernel at this time).

To process all branches of Cpp conditionals, P-Cpp considers the conditions in all branches to be potentially true. P-Cpp does not even evaluate the conditions. Nevertheless, the user may specify some conditions to be *always false*, causing P-Cpp to ignore the Cpp conditional branches with that condition. Examples of false conditions would be "0" used for commenting out code, or something like "defined __cplusplus", which requires a different parser (a C++ parser). Then, except for those listed by the user, each conditional directive is analyzed considering that its condition is potentially true.

Besides false conditions, the user can also specify pairs of *incompatible conditions*. Incompatible conditions are those that cannot be true at the same time. For example, Figure 4.19 shows two pieces of code from two files of the Linux kernel. The file on the left defines memcpy as a macro, whereas the file on the right defines memcpy as a function. The problem is that the file on the right includes the file on the left early on, so when it comes to the definition of memcpy as a function, the tool will do macro expansion on the name of the function.

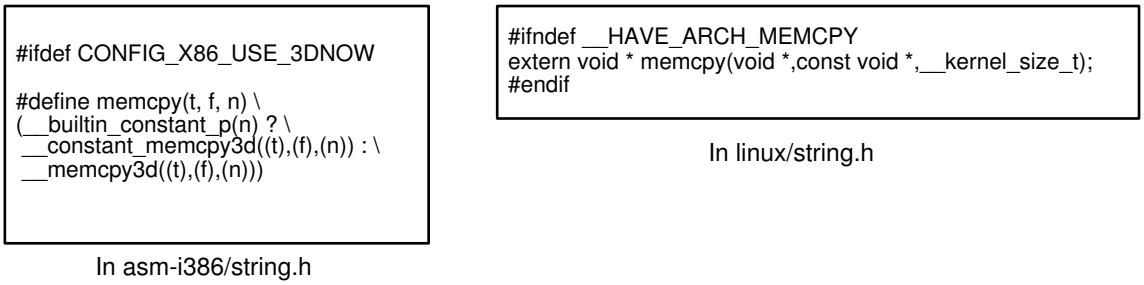

Figure 4.19: Example of incompatible conditions

To prevent this, conditions "defined CONFIG_X86_USE_3DNOW" and "not defined __HAVE_ARCH_MEMCPY" are declared as *incompatible* in the initial configuration.

Both pieces of code will be processed, but the two definitions for memcpy will not clash.

Even though conditions are not evaluated, they do need to be represented for various reasons, for example, to compare them with the false conditions and to label IDG edges and macro definitions. The next section describes the representation of conditions.

4.5.1 Representation of conditions

We can say that every point in the source code, i.e., a character, has an associated condition or *guarding condition*. To describe the condition at each point in the source code (in Definition 2), it is first necessary to describe the condition associated with a conditional directive (in Definition 1). Those two definitions are as follows.

Definition 1: Condition associated with a conditional directive

A conditional directive CD, which creates a branch in a Cpp conditional construct PC, has an associated condition C, which is defined as follows:

- If CD is `#if` exp, $C = exp$.
- If CD is `#ifdef` id, $C = defined(id)$; similarly, if CD is `#ifndef` id, $C = \neg defined(id)$.
- If CD is `#elif` exp, then $C = \neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{i-1} \wedge exp$, where C_1 is the condition of the first conditional directive in PC and C_2 to C_{i-1} are the conditions associated with the previous `#elif` directives in PC, if any.
- If CD is `#else`, $C = \neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{n-1}$, where C_1 is the condition of the first conditional directive in PC and C_2 to C_{n-1} are the conditions associated with each intermediate `#elif` directives in PC, if any.

Definition 2: Condition guarding each point in the source code

Every point P in the source code is guarded by a condition C, where C is defined as follows:

- If P is not inside a Cpp conditional, i.e., it is at level 0 when counting the nesting of conditionals, $C = true$.
- If P is in the branch created by a conditional directive CD so that P is at level 1 (meaning CD is not inside the branch of another Cpp conditional), C is the condition associated with CD, as defined in Definition 1.
- If P is in the branch created by a conditional directive CD so that P is at level l, $C = C_1 \wedge ... \wedge C_l$ where C_l is the condition associated with CD and each C_i from 1 to $l - 1$ is the condition associated with the conditional directive that encloses P at level i.

For example, the condition associated with the point at the start of token `pointer` in Figure 4.20 is $(\neg defined(\_\_GNUC\_\_) \wedge \neg defined(alloca) \wedge \_\_STDC\_\_)$.

```
#if !defined (__GNUC__)
#ifndef alloca
#if __STDC__
typedef void *pointer;
```

Figure 4.20: Condition associated with a point in the source code

There is one exception to Definition 2. Figure 4.21 shows a piece of code with a macro definition for `NSEC_PER_SEC` under condition $\neg defined(NSEC\_PER\_SEC)$. Later on, in the same file, or another one, there is a reference to `NSEC_PER_SEC` to initialize variable `xtime` under condition $defined(NSEC\_PER\_SEC)$. Is that reference a call to macro `NSEC_PER_SEC`? It should be, because Cpp's behavior would be: if the macro is not defined, then define it; then, if the macro is defined, use it. However, the behavior of P-Cpp described in Section 4.4 will not consider that

reference as a macro call and `NSEC_PER_SEC` would be an undefined symbol. The reason is that the condition at that reference to `NSEC_PER_SEC` is the negation of the condition at the definition (therefore the conditions are *incompatible*).

```
#ifndef NSEC_PER_SEC
#define NSEC_PER_SEC (1000000000L)
...
...
#ifdef NSEC_PER_SEC
extern int xtime = TICK_NSEC / NSEC_PER_SEC;
```

Figure 4.21: Example of exception of Definition 2

P-Cpp has some heuristics that solve this problem: a definition for a macro M that is under condition $\neg defined(M)$, cancels out the condition $\neg defined(M)$. More precisely, a definition for a macro M at a point in the code where, by Definition 2, the guarding condition is

$$C = C_1 \wedge ... \wedge C_{i-1} \wedge \neg defined(M) \wedge C_{i+1} \wedge ... \wedge C_n$$

causes P-Cpp to remove the condition $\neg defined(M)$ from C, so that the new condition guarding the tokens in that Cpp conditional branch will be

$$C' = C_1 \wedge ... \wedge C_{i-1} \wedge C_{i+1} \wedge ... \wedge C_n$$

To understand how the heuristics work, let us expand on the example of Figure 4.21, adding other definitions for `NSEC_PER_SEC` under other conditions. The new example in Figure 4.22 occurs often in real code, where each definition usually appears in different header files.

The expansion for the call to `NSEC_PER_SEC` in the last line that appears in Fig. 4.22, will be a Cpp conditional with three branches, one for each possible expansion of `NSEC_PER_SEC`. The condition associated with each branch in the macro expansion, according to Definition 1, will be:

- $MIPS$

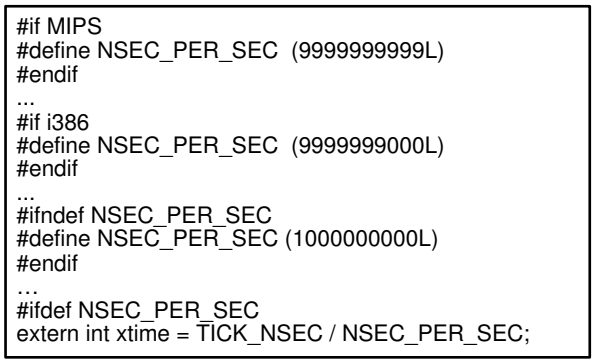

Figure 4.22: Example of multiple definition for 'NSEC_PER_SEC'

- $\neg MIPS \wedge i386$
- $\neg MIPS \wedge \neg i386$

The condition associated with the last branch models the situation correctly: if the other conditions are both false, it means `NSEC_PER_SEC` was not defined before, and that is the condition under which this definition was created.

To model the conditions of conditional directives, CRefactory has a class hierarchy rooted at AbstractCppCondition. There is a special parser for conditions, CppConditionParser, used by P-Cpp to create instances of the classes in the hierarchy of AbstractCppCondition. One of the subclasses is CppCondition, which represents the constant expression that follows a `#if` or a `#elif`. Another class is CppDefinedCondition, which represents a "defined" condition. The class CppNotCondition represents a "not" or $\neg$ operator. Finally, the condition for a token at level 0 is represented by CppTrueCondition, while the condition for a token at level l ($l > 0$) is represented by CppAndCondition, which "ands" the nested conditions from level 1 to l.

For example, the condition associated with the start of token `pointer` in Figure 4.20 translates in CRefactory's representation to:

$$
\begin{array}{l}
CppAndCondition(\\
\quad CppAndCondition(\\
\quad\quad CppNotCondition(CppDefinedCondition(\_\_GNUC\_\_)), \\
\quad\quad CppNotCondition(CppDefinedCondition(alloca))), \\
\quad CppCondition(\_\_STDC\_\_))
\end{array}
$$

4.5.2 Conditions as labels

Cpp conditionals are often used to select between alternative files to include in the compilation unit. For this reason, and as explained in Section 4.3, edges in the Include Dependency Graph are labelled with the condition under which the file inclusion occurs, which is the condition at the point where the `#include` directive starts.

Moreover, Cpp conditionals are also used to provide alternative definition for the same macro. Figure 4.11 showed an example. As explained in Section 4.4, entries in the macro table are labelled with the condition under which the macro definition occurs, which is the condition at the point where the `#define` directive starts (note that the condition cannot change half way in the macro definition, since a conditional directive cannot appear inside it).

Other elements that can have alternative definitions thanks to conditional directives are C program elements, such as variables or functions, as the case of type pointer in Figure 4.18. However, P-Cpp does not create the symbol table as it does with the macro table, since creating the symbol table is the task of the semantic analyzer that runs after parsing and AST construction. Nevertheless, P-Cpp generates the necessary information for the semantic analyzer to construct the symbol table with the condition that guards each symbol definition, by labelling each token in the tokenized output with the condition under which it applies. The condition associated with a token is the condition at the point where the token starts. Note that the condition at the point where a token starts remains the same until the token ends, i.e., it cannot change half-way within a token.

In order to label IDG edges, macros and tokens with their guarding condition, P-Cpp needs to keep track of the current condition at each point in a file, for every file being preprocessed. The representation of the current condition inside a file is implemented with a stack, called **Current Condition Stack**. Upon a conditional directive, P-Cpp parses the condition associated with it, creates a "Condition" object from the hierarchy of AbstractCppCondition to represent it, and pushes this object into the Current Condition Stack. Then, P-Cpp uses the conjunction of Condition objects in the Current Condition Stack to label each token inside the branch. Conditions are popped from the stack upon the next conditional directive in the same Cpp conditional or upon a `#endif`.

The Current Condition Stack of each ProgramFile being preprocessed is stored in a stack, called curCond, which makes it a stack of stacks of Condition objects. Each element of curCond corresponds one-to-one with the elements in the stack of ProgramFiles.

Figure 4.23 shows the module CONDITIONS that specifies the conditions created by P-Cpp upon a conditional directive. All subclasses in the hierarchy of AbstractCppCondition are represented with a single sort: CppCondition. The other sorts defined in the module are CondStack, which is the sort of the Current Condition Stack for each Program File; CondStackStack, which is the stack of CondStacks for all Program Files being preprocessed; CondSet, to represent the set of false conditions that the user specifies in the input to P-Cpp; CondPair and CondPairSet, which are used to represent pairs of incompatible conditions, again listed by the user in the input to P-Cpp (see Section 4.2).

Module CONDITIONS contains the equation for the predicate compatible holding between two conditions, as described in Section 4.4 in the context of macro expansion. The third argument of this predicate is the set of incompatible conditions, constructed from the user input. The operation isNegationOf returns true if the first argument is

```
fmod CONDITIONS is
  pr ALL-COND-EXP-SYNTAX .
  sorts CppCondition CondStack CondStackStack CondSet CondPair CondPairSet .
  subsort CppCondition < CondStack .
  subsort CondStack < CondStackStack .
  subsort CppCondition < CondSet .
  subsort CondPair < CondPairSet .
  op condition : CondExp -> CppCondition [ctor] .
  op trueCondition : -> CppCondition [ctor] .
  op _and_ : CppCondition CppCondition -> CppCondition .
  op _isNegationOf_ : CppCondition CppCondition -> Bool .
  op compatible : CppCondition CppCondition CondPairSet -> Bool .
  op nil : -> CondStack .
  op _;_ : CondStack CondStack -> CondStack [assoc id: nil] .
  op nil : -> CondStackStack .
  op _;_ : CondStackStack CondStackStack -> CondStackStack [assoc id: nil] .

  op empty : -> CondSet .
  op __ : CondSet CondSet -> CondSet [assoc comm id: empty] .
  op _in_ : CppCondition CondSet -> Bool .
  op <_;_> : CppCondition CppCondition -> CondPair [ctor] .
  op empty : -> CondPairSet .
  op __ : CondPairSet CondPairSet -> CondPairSet [assoc comm id: empty] .
  op _in_ : CondPair CondPairSet -> Bool .
  op condFromStack : CondStack -> CppCondition .
  op condFromStackStack : CondStackStack -> CppCondition .
  op tokenize : CppCondition -> TokenSequence .

  vars C C' : CppCondition . var Incomp : CondPairSet . var S : CondStack .
  eq compatible(C, C', Incomp) = not (C isNegationOf C') and not (< C ; C' > in Incomp) .
  eq condFromStack(nil) = trueCondition .
  eq condFromStack(C ; nil) = C .
  eq condFromStack(trueCondition ; S) = condFromStack(S) .
  eq condFromStack(C ; S) = C and condFromStack(S) .
endfm
```

Figure 4.23: Specification of conditions

the logical negation of the second, like $defined(M)$ with $\neg defined(M)$.

Moreover, Fig. 4.23 shows the equations for operation condFromStack, which returns the condition that will label a token, given a CondStack that represents the Current Condition Stack.

Figure 4.24 shows a partial version of module COND-DIR-SEMANTICS, which specifies the behavior of P-Cpp for a `#if` directive. The equations in module COND-DIR-SEMANTICS use five PcppStateAttributes: falseConds is the set of false conditions specified by the user; curCond represents the stack of Current Condition Stacks; skip and nestLevelOfSkipped have the same purpose they had in the state of Cpp (see Fig. 3.16) and curPF is the stack of ProgramFiles currently being processed (introduced in Section 4.3). The equations are described after the figure.

```
fmod COND-DIR-SEMANTICS is pr COND-DIR-SYNTAX .
  ex PCPP-DIR-SEMANTICS .  pr COND-EXP-SEMANTICS .
  var CE : CondExp . var FC : CondSet . var CS : CondStack . var CSS : CondStackStack .
  var S : PcppState . var N : Nat . var PF : ProgramFile . var PFS : ProgramFileStack .
  --- Case 1 of #if: Not skipping -> Not skipping
  ceq state(#if CE cr, (falseConds(FC), curCond(CS ; CSS), skip(false), curPF(PF ; PFS), S))
    = falseConds(FC), curCond((eval(CE) ; CS) ; CSS), skip(false),
       curPF(appendOutputToken(PF, value '#if macroCalls nil cond condFromStack(CS)) ; PFS), S
    if not(eval(CE) in FC) .
  --- Case 2 of #if: Not skipping -> Skipping
  eq state(#if CE cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S  [owise] .
  --- Case 3 of #if: Skipping -> Skipping
  eq state(#if CE cr, (skip(true), nestLevelOfSkipped(N), S))
    = skip(true), nestLevelOfSkipped(N + 1), S .
endfm
```

Figure 4.24: Semantics of P-Cpp upon conditional directives

The equation for Case 1 uses the operation eval on a CondExp, which instead of actually evaluating the condition (as Cpp would) creates a representation of `CE` as a CppCondition (see Appendix B). This equation will be applied as long as the CppCondition resulting from `CE` is not in the set of false conditions. In that case the branch for this conditional directive is processed and the CppCondition is pushed on top of curCond. Moreover, a token representing the `#if` line is appended to the ProgramFile at the top of curPF. The token will have as value the quoted identifier `'#if` (note that this is a simplification but the whole directive line is actually stored in the token). The token will also have the condition resulting from the Current Condition Stack (curCond) as condition label (cond field in CRToken).

The other two equations are for the case that the branch of the conditional directive is not processed, either because the associated condition was in the set of false conditions (case 2 in Fig. 4.24) or because the enclosing conditional was already being skipped (case 3).

Figure 4.24 also shows that P-Cpp leaves conditional directives in the tokenized output, as it does with the other directives.

There is one more problem with conditional directives that is actually the hardest to solve. Our solution to this problem is the key of CRefactory's novel approach to representing C programs with Cpp directives. The next section describes the problem

and gives a high-level view of the solution, and following sections describe the solution in detail.

4.5.3 Problem with conditional directives: Incomplete syntactic units

The main problem with conditional directives is that they usually "break" statements and other C constructs. The problem can also be stated in the following way: the branches created by Cpp conditionals are usually *syntactically incomplete*, i.e., they do not contain complete syntactical units. Figure 4.25 shows an example from file "rule.c" in make-3.80, where the branches of the Cpp conditional contain only the start of the if statement, i.e., the branches are not complete syntactical units and cannot be parsed together.

```
#ifdef VMS
    if (dep->changed && strchr (name, ' :' ) != 0)
#else
    if (dep->changed && *name == ' /' )
#endif
    {
        freerule (rule, lastrule);
        …
    }
```

Figure 4.25: Incomplete syntactic units

Allowing conditional directives at any point in each C grammar production is not viable, as it would result in a large and ambiguous grammar. The tool DMS solves this problem by restricting conditional directives to appear at certain places in the grammar, and manually modifying the code that does not comply [53]. This solution is simple but not scalable to large, open-source projects. We do not want to restrict the places where conditional directives can occur.

Our solution consists of manipulating conditional directives in the internal representation of the code, so that each conditional directive appears only in between the

C constructs listed in Table 4.1, i.e., at the same level as statements or declarations, structure field declarations, enumeration values or array initializer values. The general idea is to complete the branches of a Cpp conditional with the text that precedes and/or follows the conditional, until each branch is *syntactically complete*, i.e., until it can be parsed independently of the other branches. With this transformation of conditionals, CRefactory is able to obtain a single representation of the source code for all possible system configurations, so the CRefactory parser then works in a *single pass*.

The process of completing conditionals is accomplished in two passes of P-Cpp through the source code. In the first pass, P-Cpp tokenizes the input and recognizes incomplete Cpp conditionals, creating descriptors that contain information of how to complete them. In the second pass, incomplete conditionals are completed by applying the **Conditional Completion Algorithm**. This algorithm moves and copies tokens as stated by the descriptors created in the first pass. After this second pass all Cpp conditionals are complete syntactical units, i.e., they can be integrated in the C grammar and the source code can now be parsed.

4.5.4 Recognizing incomplete Cpp conditionals

A Cpp conditional is considered *complete* when its branches enclose a whole syntactic construct, or a whole list of them, from the constructs that appeared in Table 4.1. To recognize if a conditional is incomplete, P-Cpp needs to distinguish the tokens that mark the beginning and end of each of those constructs. While P-Cpp tokenizes the input in the first pass through the source code of a file, P-Cpp works as a pushdown automaton, keeping track of each recognized syntactic construct by maintaining a state stack, similar to what a parser would do, but only for the constructs in Table 4.1.

We will use Maude's rewrite rules to describe the possible states and transitions

of P-Cpp while it recognizes syntactic construct in the first pass. These rewrite rules are specified in a *system module*, which describes *states* as constructor operations and *transitions* with rewrite laws. Figure 4.26 shows the system module PCPP-FIRST-PASS with the states and transitions possible for P-Cpp.

The system module PCPP-FIRST-PASS first imports the functional module TOKEN specified in the syntax of Cpp (Chapter 3). The beginning or end of syntactic constructs are represented with the sort Constr. All possible Constrs are listed as Constr constructors. For example, the constructor inConstruct represents a statement or declaration and endOfConstruct represents the end of it. The constructor inFor represents the first part of a for-statement: the keyword "for" plus the expressions inside parentheses. The distinction of "for" from other statements was made to prevent confusing the use of ';' inside the expressions of a for statement (where conditional directives are not allowed) with the use of ';' as a statement separator. The constructor inFor takes a natural number as parameter that counts the number of open parentheses. Similarly, inInitializer, which represents an array or struct initializer, takes a natural number as parameter that counts the number of open braces. Only when those numbers get to 0 has the construct ended. The sort ConstrStack represents the stack of opened syntactic constructs. The sort Input-ConstrState is then a pair of TokenSequence, the input tokens, and ConstrStack, the current opened constructs.

Rewrite laws represent allowed transitions between Input-ConstrStates. Maude's rewrite laws are declared with the keyword `rl` followed by the name of the law in brackets. The transition between states is denoted with the symbol "=>". A conditional rewrite rule is declared with the keyword `crl` and takes a condition at the end, just like conditional equations. Some of the rewrite laws in module PCPP-FIRST-PASS are:

startComp. When the top of the ConstrStack is anything except inStruct, the appearance of an open brace makes P-Cpp push inCompositeStmt and endOfCon-

```
mod PCPP-FIRST-PASS is
  pr TOKEN .
  sorts Constr ConstrStack Input-ConstrState .
  subsort Constr < ConstrStack .
  op inConstruct        : -> Constr [ctor] .
  op endOfConstruct     : -> Constr [ctor] .
  op inCompositeStmt    : -> Constr [ctor] .
  op inEnum         : -> Constr [ctor] .
  op inEnumElem         : -> Constr [ctor] .
  op endOfEnumElem  : -> Constr [ctor] .
  op inFor          : Nat -> Constr [ctor] .
  op inInitializer  : Nat -> Constr [ctor] .
  op inInitValue        : -> Constr [ctor] .
  op endOfInitValue     : -> Constr [ctor] .
  op inStruct       : -> Constr [ctor] .
  op nil : -> ConstrStack [ctor] .
  op __ : ConstrStack ConstrStack -> ConstrStack [assoc id: nil] .
  op <_;_> : TokenSequence ConstrStack -> Input-ConstrState [ctor] .
  vars X X2 : Token . var TS : TokenSequence . var P : Constr . var S : ConstrStack . var N : Nat .
  crl [startComp]   : < ''{ TS ; P S > => < TS ; endOfConstruct inCompositeStmt S >
        if P =/= inStruct .
  rl [endComp]  : < ''} TS ; inCompositeStmt S > => < TS ; endOfConstruct S > .
  rl [startEnum]    : < 'enum TS ; S > => < TS ; inEnum S > .
  crl [stayEnum]    :  < X TS ; inEnum S > => < TS ; inEnum S > if (X =/= ''{) and (X =/= ''}) .
  rl [startEnumValues] : < ''{ TS ; inEnum S > => < TS ; inEnumElem inEnum S > .
  crl [stayEnumValue] : < X TS ; inEnumElem S > => < TS ; inEnumElem S >
               if (X =/= '',) and (X =/= ''}).
  rl [endEnumValue] : < '', TS ; inEnumElem S > => < TS ; endOfEnumElem S > .
  crl [startEnumValue] : < X TS ; endOfEnumElem S > => < TS ; inEnumElem S > if X =/= ''} .
  rl [endEnumValues] : < ''} TS ; inEnumElem S > => < ''} TS ; S > .
  rl [endEnumValues2] : < ''} TS ; endOfEnumElem S > => < ''} TS ; S > .
  rl [endEnum]  : < ''} TS ; inEnum S > => < TS ; inConstruct S > .
  rl [startFor]     : < 'for TS ; S > => < TS ; inFor(0) S > .
  crl [stayFor] : < X TS ; inFor(N) S > => < TS ; inFor(N) S > if ( X =/= ''( ) and ( X =/= '') ) .
  rl [forOpenPar] : < ''( TS ; inFor(N) S > => < TS ; inFor(s(N)) S > .
  crl [forClosePar] : < '') TS ; inFor(s(N)) S > => < TS ; inFor(N) S > if (N =/= 0) .
  rl [endFor]   :  < '') TS ; inFor(1) S > => < TS ; inConstruct S > .
  rl [startInit]    : < ''= ''{ TS ; S > => < TS ; inInitializer(0) S > .
  crl [stayInit]    : < X TS ; inInitializer(0) S > => < TS ; inInitializer(0) S >
          if (X =/= ''{) and (X =/= ''}) .
  rl [startInitValues] : < ''{ TS ; inInitializer(0) S > => < TS ; inInitValue inInitializer(1) S > .
  crl [stayInitValue] : < X TS ; inInitValue S > => < TS ; inInitValue S >
                  if (X =/= '',) and (X =/= ''{) and (X =/= ''}).
  rl [stayInitValue2] : < ''{ TS ; inInitValue inInitializer(N) S >
               => < TS ; inInitValue inInitializer(s(N)) S > .
  crl [stayInitValue3] : < ''} TS ; inInitValue inInitializer(s(N)) S >
               => < TS ; inInitValue inInitializer(N) S >   if (N =/= 0) .
  rl [endInitValue] : < '', TS ; inInitValue S > => < TS ; endOfInitValue S > .
  crl [startInitValue] : < X TS ; endOfInitValue S > => < TS ; inInitValue S > if X =/= ''} .
  rl [endInitValues] : < ''} TS ; inInitValue inInitializer(0) S > => < ''} TS ; S > .
  rl [endInitValues2] : < ''} TS ; endOfInitValue inInitializer(0) S > => < ''} TS ; S > .
  rl [endInit]  : < ''} TS ; inInitializer(0) S > => < TS ; inConstruct S > .
  rl [startStruct] : < 'struct TS ; endOfConstruct S > => < TS ; inStruct S > .
  crl [startStruct2] : < 'struct TS ; P S > => < TS ; inStruct P S >  if (P =/= endOfConstruct) .
  crl [stayStruct] : < X TS ; inStruct S > => < TS ; inStruct S > if (X =/= ''{) and (X =/= ''}) .
  rl [startField] : < ''{ TS ; inStruct S > => < TS ; endOfConstruct inStruct S > .
  rl [endStruct]    : < ''} TS ; inStruct S > => < TS ; inConstruct S > .
  rl [endConst] : < ''} TS ; endOfConstruct S > => < ''} TS ; S > .
  crl [startConstr] : < X X2 TS ; endOfConstruct S > => < X2 TS ; inConstruct S >
    if (X =/= 'enum) and (X =/= 'for) and not (X == ''= and X2 == ''{) and (X =/= 'struct)
        and (X =/= ''{) and (X =/= ''}) .
  crl [stayConstr] : < X X2 TS ; inConstruct S > => < X2 TS ; inConstruct S >
    if (X =/= 'enum) and (X =/= 'for) and not (X == ''= and X2 == ''{) and (X =/= 'struct)
        and (X =/= ''{) and (X =/= ''}) .
  rl [endConstr] : < '; TS ; inConstruct S > => < TS ; endOfConstruct S > .
endm
```

Figure 4.26: Specification of P-Cpp's pushdown automata

struct at the top of the stack. While in a composite statement, inConstruct and endOfConstruct are at the top of the stack to represent inner statements or declarations.

startEnum. The token `'enum` makes P-Cpp consume the token and push inEnum (an enumerator construct) at the top of the ConstrStack.

startEnumValues. When inEnum is at the top of the stack, the open brace makes P-Cpp push inEnumElem (the start of an enumerator value) in the stack.

endConstr. Simple statements and declarations finish with a ';' so the appearance of this token makes P-Cpp replace inConstruct by endOfConstruct at the top of the ConstrStack.

With this representation of states, a Cpp conditional is complete *if and only if* its branches start and end when the top of the ConstrStack is **endOfConstruct**. Conversely, if a Cpp conditional *starts* while **inConstruct** is at the top of the stack, the Cpp conditional is set to have a *bad start*. Moreover, if a Cpp conditional *ends* when the top of the stack is **inConstruct**, the Cpp conditional has a *bad ending*.

When P-Cpp encounters conditional directives in this first pass, it creates descriptors for them. These descriptors contain enough information to fix incomplete conditionals in the second pass. Some of the information in a CppConditionalDescriptor appears in Table 4.2. CppConditionalDescriptors form a tree that represents the nesting of conditionals.

Figure 4.27 shows the pseudo-code for how P-Cpp creates and sets the attributes of CppConditionalDescriptors and CppConditionalBranchDescriptors.

In the pseudo-code, the objects that represent constructs in the construct stack (sort Constr in Fig. 4.26) have two attributes: $startPosition$ and $condsWBadEnding$. The attribute $startPosition$ stores the source code position at which the construct started. This value is used to set the field $startPosShouldBe$ of CppCondition-

Table 4.2: Data in a CppConditionalDescriptor

startPosition	The position where the Cpp conditional starts in the current file
endPosition	The position where it ends
badStart	True if the Cpp conditional has a bad start
badEnding	True if it has a bad ending
startPosShouldBe	The position where the Cpp conditional should start to be complete
endPosShouldBe	The position where the Cpp conditional should end to be complete
branches	A sequence of CppConditionalBranchDescriptors

alDescriptors of conditionals that start in the middle of the construct. The attribute *condsWBadEnding* holds a list of CppConditionalDescriptors of conditionals that break the current construct. Later on, when P-Cpp finds the end of the current construct, it sets the value *endPosShouldBe* of all descriptors in the list *condsWBadEnding* to be the current position.

4.5.5 Conditional Completion Algorithm

Once the tree of CppConditionalDescriptors has been created, the **Conditional Completion Algorithm** is run to fix incomplete conditionals. The Conditional Completion Algorithm applies a behavior-preserving transformation to the tokenized version of the source code that turns incomplete conditionals into complete ones. The algorithm works moving and copying tokens as dictated by the CppConditionalDescriptors.

The Conditional Completion Algorithm works by matching each Cpp conditional with one of seven cases that it can handle. The seven cases are in two major categories: *non-overlapping conditionals* and *overlapping conditionals*. The non-overlapping are the first three cases, which apply when a Cpp conditional may or may not have nested Cpp conditionals inside its branches (children) but when being completed, it will not overlap with another conditional (sibling). The overlapping cases are the last four, which apply when there is overlap, meaning that sibling conditionals break the *same*

```
case (current token = #if, #ifdef or #ifndef) {
  desc := new CppConditionalDescriptor.
  desc.startPosition := current position.
  If( top(contrStack) = endOfConstruct)
      desc.badStart := false
  else {
      desc.badStart := true.
      desc.startPosShouldBe := (top(constrStack)).startPosition.
  } }
case (current token = #elif or #else){
  branch := new CppConditionalBranchDescriptor.
  branch.startPosition := current position.
  Add branch to branches of current Cpp conditional.
}
case (current token = #endif) {
  desc := current Cpp conditional.
  desc.endPosition := current position.
  If( top(constrStack) = endOfConstruct)
      desc.badEnding := false.
  else {
      desc.badEnding := true.
      Add desc to (top(constrStack)).condsWBadEnding.
  } }
case (current token marks beginning of construct) {
  Push corresponding Constr into constrStack with
        newState.startPosition := current position.
}
case (current token marks end of current construct) {
  For each desc ∈ ((top(constrStack)).condsWBadEnding) {
      desc.endPosShouldBe := current position. }
  Push appropriate Constr into constrStack
}
```

Figure 4.27: Pseudo-code for first pass of P-Cpp

syntactic construct. Although some cases may be considered simple variations of another, the distinction of different cases inside each category allows the algorithm to specialize and improve the performance on individual cases.

Figure 4.28 depicts the seven cases that the Conditional Completion Algorithm can handle. For each case, the figure represent a statement with a grayed rectangle, and a Cpp conditional with the letter '`E`' (each leg of the letter is a conditional directive and the space between legs has the code inside a branch). This does not mean that the Cpp conditional can only have two branches, on the contrary, there is no limit on the number of branches. The small '`E`'s inside a bigger one mean that the

Cpp conditional represented by the bigger `E` can have children (inner conditionals). In Case 4, each Cpp conditional is represented by a block letter '`C`' because they can only have one branch.

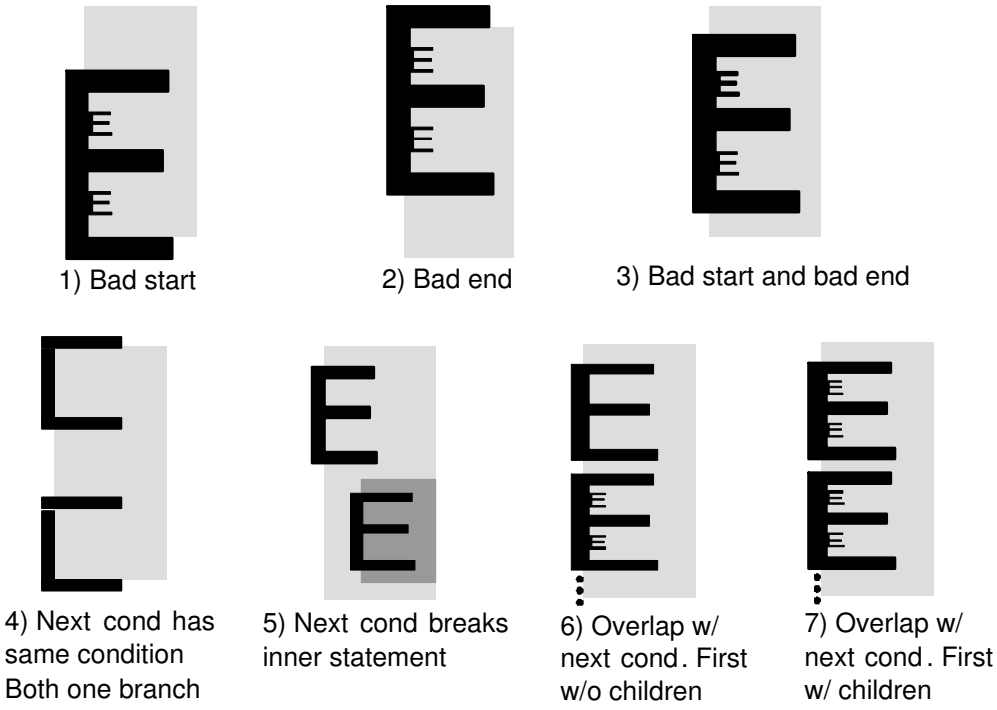

Figure 4.28: Cases of the Conditional Completion Algorithm

Following is a description of each of the seven cases.

Case 1 *The Cpp conditional PC, which is at level of nesting i, starts in the middle of a syntactic construct. PC may or may not have children (other Cpp conditionals inside its branches). There is no other Cpp conditional at level i breaking the same syntactic construct.*

In this case, *PC* has a bad start and breaks a syntactic construct in two parts, with the first part of the construct being at level $i - 1$. The tokens in the first part of the construct are moved and copied to the beginning of each branch in *PC*, as shown in Figure 4.29.

If *PC* has children, the tokens in the first part of the broken construct are

moved to the innermost child, appending the tokens between Cpp conditionals in the path to the innermost child, if there were any.

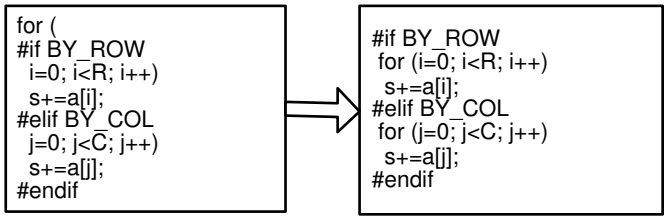

Figure 4.29: Case 1 of completing conditionals

Case 2 *The Cpp conditional PC, which is at level of nesting i, ends in the middle of a syntactic construct. PC may or may not have children. There is no other Cpp conditional at level i breaking the same syntactic construct.*

In this case, *PC* has a bad ending and breaks a syntactic construct in two parts, with the second part of the construct being at level $i - 1$. The tokens in the second part of the construct are moved and copied to the end of each branch in *PC*. If the conditional does not have an `#else` line, one is added with the text that completes the Cpp conditional. See the example in Figure 4.30.

If *PC* has children, the tokens in the second part of the broken construct are moved to the innermost child, after the tokens between Cpp conditionals in the path to the innermost child, if there were any.

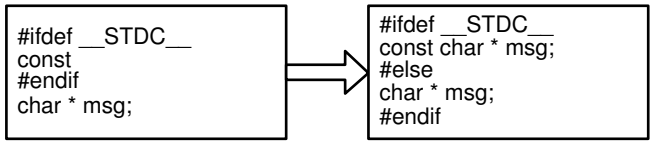

Figure 4.30: Case 2 of completing conditionals

Case 3 *The Cpp conditional PC, which is at level of nesting i, starts in the middle of a syntactic construct and ends in the middle of the same or another construct.*

PC may or may not have children. There is no other Cpp conditional at level i breaking the same syntactic constructs as PC.

This is the combination of cases 1 and 2. First the start of the conditional is completed as in case 1. Then, its end is completed as in case 2.

Case 4 *The Cpp conditional PC_1 is at level of nesting i, it has a single branch with condition C and ends in the middle of a syntactic construct (it has a bad ending). The second part of the statement is in another Cpp conditional PC_2 at the same level i, having a single branch with the same condition C (PC_2 has a bad start). Neither PC_1 nor PC_2 have children.*

An example of this case appears to the left of Figure 4.31. In this case, PC_1 is combined with PC_2 to form a single Cpp conditional. Before the `#endif` for PC_1, P-Cpp adds the text in between both Cpp conditionals plus the text in the matching branch of PC_2. Then an `#else` line is added with the text that was in between the Cpp conditionals, as shown to the right of in Figure 4.31.

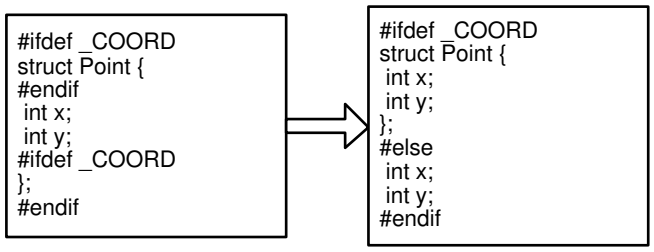

Figure 4.31: Case 4 of completing conditionals

Case 5 *The Cpp conditional PC_1, which is at level of nesting i, has a bad ending and may or may not have a bad start. The next conditional at the same level i, PC_2, has at least a bad start and when being complete, will become the child of PC_1. Neither PC_1 nor PC_2 have children.*

Figure 4.32 shows an example where the first conditional breaks the while statement and the second conditional breaks the assignment statement inside the

body of the while. In this case, the next conditional is completed inside the stream of tokens that complete the end of the current conditional.

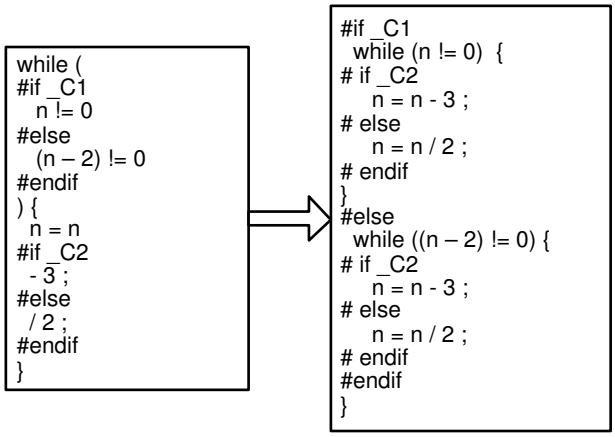

Figure 4.32: Case 5 of completing conditionals

Case 6 *The Cpp conditional PC_1, which is at level of nesting i, breaks a syntactic construct (it either has a bad start or a bad ending or both). The next conditional at the same level i, PC_2, breaks the same construct than PC_1. PC_1 does not have children and PC_2 may have. PC_2 may also be in Case 5 or Case 6 with its next conditional.*

This case is common once macros have been expanded by P-Cpp, and there are two or more macro calls in the same statement, each macro call binding to more than one definition. In the case that PC_1 and PC_2 come from the expansion of the *same* macro, there is no need to make all combinations. P-Cpp combines PC_1 and PC_2 in a single Cpp conditional.

Figure 4.33 shows an example of Case 6. The second conditional (PC_2) becomes nested in each branch of the first one (PC_1), and gets completed inside each branch of PC_1. The tokens that complete the start of PC_2 inside each branch B of PC_1, are the tokens that complete the start of PC_1, plus the tokens in

the branch B plus the tokens in between both conditionals. This creates all combinations of conditions in PC_1 and PC_2.

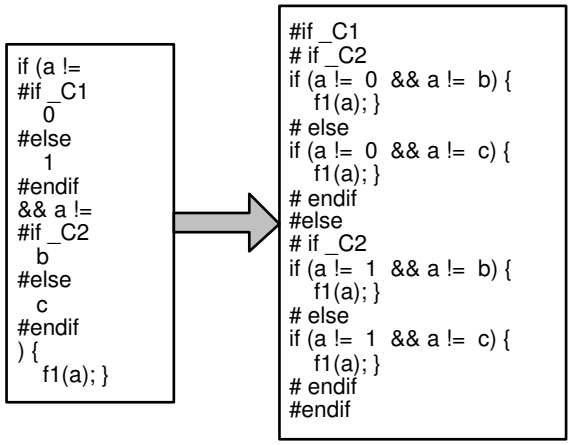

Figure 4.33: Case 6 of completing conditionals

Case 7 *The Cpp conditional PC_1, which is at level of nesting i, breaks a syntactic construct (it either has a bad start or a bad ending or both). The next conditional at the same level i, PC_2, breaks the same construct than PC_1. PC_1 has children and PC_2 may have children. That is, PC_2 is in case 6 with PC_1's children. PC_1's children may be in Case 5 or Case 6 among them.*

This case also appears between Cpp conditionals that P-Cpp introduces with macro expansions.

Figure 4.34 shows an example. The first Cpp conditional (PC_1) and the last one (PC_2) break the same if-else construct. Since PC_1 has a child ($PC_{1.1}$), PC_2 becomes nested in each branch of $PC_{1.1}$. That is, Case 7 is solved by applying Case 6 between PC_1's children and PC_2. The figure shows the tokens inside $PC_{1.1}$ and PC_2 in bold font, to help the reader visualize all four combinations when the conditionals get completed.

These seven cases are the ones we have found in our case studies (open source packages). Note that Case 4 is actually a variation of Case 6, specialized for the case

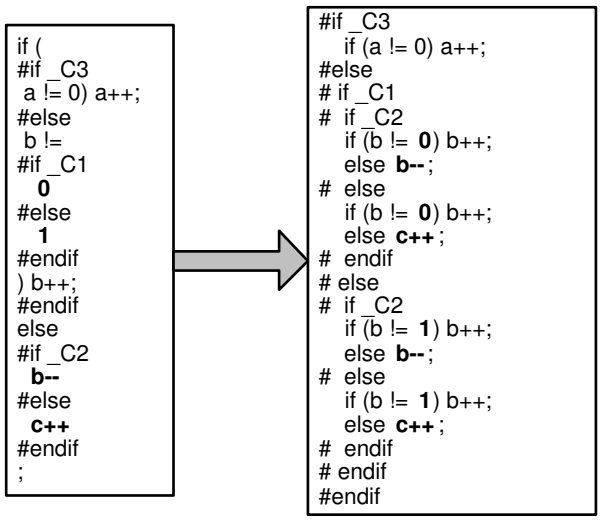

Figure 4.34: Case 7 of completing conditionals

that the conditions in both conditionals are the same. The case not supported by the Conditional Completion Algorithm is a variation of Case 5, where the conditionals may have children. We have never encountered this situation in practice.

The description of the cases above gives an overview of how the Conditional Completion Algorithm works in each case. What follows is a detailed description.

Figure 4.35 has the pseudo-code for function completeConditional in the the Conditional Completion Algorithm. Function completeConditional receives a CppConditionalDescriptor, *desc*, as parameter. It first checks if the Cpp conditional that *desc* represents has a bad start (Case 1), and if so it calculates the tokens that complete the start of the conditional. Second, the function checks if *desc* is in Case 6 with the next conditional (*nextDesc*), that is, if the Cpp conditionals represented by *desc* and *nextDesc* break the same construct. In this case, the next conditional is completed in each branch of the current conditional, creating this way all combinations. Following conditionals are checked recursively for intersections with the previous ones.

If *desc* is not in Case 6 with *nextDesc*, the function checks if they fall in Case 7. If so, each child of *desc* is completed as in Case 6 with *nextDesc*.

If the next conditional does not break the same construct (is not in Case 6 or 7

```
completeConditional(desc: CppConditionalDescriptor)
{
    if (desc.badStart) {
        desc.tokensCompletingStart := tokens from desc.startPosShouldBe
                                      to desc.startPosition. }
    nextDesc := nextConditional(desc).
    if (isCase6(desc, nextDesc))
        for each branch ∈ desc.branches
            Complete nextDesc inside branch.
    else if (isCase7(desc, nextDesc))
        for each branch ∈ desc.branches
            for each childDesc ∈ children(desc, brach)
                Complete childDesc as case 6 with nextDesc.
    else {
        if (desc.badEnding) {
          if (¬ isCase4(desc, nextDesc) ∧¬ isCase5(desc, nextDesc)) {
              desc.tokensCompletingEnd := tokens from desc.endPosition
                                          to desc.endPosShouldBe.}
          else if (isCase4(desc, nextDesc))
              Compute ending in next conditional.
          else if (isCase5(desc, nextDesc))
              Complete nextDesc inside desc.tokensCompletingEnd.
        }
        for each branch ∈ desc.branches
            completeConditionalBranch(desc, branch).
    }
}
```

Figure 4.35: Pseudo-code of completeConditional

with *desc*), and if the conditional that *desc* represents has a bad ending (Case 2), the function calculates the tokens that complete the end of the conditional. Here it is also possible that the next conditional is in Case 4 or in Case 5 with *desc*. In Case 5, the next conditional is completed inside the stream of tokens that complete the end of the current conditional.

After the tokens that complete the start and the end of each incomplete conditional have been calculated, P-Cpp actually moves and copies those tokens, while it labels them, in each branch of the conditional. Figure 4.36 shows the pseudo-code for how P-Cpp completes each branch. The basic idea is that the tokens that complete the start of the conditional are *moved* to the beginning of the first branch and *copied* to the beginning of the other branches. If the conditional has a bad ending, the tokens

that complete the end of the conditional are *moved* to the end of the last branch and *copied* to the end of the other branches. Tokens are labelled accordingly so this manipulation can be reversed as described in the next section.

```
completeConditionalBranch(desc, branchDesc)
{
   if (desc.badStart)
      if (isFirstBranch(desc, branchDesc)) {
         Move desc.tokensCompletingStart to beginning of branch while
               labelling these tokens as 'moved forward'.}
      else {
         Copy desc.tokensCompletingStart to beginning of branch while
               labelling these tokens as 'copied'. }
   for each childDesc ∈ children(desc, brachDesc)
         completeConditional(childDesc).
   if (desc.badEnding)
      if (isLastBranch(desc, branchDesc))
         Move desc.tokensCompletingEnd to end of branch while labelling
               these tokens as 'moved backwards'.
      else
         Copy desc.tokensCompletingEnd to end of branch while labelling
               these tokens as 'copied'.
}
```

Figure 4.36: Pseudo-code of completeConditionalBranch

4.5.6 Pretty-printing of Cpp conditionals

Pretty-printing is not the job of P-Cpp, but we include this section here to explain how the labelling that P-Cpp does on tokens actually works at the time of pretty-printing a Cpp conditional to reverse the completion of the conditional.

The pretty-printer is a visitor of the abstract syntax tree (AST). It prints the leaf nodes of the AST according to the tokens they represent and the tokens' labels. The pretty-printer uses two queues: $ifsQueue$ stores the start directive of Cpp conditionals with bad start. These start directives go later in the output, after the tokens that have been *moved forward* to complete the first branch. The other queue is the $movedBackQueue$, and this one is for nodes representing tokens that have been *moved backwards* to complete the end of a conditional. The nodes in the $movedBackQueue$

are printed after the #endif of the current conditional. Nodes that represent *copied* tokens are not printed. Figure 4.37 shows the pseudo-code for pretty-printing a leaf node that does not come from macro expansion.

```
pretty-print(node)
{
    case(node represents #if, #ifdef or #ifndef)
        if ((assocCppConditional(node)).badStart)
          queue(ifsQueue, node).
        else Print node.
    case (label(node) 'not moved' or 'moved forward')
        Check if top(ifsQueue) should be printed.
        Print node.
    case (label(node) = 'copied')
        /* do nothing */
    case (node represents #elif or #else)
        Print node.
    case (label(node) = 'moved backwards')
        queue(movedBackQueue, node).
    case (node represents #endif) {
        Print node.
        Print all nodes in movedBackQueue }
}
```

Figure 4.37: Pseudo-code for pretty-printing

For Cases 4, 5, 6 and 7 of the Conditional Completion Algorithm, where independent conditionals get combined and the conditionals do not come from macro expansion, the pretty-printing algorithm is more complex. For instance in Case 6, the pretty-printer requires two more queues. The *middleQueue* stores the tokens that have been labelled as 'moved forward' by the second conditional but that go after the end of the first conditional (i.e., in the middle of both conditionals). The *secondCondQueue* stores the 'not moved' tokens that belong to the second conditional. Further details of these cases are not provided because they hardly appear in practice.

4.6 Reusing Representations

This section finally explains in detail how P-Cpp reuses previously generated representations. Let us first review the example of Figure 4.5, repeating it here but adding

the code for file “A.h”, with a conditional macro definition. Figure 4.38 shows the example.

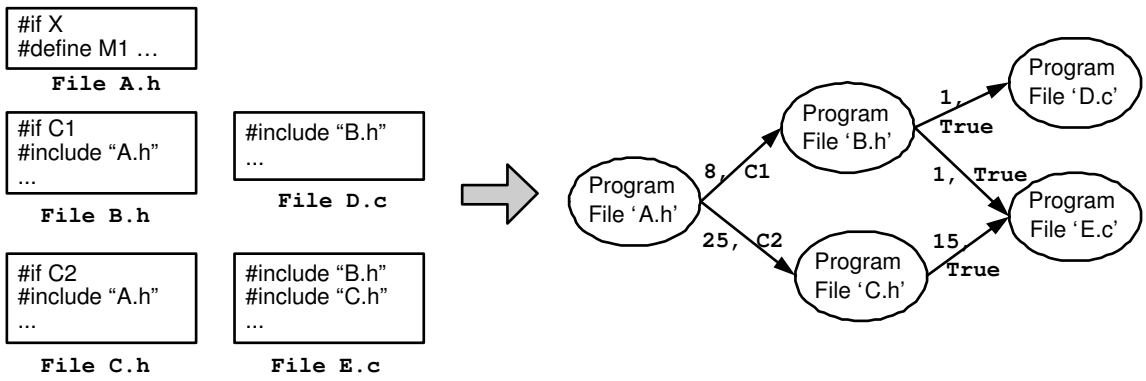

Figure 4.38: Example of include dependencies revisited

Consider the case when P-Cpp's current program file stack (curPF in the specification) has the following program files: (“A.h” ; “B.h” ; “E.c”) and P-Cpp is processing the macro definition in “A.h”. The condition guarding the definition of macro `M1` at this point is:

$$CppAndCondition(CppCondition(C1), CppCondition(X))$$

and that is the condition with which `M1` is added to the active macro table of P-Cpp (macroTbl in the specification).

Let us suppose that P-Cpp processes a file every time it is included. Now consider the second case when P-Cpp's current program file stack has the following program files: (“A.h” ; “C.h” ; “E.c”) and P-Cpp is processing the macro definition in “A.h”. The condition guarding the definition of macro `M1` this time is:

$$CppAndCondition(CppCondition(C2), CppCondition(X))$$

and P-Cpp's macro table should be updated with this new condition for `M1`.

If instead of preprocessing file “A.h” again, P-Cpp wants to reuse the representation for “A.h”, it needs to know the macros defined in “A.h” so it can reinsert

them in the macro table. Therefore, the ProgramFile for "A.h" has another attribute: macrosDefined, which stores the macros defined in "A.h". However, which condition should label macro `M1` inside the ProgramFile for "A.h"? It should be the current condition inside "A.h", represented by the Current Condition Stack, ($CppCondition(X)$ in the example). The condition of the inclusion of file "A.h" is stored in the edge from "A.h", as shown in Fig. 4.38. When macros are reinserted in the active macro table of P-Cpp, the guarding condition of each one is the conjunction of the conditions of all predecessor edges in the path to the file and the condition inside the file. The complete specification of this process appears in Figure 4.39. The figure shows operations and equations in different modules that make possible to reuse ProgramFile representations.

With this strategy, P-Cpp works more efficiently than Cpp minimizing the number of times a file is processed, by reusing the representation of a file instead of rebuilding it each time it is included.

There is one exception to reusing the representation already created for a file and that is when the definition of macros that the file calls (if any) are not the same than the previous time the file was preprocessed. If the macros called by the file changed, the tokenization of macro expansions will differ. This may happen when an included file `F` calls macros defined in the file that includes `F` prior to the `#include F` line. Although this is considered bad practice, Cpp allows it because of the copy-and-paste style in which file inclusion is implemented. Figure 4.40 shows an example with set of files and their source code.

In the example of Figure 4.40, the file "A.h" calls macro `BO_EXBITS`, which is defined in file "B.h" before the include for "A.h". Moreover, there is a file "C.h" with a different definition of macro `BO_EXBITS`, and "C.h" also includes "A.h" after the macro definition. There is also a file "X.c" that includes both "B.h" and "C.h". Assuming `var1` is only assigned to in "A.h", the value of `var1` in the code of "X.c"

```
fmod DEFINE-SEMANTICS is
  ...
  var I : Identifier . var TS : TokenSequence . var PF : ProgramFile .
  var MT : MacroTable . var CS : CondStack . var CSS : CondStackStack . var L : Location .
  var S : PcppState . var PFS : ProgramFileStack .

  eq state(#define I TS cr, (curPF(PF; PFS), macroTbl(MT), curCond(CS; CSS), curLoc(L), S))
    = curPF(addMacroDefinition(appendOutputToken(PF, ...),
          (name I def (#define I TS cr) defLoc L condition condFromStack(CS))) ; PFS),
      macroTbl([I : (name I def (#define I TS cr) defLoc L
                     condition condFromStackStack(CS ; CSS))] MT),
      curCond(CS ; CSS), curLoc(update(L)), S .
endfm

fmod INCLUDE-SEMANTICS is
  ...
  ceq state(#include FN cr, (idg(PF' PF IDG), curPF(PF ; PFS), macroTbl(MT),
        curMacroStack(MCS), curCond(CS ; CSS), curLoc(LO), S))
    = idg((addEdgeFrom PF' to PF at LO under condFromStack(CS)) PF IDG),
      curPF(appendOutputToken(PF,
           value qid("#include" + FN) macroCalls MCS cond condFromStack(CS)) ; PFS),
      macroTbl(MT macrosDefInPredsOf(PF')), curMacroStack(MCS), curCond(CS ; CSS), curLoc(LO), S
    if name(PF') == FN .
endfm

fmod INCLUDE-DEP-GRAPH is
  ...
  op macrosDefInPredsOf : ProgramFile -> MacroTable .
  op macrosDefIn : IdgEdgeList -> MacroTable .
  var FN : String . var TS : CRTokenStream . vars Suc Pred : IdgEdgeList . var MD : MacroTable .
  var PF : ProgramFile . var L : Location . var C : CppCondition .
  eq macrosDefInPredsOf(name FN tokenStream TS includingFiles Suc includedFiles Pred
        macrosDefined MD) = MD macrosDefIn(Pred).
  eq macrosDefIn(nil) = empty .
  eq macrosDefIn(((dest PF pos L under C), Pred))
    = andConditionToAll(macrosDefInPredsOf(PF), C) macrosDefIn(Pred) .
endfm

fmod MACRO-TABLE is
  ...
  op andConditionToAll : MacroTable CppCondition -> MacroTable .
  var N : Identifier . var L : MacroDefList . var MT : MacroTable . var C : CppCondition .
  eq andConditionToAll(([N : L] MT), C) = [N : andGuardCondToAll(L, C)] andConditionToAll(MT, C) .
endfm

fmod MACRO-DEF is
  ...
  op andGuardCondToAll : MacroDefList CppCondition -> MacroDefList .
  op andGuardCond : MacroDef CppCondition -> MacroDef .
  vars C C' : CppCondition . var M : MacroDef . var MDL : MacroDefList . var N : Identifier .
  var D : MacroDefDir . vars L UL : Location . var MCL : MacroCallDescrList .
  eq andGuardCondToAll(nil, C) = nil .
  eq andGuardCondToAll((M , MDL), C) = andGuardCond(M, C) , andGuardCondToAll(MDL, C) .
  eq andGuardCond(name N def D defLoc L condition C calls MCL undefLoc UL, C')
    = name N def D defLoc L condition (C and C') calls MCL undefLoc UL .
endfm
```

Figure 4.39: Additions to P-Cpp's specification to allow reuse of representations

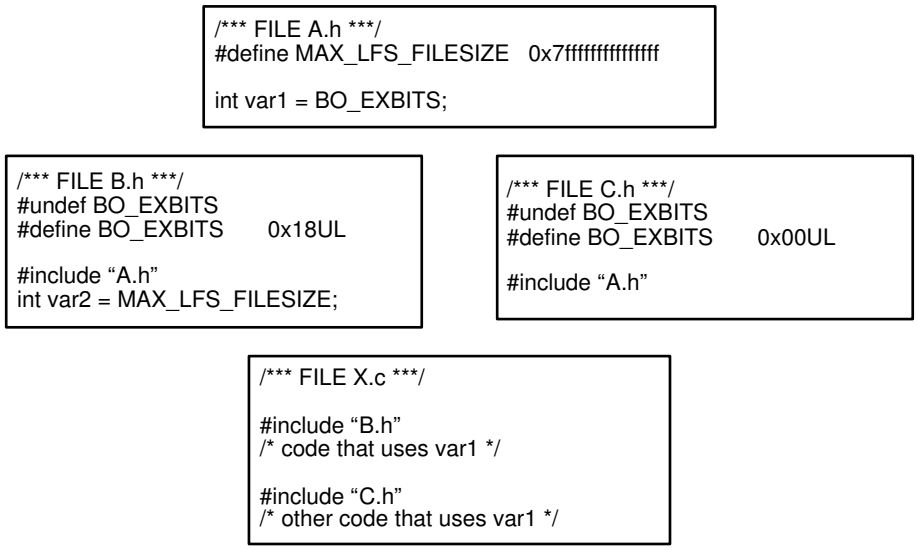

Figure 4.40: Different definition of a macro depending on the order of file inclusion

that follows the `#include` "`B.h`" line is `0x18UL`. However, the value of `var1` after the `#include` "`C.h`" line is `0x00UL`.

P-Cpp solves this situation by modifying the macro table slightly and preprocessing the file again. When a file `F` is processed for the first time, P-Cpp saves the current macro table at the start of `F`, in an attribute of the ProgramFile for `F` called activeMacrosAtStart. Then, when P-Cpp finds a `#include F` line again, it checks if the current definition of macros that `F` calls (if any) are the same than the previous time `F` was preprocessed (i.e., it compares the current macro table with the macros in activeMacrosAtStart inside the representation of `F`). If the macro definitions are the same, no further work is necessary, as `F` has been already tokenized correctly. However, if the macro definitions differ, P-Cpp adds all conflicting macro definitions to the macro table and distinguishes them by labelling each macro with a CppIncludePathCondition, a fabricated condition that names the file that contains the definition. The macro table for the example of Figure 4.40 would look as shown in Figure 4.41. P-Cpp then pseudo-preprocesses `F` again so the tokenization reflects the new possible macro expansions.

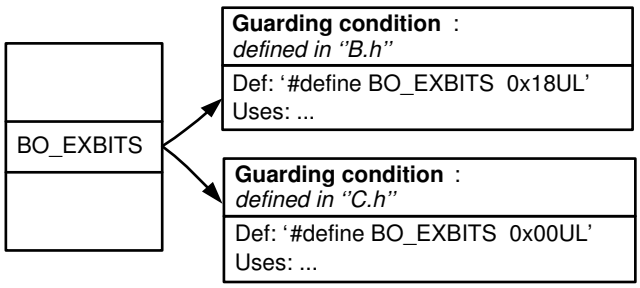

Figure 4.41: Entry for a macro whose definition depends on the order of file inclusion

Therefore, even when this problem occurs, P-Cpp works around it to obtain a single representation of each file, which integrates all possible configurations, all possible macro expansions and all possible orders in which the inclusion of a file may happen.

Chapter 5

Program Representations that Integrate C and Cpp

This chapter describes the program representations that CRefactory creates. These representations integrate C program elements, like variables and functions, with Cpp directives and macro calls.

The set of these representations is called the *program model*. While this chapter describes CRefactory's program model, the next chapter will show how the model can be used to analyze the preconditions of refactorings and execute refactorings in the presence of preprocessor directives, preserving the *un-preprocessed* source code.

A C program is composed of a set of files. CRefactory's representation of a C program is an Include Dependency Graph (IDG), where each node represents a file and each edge represents an inclusion dependency. The IDG was described in the previous chapter (Section 4.3) because its nodes and edges are constructed by P-Cpp. Moreover, for each IDG node (a Program File), P-Cpp creates a tokenized representation. After pseudo-preprocessing, there is a component in CRefactory called CRProcessor that takes the output of P-Cpp and builds, for each Program File in the IDG, an **abstract syntax tree** and a **symbol table** [9]. These are the two main data structures used during refactoring.

Furthermore, abstract syntax trees and symbol tables are indexed by a **Program Repository** that can be queried during refactoring. This chapter first describes the structure of abstract syntax trees and symbol tables. The Program Repository and its components are next presented, with the list of queries that the Program Repository understands. The last section of the chapter talks about the algorithms that update the program model after a refactoring.

5.1 Abstract Syntax Tree

Abstract syntax trees (ASTs) are simple but powerful representations of programs. Although there are more complex program representations, such as program dependency graphs (PDGs) [36], they are complex and require considerable time to build [6]. For refactoring tools to be useful, speed is very important. ASTs provide sufficient information to implement powerful and fast refactorings, as demonstrated by the Refactoring Browser [7].

In the same spirit as the Refactoring Browser and the majority of refactoring tools [51], CRefactory uses the AST of a program not only to perform analysis of the code but also to transform it. That is, refactorings execute by replacing nodes in the AST. This approach differs from the one taken by Xrefactory where analysis is performed on the AST but replacing occurs in the source text [49]. We believe that our approach is more scalable for complex refactorings.

Since refactorings are performed on ASTs, the trees must include information of Cpp directives if the goal is to refactor them together with the rest of the C code. For example, it is necessary to know if an AST node derives from a macro expansion or if it belongs to a specific branch of a Cpp conditional. As far as we know, this thesis is the first publication that approaches the construction of ASTs including complete information of Cpp directives.

ASTs are constructed during parsing, by attaching a script to each grammar production that creates nodes representing that production [67]. Therefore, the grammar must have productions for Cpp directives, so nodes can be created to represent them. We have extended the grammar for the C language with productions for Cpp directives. As described in the previous chapter, the extended grammar allows Cpp directives in between the syntactic constructs that appear in Table 5.1.

Table 5.1: C syntactic constructs allowed in between Cpp directives

Statement
Declaration
Structure field
Enumerator value
Array initializer value

The tokenized representation that P-Cpp creates has Cpp directive lines represented with a single token. For example, the token that represents a conditional directive stores, besides the directive keyword and its source code position, the condition associated with it. Therefore, Cpp directives are terminals in the grammar specification.

Figure 5.1 shows the grammar productions for a `statement`, a `controlLine` (i.e., a Cpp directive), a `macroDirective` and a `conditionalDirective`. The figure uses standard grammar notation, where each grammar production names the non-terminal represented, then a semi-colon, and then the different alternatives separated by a '|' character. Terminals of the grammar are surrounded by angle brackets. The scripts that create AST nodes appear in between curly braces. They have Smalltalk code that creates a node, listing the node's class name first and a message that sets the token on the node. The expression '1' represents the first element in the right-hand side of the production.

Appendix C shows the complete extended grammar accepted by CRParser, the

CRefactory parser. CRefactory uses SmaCC [68], a parser generator for Smalltalk, to generate CRParser from the grammar specification.

```
statement
    : labeledStatement
    | compoundStatement
    | expressionStatement
    | selectionStatement
    | iterationStatement
    | jumpStatement
    | controlLine ;

controlLine
    : <INCLUDE>
        {CRControlIncludeNode token: '1'}
    | macroDirective
    | conditionalDirective
    | <OTHER_DIRECTIVE>
        {CRControlOtherNode token: '1'} ;

macroDirective
    : <DEFINE>
        {CRControlDefineNode token: '1'}
    | <UNDEF>
        {CRControlUndefineNode token: '1'} ;

conditionalDirective
    : <CONDITIONAL_START_IF>
        {CRControlConditionalStartIfNode token: '1'}
    | <CONDITIONAL_START_IFDEF>
        {CRControlConditionalStartIfdefNode token: '1'}
    | <CONDITIONAL_ELIF>
        {CRControlConditionalElifNode token: '1'}
    | <CONDITIONAL_ELSE>
        {CRControlConditionalElseNode token: '1'}
    | <CONDITIONAL_END>
        {CRControlConditionalEndNode token: '1'} ;
```

Figure 5.1: Grammar productions for statement and Cpp directives

Figure 5.2 shows some source code with Cpp directives and the resulting AST. The use of colors in the figure represents the condition that labels a node. Light gray is used in the subtree rooted at `Declaration` to represent the node label: CppDefinedCondition(X86_PC9800), while dark gray is used to label node `CppDefine` with CppNotCondition(CppDefinedCondition(X86_PC9800)) (see Subsection 4.5.1 for a specification of Condition objects).

Figure 5.2 also shows that the AST does not represent the nesting of Cpp conditional constructs, i.e., the nodes inside a Cpp conditional branch are at the same level as the node for the conditional directive that starts the branch. Although we could have added Cpp conditional constructs to the C grammar, the added complexity was

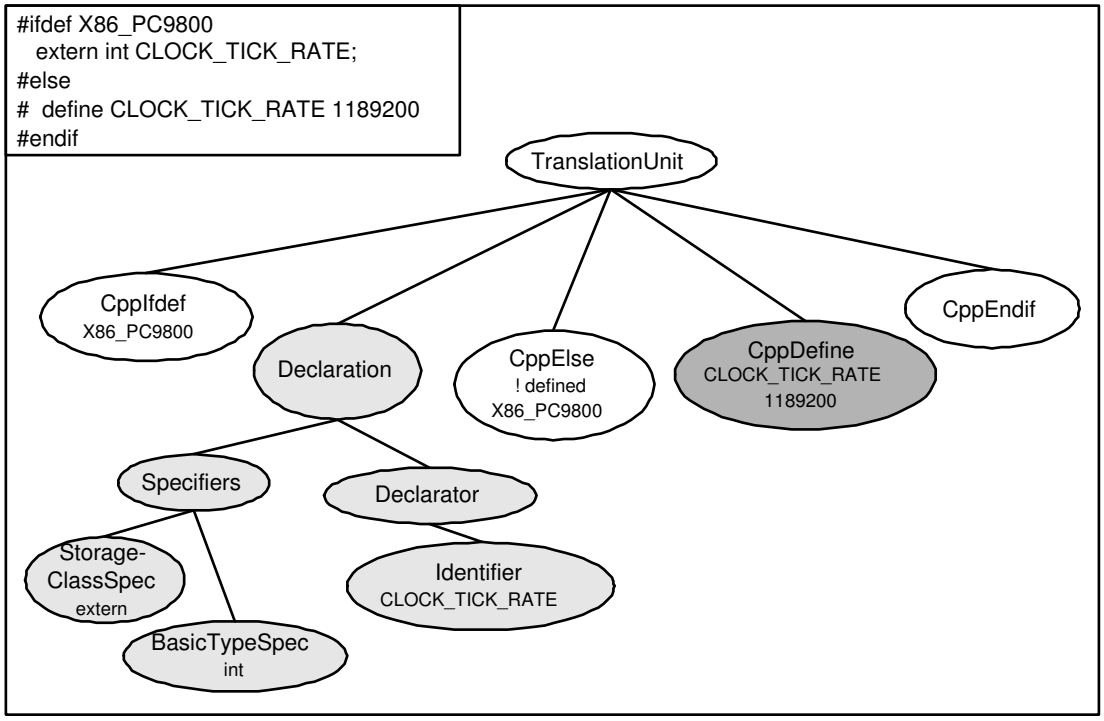

Figure 5.2: Abstract syntax tree with Cpp directives as nodes

not needed. Having the nodes for conditional directives and the conditions that label the nodes in one branch is sufficient for our purposes. Nevertheless, if Cpp conditional constructs were needed, they are represented in the tree of CppConditionalDescriptors created by P-Cpp (see Subsection 4.5.4).

Figure 5.3 shows the AST resulting from some source code with a Cpp conditional that had to be completed by P-Cpp. The only node labelled with the condition CppDefinedCondition(__STDC__) is the one corresponding to the only token that was inside the Cpp conditional before completion. That is, the completion of conditionals does not scramble the correct labelling of nodes with conditions.

Conditions are not the only labels for AST nodes. As explained in Chapter 4, macro calls cannot be directly represented in the grammar, and they need to be expanded. Tokens resulting from macro expansion are labelled by P-Cpp with a reference to the macro call they come from. Since AST nodes representing terminals know their tokens, they "inherit" their tokens' labels. Figure 5.4 shows the AST

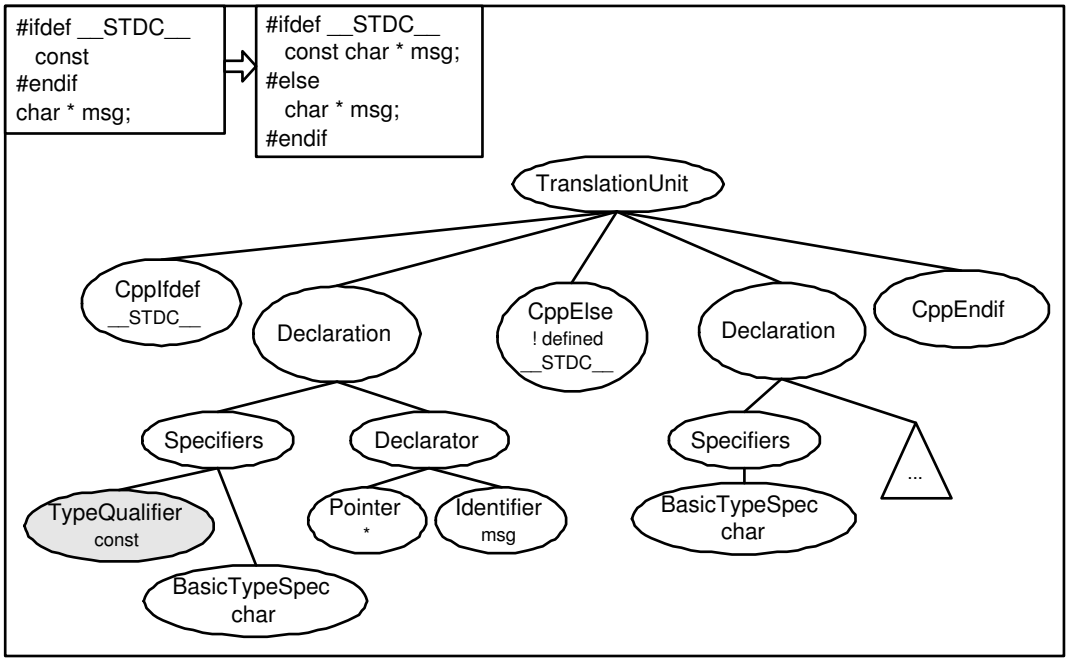

Figure 5.3: Abstract syntax tree after conditional completion

resulting from a piece of code with a macro call. The piece of code is the same as in Figure 4.9, which showed the tokenization of the macro call expansion. Figure 5.4 shows the same Macro Definition and Macro Call objects that appeared in Fig. 4.9, and how AST nodes refer to them. The figure also shows that the node representing the macro definition (a CppDefine node) refers to the Macro Definition object.

Turning to the implementation of the AST, all nodes belong to the hierarchy of ProgramNode. There are 81 subclasses of ProgramNode, that is, 81 different types of nodes, although 4 of them are abstract classes: AbstractExpressionNode, StatementNode, DeclarationSpecifier and CppControlNode (for Cpp directives).

There are different classes that can traverse ASTs, which follow the "Visitor" design pattern [69]. These classes belong to the hierarchy of ProgramNodeVisitor. One of the subclasses is the ASTInfoCollector, which constructs the symbol tables and populates the Program Repository. Another class is the CRFormatter, which pretty-prints the ASTs.

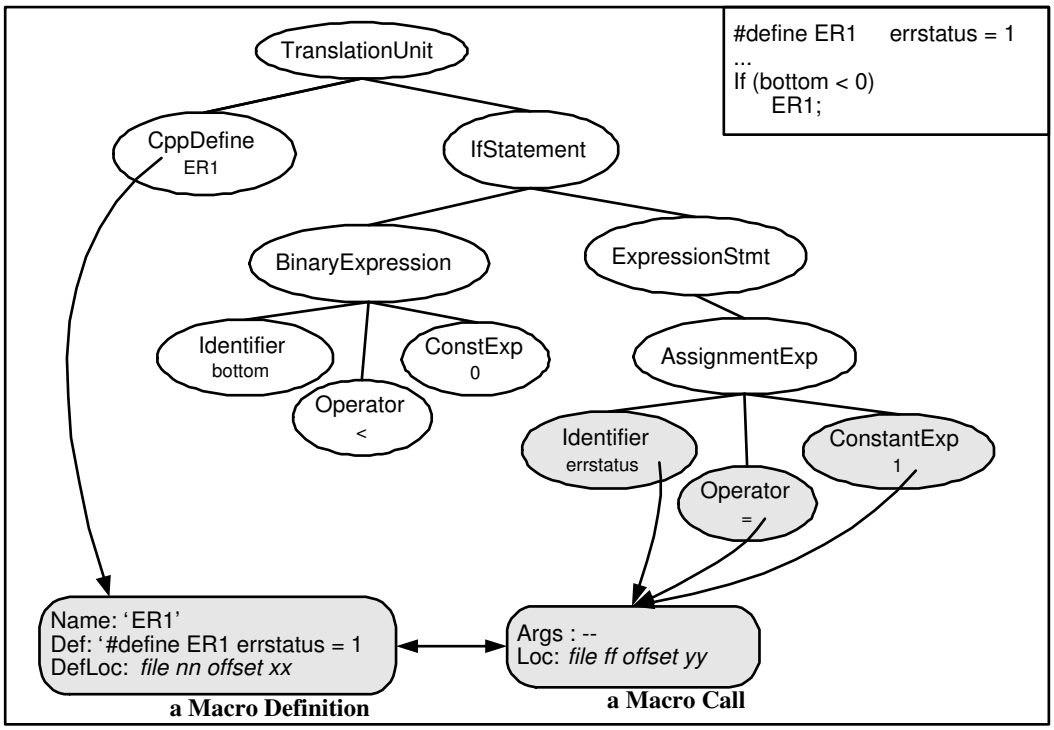

Figure 5.4: Abstract syntax tree with labels for macro expansion

5.2 Symbol Table

A *symbol table* records the identifiers declared in a *program scope* and their attributes [70]. In CRefactory, symbol tables are instances of the class CRSymbolTable. Symbol tables are constructed by an ASTInfoCollector, a visitor of the AST. There is one symbol table for each ProgramFile in the IDG with the global definitions in that file, i.e., the definitions that occur in the file scope, outside any function. There is also a symbol table for each nested scope inside a file. That is, there is a symbol table associated with each function and each block inside a function. There is also a global symbol table for *external symbols*, i.e., variables and functions declared in two or more files that can be linked together [26]. Scopes will be discussed further in the next section.

A symbol table looks exactly like a macro table, but with different kinds of entries. As shown in Chapter 4, conditional directives may introduce multiple definitions for

the same symbol (see Figure 4.18). Like the macro table, each entry in CRefactory's symbol table can point to more than one definition, and definitions for the same symbol are distinguished by their guarding condition, i.e., the condition under which the definition applies. Aversano et al. propose a similar symbol table with multiple entries, although the entries in their symbol table are labelled with configuration parameter values instead of guarding conditions [54].

Chapter 4 also mentioned the problem of having a symbol defined as both a macro under a Cpp conditional branch and as a C language element in another branch or another Cpp conditional (Figure 5.2 showed an example). This implies that the symbol table must allow entries for macro definitions as well as C language elements. For example, the symbol table entry for symbol `CLOCK_TICK_RATE` in the code of Figure 5.2 looks as in Figure 5.5 below.

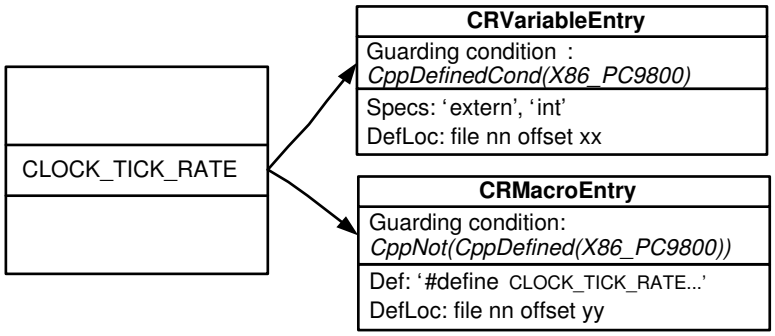

Figure 5.5: Symbol table entry for a variable and a macro with the same name

Table 5.2 lists the different kinds of entries in CRefactory's symbol tables. The table also shows the names of the classes that represent symbol table entries. These classes belong to the hierarchy rooted at CRSymbolTableEntry.

A CRStructEntry is used to represent both structures and unions. The fields of a structure or union are stored in a separate symbol table inside their CRStructEntry, since the enclosing structure creates a different *name space* for its fields [26]. Conversely, enumeration constants are not stored within their enumerated type because they have the same scope and name space as the enumerated type. All other

Table 5.2: Types of entries in CRefactory's symbol table

Kind of entry	Class that represents it
Variable	CRVariableEntry
Function	CRFunctionEntry
Structure	CRStructEntry
Enumerated type	CREnumEntry
Enumeration constant	CREnumConstEntry
Label	CRLabelEntry
User-defined type	CRUserDefinedType
Macro definition	CRMacroEntry

`typedef` declarations that are not structures or enumeratives are represented with class CRUserDefinedType.

The attributes that are stored for each type of symbol table entry appear in Figure 5.6. The figure shows the hierarchy of classes rooted at CRSymbolTableEntry, and the attributes defined in each class.

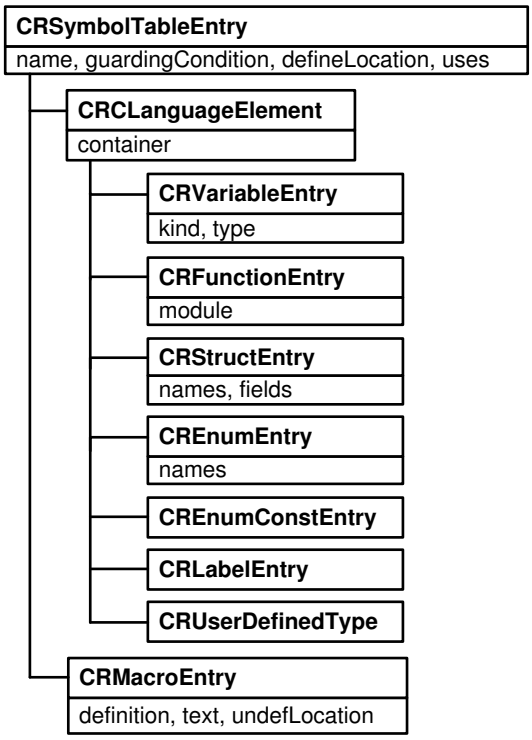

Figure 5.6: Hierarchy of CRSymbolTableEntry and attributes of each class

The '`container`' of a CRCLanguageElement is the scope in which it is defined. Section 5.3 will describe the representation of scopes. The '`kind`' of a CRVariableEntry can be "global", "local" or "extern". The '`module`' in a CRFunctionEntry is the scope created by this function (more on this in Section 5.3). The '`names`' of a CRProgramStruct or a CREnumEntry are all the type names assigned to the structure or enumerative in its definition. For example, in a definition like:

```
typedef struct S {...} Stype, *SPtrType;
```

the names that refer to the same entity are: "S", "Stype" and "SPtrType" (although the reference is indirect in the last case).

The '`guardingCondition`' assigned to a symbol table entry is the conjunction of the conditions in the tokens that make up the symbol definition. For example, for the AST that appeared in Figure 5.3, there will be two variable entries: one entry for '`const char * msg;`' with condition CppDefinedCondition(__STDC__), and another entry for '`char * msg;`' with CppTrueCondition as guarding condition.

The list of '`uses`' of a CRCLanguageElement is a list of CRSymbolLocations (pairs of $\langle filename, offset \rangle$) where the definition is referenced. A reference to a symbol that appears under a condition C_s is bound to the definition guarded by the same condition, i.e., to the definition under condition C_s. If there is no such definition, the reference is bound to a set $DefSet$ of CRCLanguageElements such that:

$$\forall\ Def \in DefSet : compatible(guardingCondition(Def), C_s)$$

This is the same binding rule used by P-Cpp to bind a macro call to its macro definition(s) (see Section 4.4).

Figure 5.7 shows some sample code and the corresponding symbol table entry for `var1`. For simplicity, locations appear as line numbers and $C1$ and $C2$ stand for some constant expressions and do not negate each other. If the pair $< C1; C2 >$ does not belong to the set of incompatible conditions of the initial configuration (Section 4.2), the uses of `var1` in line 7 and in line 9 are bound to both definitions, since

CRefactory conservatively allows every combination of conditions to be possible (in this case, $C1 \wedge C2$, $C1 \wedge \neg C2$, $\neg C1 \wedge C2$, $\neg C1 \wedge \neg C2$).

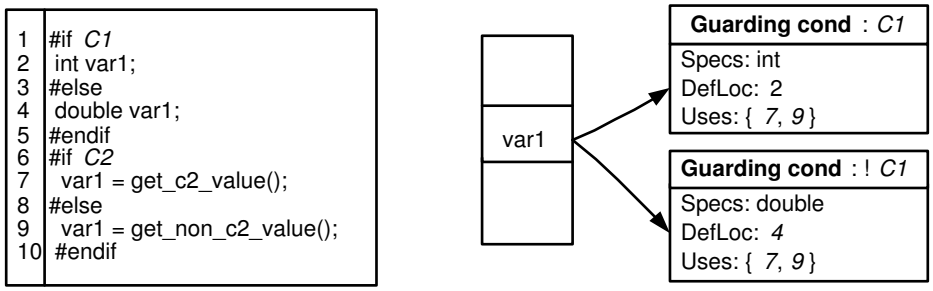

Figure 5.7: Uses of a symbol binding to more than one definition

When `#include` directives are involved, computing the set $DefSet$ of definitions that bind to a use turns more complicated, since `#include` directives change the *visibility* of definitions. We define below the rule of visibility used in CRefactory, which includes the C notion of visibility but adds to it, to handle file inclusion and conditional directives. In particular, the standard notion of visibility is **extended** by file inclusion directives and **restricted** by conditional directives.

Rule of Visibility of Definitions. A definition for a symbol N, which is located at position P inside a scope SC and under condition C_d is visible:

1. Inside SC after P. If SC is a local scope, N is not visible outside SC.

2. In inner scopes of SC that do not contain a redefinition of N. A redefinition of N is a definition of N under the same condition C_d.

3. If SC is the global scope of a file F and F is a header file, N is visible in all files that include F directly or indirectly, after the file inclusion (i.e., including a file makes visible the definitions in that file).

4. If SC is the global scope of a file F, N is visible inside the files that F includes after P (i.e., including a file makes all current definitions be visible for that file).

5. In the four previous sub-rules, visibility is restricted to the pieces of code under conditions compatible with C_d.

The Rule of Visibility derives from the Cpp implementation of `#include` as copy-and-paste, instead of a "module importation" of a language like Java or Maude. The next section will describe how the Program Repository implements the Rule of Visibility.

Let us see an example. Figure 5.8 shows three files on the left and the result of preprocessing the file inclusion with Cpp on the right.

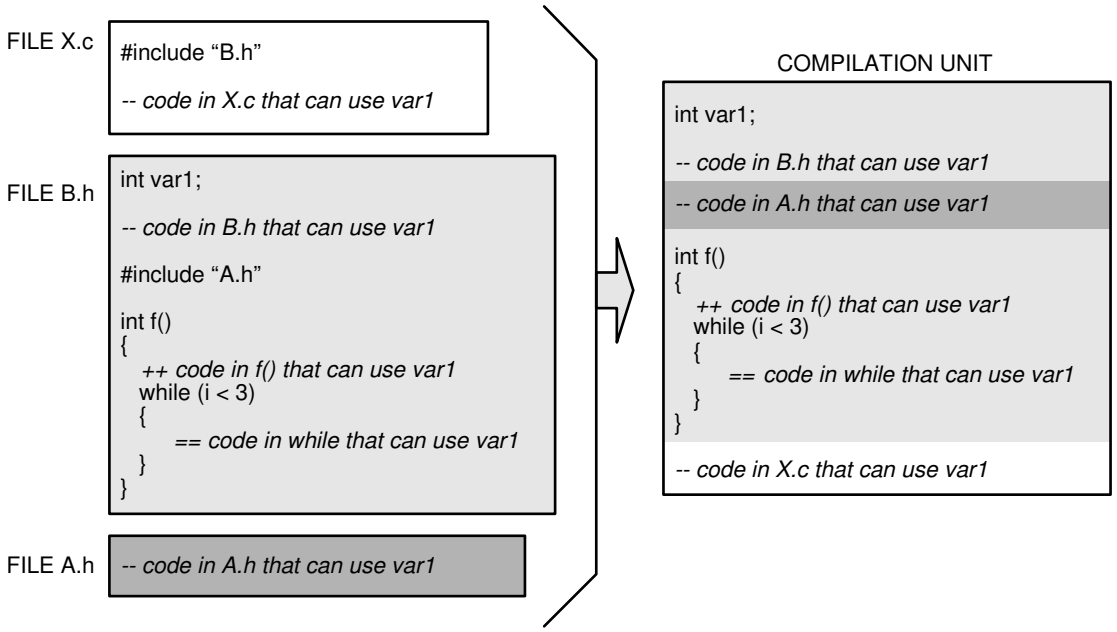

Figure 5.8: Example of visibility with #include directives

Figure 5.8 shows that after preprocessing the `#include` directives, the definition of `var1` is at the top of the compilation unit, and thus visible in:

1. In the file scope of "B.h"
2. In the inner scopes for function `f()` and the `while` block.
3. In "X.c" after the inclusion for "B.h"
4. In "A.h", which is included by "B.h" after the definition of `var1`.

5.3 The Program Repository

CRefactory's Program Repository constitutes what Roberts and Brant call "Program Database" [51], a searchable repository to analyze the preconditions of refactorings.

The class that implements the Program Repository is called CRProgramRepository. There is a single instance of this class at any time in the system, and this instance is called CProgramDB. The CProgramDB contains a reference to the initial configuration received as input (described in Section 4.2) and the Include Dependency Graph (IDG). Moreover, it has a symbol table of external symbols, i.e., global variables and functions shared between different source files [26].

The CProgramDB is populated by the ASTInfoCollector while visiting the ASTs. Inside each ProgramFile in the IDG, the ASTInfoCollector creates *modules* to represent each sub-scope. Modules index the AST in the container file and have their own symbol tables. They are represented by the class CRProgramModule. Concrete subclasses are CRProgramFile, CRProgramFunction and CRProgramBlock, that is, one class for each unit of program structure that may have variable declarations. A CRProgramBlock represents a compound statement in the C program. Figure 5.9 shows the hierarchy of CRProgramModules and the attributes defined in each class.

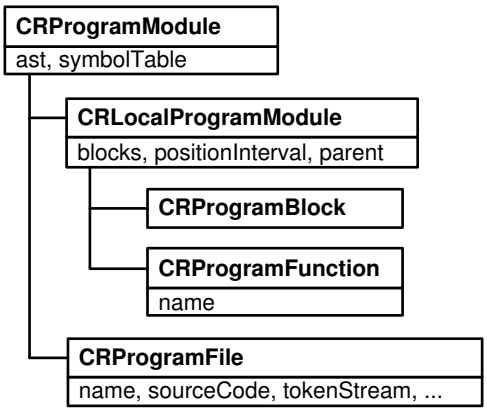

Figure 5.9: Hierarchy of CRProgramModule and attributes of each class

All CRProgramModules have an AST (which may be a reference to a sub-tree)

and a symbol table. CRProgramFunctions and CRProgramBlocks are local scopes, and have a position inside the enclosing (parent) scope. They also have a list of the inner blocks inside them. CRProgramFunctions are distinguished by the function's name. A CRProgramFile has a name, its source code, the token stream created by P-Cpp, and other attributes presented in the specification given in Chapter 4.

A CRProgramFile knows its inner CRProgramFunctions by way of its symbol table. There are actually two classes representing a function: CRFunctionEntry and CRProgramFunction. Each of them represents different aspects of a function. A CRFunctionEntry represents the function as a C symbol, an entry in the symbol table, whereas a CRProgramFunction represents the scope created by a function. A CRFunctionEntry knows its associated CRProgramFunction and acts as a *wrapper* around its CRProgramFunction, i.e., it delegates to the CRProgramFunction all requests it cannot answer [69]. Table 5.3 shows the attributes that distinguish one representation of a function from the other.

Table 5.3: Attributes that distinguish a CRFunctionEntry from a CRProgramFunction

CRFunctionEntry	name, guardingCondition, defineLocation, uses, container, isExtern, module
CRProgramFunction	name, ast, symbolTable, parent, blocks, positionInterval

The Program Repository provides answers to various queries during semantic analysis and refactoring. Some of these queries return scopes of definitions. When the scope is a file, function or block, the answer is an instance of the corresponding subclass of CRProgramModule. However, sometimes a definition affects a set of files, either because of `#include` directives or because of external definitions. In the case that the scope is a set of files, the result of the query is an instance of class CRProgramFileSet. This class is a subclass of CRProgramRepositoryElement, as it is CRProgramModule. When the scope of a refactoring is a CRProgramFileSet, the transformation affects all

files in the set and they may all get changed. The next section describes the queries that can be submitted to the Program Repository.

5.4 Queries Answered by the Program Repository

Some of the queries answered by the Program Repository are listed below. The name of each query in the list is the name of the method in class CRProgramRepository that implements it.

- **allCSymbolsForName:** `aString` **underCondition:** `aCppCond`
  startingAt: `aPgrmModule`

  This query is sent by the ASTInfoCollector when it finds a use of a symbol. The query returns all definitions of C language elements to which the use binds, following the Rule of Visibility defined in the previous section. Specifically, it returns all CRCLanguageElements with name `aString`, whose condition is compatible with `aCppCond` and that are visible in `aPgrmModule`.

  Figure 5.10 shows the pseudo-code that implements this query, called allCSymbolForName() and related subroutines.

  Thus, allCSymbolForName() works its way up from the scope where the use was found ($aPgrmModule$) until the file scope, collecting definitions for the symbol named $aString$. It removes from the set the overwritten definitions (those definitions for $aString$ under the same condition of a definition already in $defSet$). Then it invokes allCSymbolsInFile&Preds&Sucs(). This function first collects all definitions in the file and its direct and indirect predecessors in the IDG, by calling allCSymbolsInFile&Preds(). The predecessors of a file in the IDG are the files it includes. Then the function collects the definitions in the successors (files that include the current file), and their predecessors and successors recursively. Note that when including a definition from a different

```
allCSymbolsForName(aString, aCppCond, aPgrmModule, pos)
{
    scope := aPgrmModule.
    defSet := ∅.
    while (scope is not file scope) {
        col := allCSymbols(symbolTable(scope), aString, aCppCond, pos).
        Remove from col all overwritten definitions (those already in defSet).
        defSet := defSet ⋃ col.
        scope := parent scope.
    }
    defSet := defSet ⋃
              allCSymbolsInFile&Preds&Sucs(file scope, aString, aCppCond, pos).
    return defSet.
}

allCSymbolsInFile&Preds&Sucs(pFile, aString, aCppCond, pos)
{
    defSet := allCSymbolsInFile&Preds(pFile, aString, aCppCond, pos).
    succs := from allDirectSuccessors(idg, pFile) select those which include
             pFile under a condition compatible with aCppCond.
    for each suc ∈ succs {
        col := allCSymbolsInFile&Preds&Sucs(suc, aString, aCppCond, pos of include).
        for (each def ∈ col) andGuardingCondition(def, guardCond(edge to suc)).
        defSet := defSet ⋃ col.
    }
    return defSet.
}

allCSymbolsInFile&Preds(pFile, aString, aCppCond, pos)
{
    defSet := allCSymbols(symbolTable(pFile), aString, aCppCond, pos).
    preds := from allDirectPredecessors(idg, pFile) select those included before pos
             and under a condition compatible with aCppCond.
    for each pre ∈ preds {
        col := allCSymbolsInFile&Preds(pre, aString, aCppCond, end-of-file(pre)).
        for (each def ∈ col) andGuardingCondition(def, guardCond(edge from pre)).
        defSet := defSet ⋃ col.
    }
    return defSet.
}
```

Figure 5.10: Pseudo-code for finding definitions binding to a use

file, the condition of that definition is the conjunction of its condition in the symbol table of the file, with the condition of the file inclusion (which is in the edges of the path to or from the file).

- **fileExistsInIDG:** `aFilename`

  All refactorings require as input some selection from the user, which is represented as a pair $< aFilename, anInterval >$. This query is sent while checking the preconditions of refactoring, to validate the user input, in particular that $aFilename$ is a real filename of the loaded program.

- **containsReferenceTo:** `aString` **interval:** `anInterval` **inFile:** `aFilename`

  This query is also sent with the previous one in the preconditions of most refactorings, to check the remaining input from the user. For example, the input to Rename Variable includes the name of the variable to rename and the pair $< aFilename, anInterval >$. This query checks that the location pair actually contains a reference to the variable to rename (in parameter `aString`).

- **fileIsWriteable:** `aFilename`

  The answer to this query is needed by every refactoring, to check that the file or files affected by the transformation were not specified at load time as read-only. This prevents changes in a library header file.

- **isUniqueDefinitionOfVariable**: `varName` **inFile**: `aFilename` **inInterval**: `anInterval`

  This query is used to check if the refactoring will affect a single definition of a variable with name `varName` that the user has selected at the location specified by `aFilename` and `anInterval`. A refactoring affects a single definition of `varName` when there is only one definition of `varName` as a variable that reaches the specified location, or when there are multiple definitions of `varName` but the user-selected one does not share any uses with the other definitions (i.e., the set

of uses does not intersect with the other sets of uses).

Moreover, the location specified by `aFilename` and `anInterval` may be in the variable definition or in a use of the variable. When there are multiple definitions of `varName`, if the location is in the variable definition, this query checks that the definition does not share uses with the other definitions of `varName` that are visible in the scope. However, if the location specified in the parameters represents a use of the variable, the query checks that the selected use binds to a single definition, from all definitions of `varName` visible in the scope.

Similar queries exist for other kinds of symbol table entries.

- **isDefinitionOfSymbol:** `sName` **visibleInFile:** `aFileName` **at:** `pos`

  This query returns true if there is a definition for `sName` as a C entity or a macro, that reaches or is visible at position `pos` in `aFileName`. The result of this query is used by renaming refactorings to check that the new name chosen by the user does not overwrite or collide with the name of any other defined entity under any program configurations.

- **definitionOfVariable**: `varName` **inFile:** `aFilename` **at:** `pos`

  Once the preconditions of a refactoring checked that there is a single definition of `varName` reaching the location specified by `aFilename` and `pos`, this query returns that single definition. That is, it returns a CRVariableEntry for `varName` that is either defined or used at the specified location.

  Similar queries exist for other kinds of symbol table entries.

- **allVariablesNamed**: `varName` **visibleIn**: `scope`

  When there is more than one definition of a variable, this query is used instead of the previous one to retrieve the collection of CRVariableEntrys with name `varName` that are visible in `scope`. Similar queries exist for other kinds of symbol table entries.

- **contextForPosition:** `pos` **inFile:** `aFilename`

  It returns the innermost CRProgramModule inside file `aFilename` that contains `pos`. That is, if `pos` is inside a block, it returns the corresponding CRProgramBlock. If `pos` is not inside a block, but it is inside a function, it returns the corresponding CRProgramFunction. Otherwise, it returns the CRProgramFile with name `aFilename`. This query is used in the analysis of many refactoring, to obtain the scope of the user selection.

- **scopeAffectedBy**: `anSTEntry`

  This query is sent by several refactorings, to get the scope affected by a change to `anSTEntry`. In other words, this query calculates the visibility of the definition represented by `anSTEntry` and returns an instance of any of the subclasses of CRProgramRepositoryElement where the definition is visible.

- **unionOfScopesAffectedBy**: `anSTEntryCol`

  When a refactoring affects more than one definition, this query is sent to calculate the whole scope affected by the refactoring.

  The union of two scopes is calculated in the following way: if one scope is nested in the other, the result is the outer scope; if both scopes are in the same file but do not intersect, the result is the whole file; if the scopes belong to different files, the result is a CRProgramFileSet with the files that contain the scopes.

- **changeProgramElement:** `scope` **fromAST:** `ast` **withChange:** `aChange`

  After a refactoring transforms the AST(s) of the affected scope, a 'Change object' is created to update the Program Repository. Change objects belong to the hierarchy of CRefactoryChange. There is one subclass in this hierarchy for each kind of refactoring (see next section). Change objects send this query to the Program Repository with the affected `scope`, the transformed `ast` and itself as arguments. This query pretty-prints the `ast` back to source code, processes

the source code and gets a new AST with updated positions for all tokens, assigns the new AST to the `scope`, and finally updates the symbol tables with the information of the new AST. If `scope` is a local scope, it is necessary to process and update the whole file.

5.5 Updating the Program Model

Refactorings are executed by first checking their preconditions and then updating the Program Model. Preconditions are checked by querying the Program Repository about specific properties. The update to the Program Model is performed in two steps:

1. *The ASTs of the affected scope are transformed.*

   In many cases, a 'Parse Tree Rewriter' is created to search for a piece of code in a tree and replace it with a new subtree (like in Rename Variable). In other cases, a 'Parse Tree Searcher' is used to search for certain subtree which is then manually replaced or deleted (like in Delete Variable). Both 'Parse Tree Searcher' and 'Parse Tree Rewriter' work as "Visitors" of the AST [69].

2. *Change objects are created to update the Program Repository.*

   'Change objects' are instances of the classes in the hierarchy of CRefactoryChange. There is one subclass in this hierarchy for each kind of refactoring. Some 'change' objects only update the symbol table (like CRenameChange) while others receive a transformed AST and invoke the updating query in the Program Repository "changeProgramElement: fromAST: withChange:".

Depending on the class of change object, there are different strategies to recreate the symbol table. Specifically, there is a different AST visitor associated with different classes of change objects. For example, CDeleteUnreferenceVariableChange uses a

CRASTInfoModifier to visit the new AST, whereas CMoveVariableToStructureChange uses a CRASTInfoModifierRecreatingDeclarations.

The rest of this section is devoted to describe the different classes of 'Visitors' of the AST, used by CRefactoring throughout all stages. They are subclasses of CRProgramNodeVisitor. The first two classes are used when a program is first loaded, the following two during refactoring and the rest of them are used to update the Program Model.

CRASTInfoCollector. After a program has been pseudo-preprocessed and parsed for the first time, a CRASTInfoCollector visits the AST of each file and populates the Program Repository by creating modules to represent nested scopes inside each file (CRProgramFunctions and CRProgramBlocks). The CRASTInfoCollector also creates the entries for CRCLanguageElements in the corresponding symbol tables.

CRASTUsesModifier. Once the symbol tables have been populated with definitions of all program entities, a CRASTUsesModifier is created to revisit all ASTs and recalculate the binding of uses to definitions. Uses may bind to more global definitions at this time when all definitions visible from all include paths are available.

CRParseTreeSearcher. This class is used during refactoring. It uses one or more CRSearchRules to find the first subtree matching certain conditions given in the rules. A CRSearchRule can take some context information to find a match. A CRParseTreeSearcher also has the ability to perform a given set of actions on the matched node.

CRParseTreeSearchAll. This is a subclass of CRParseTreeSearcher used to find all matching subtrees instead of just the first one. For example, during the refactoring 'Delete Unreferenced Variable', a CRParseTreeSearchAll is used to

find all declarations for the symbol to be deleted, since there may be more than one declaration under different configurations.

CRParseTreeRewriter. This is also a subclass of CRParseTreeSearcher. This class can not only search for subtrees but also replace the matching subtrees by a new one, following a CRReplaceRule. For example, all renaming refactorings use a CRParseTreeRewriter to replace the old Identifier node for another one with the new name.

CRFormatter. When a change object requests the Program Repository to update a given scope with a transformed AST, the Program Repository first creates an instance of CRFormatter to pretty-print the AST. The CRFormatter reads the token labels on leaf nodes and is able to reverse macro expansion and conditional completion based on those labels. The CRFormatter attempts to do exact pretty-printing when possible.

CRASTInfoModifier. After the Program Repository requests the CRFormatter to pretty-print the transformed AST, the new source code is parsed and a new AST gets built. The next step is to create an AST visitor to update the symbol tables and program modules. When the positions of symbols and modules is the only necessary update, the new AST is visited by a CRASTInfoModifier.

CRASTInfoModifierRecreatingDeclarations. Any refactoring that changes declarations (by adding or removing them) requires an instance of this class to revisit the changed ASTs and reconstruct the symbol tables.

Chapter 6

Applying Refactoring

This chapter describes how CRefactory uses its Program Model to execute refactorings. Refactorings are executed by first checking their preconditions, then manipulating the nodes in the AST and finally updating the symbol tables. Preconditions are checked by querying the Program Model about specific properties. AST manipulation is performed by the Parse Tree Rewriter, which visits subtrees and replaces them. Change objects update the symbol tables.

Incorporating the ability to handle preprocessor directives in a refactoring tool is significantly difficult. File inclusion directives extend the scope of program entities. Macros can be defined and undefined anywhere, their bodies may reference global program entities and they may have hidden references to entities by using concatenation. Conditional directives can create multiple definitions of a program entity, as described in previous chapters.

The first three sections of this chapter discuss in detail the circumstances in which we found file inclusion, macros and conditional directives respectively to cause problems or violate correctness during refactoring. To solve these problems, new preconditions and execution rules must be provided for each refactoring, so that behavior is still preserved when preprocessor directives are present. The fourth section presents the enhanced preconditions and transformation steps of a few refactorings. The last

section presents new refactorings that apply to Cpp directives.

Although a formal proof of correctness of the proposed preconditions and execution rules is out of the scope of this work, we carefully describe how these rules handle the problems introduced by Cpp directives.

6.1 Handling File Inclusion During Refactoring

The `#include` directive extends the scope of program entities and thus, the refactorings performed in one file may need to be spread to several files. Because of how Cpp implements file inclusion, a change in a file not only spreads to the files that include it but also to the files included by it (as explained in Chapter 5).

Therefore, changing a program entity correctly in the presence of `#include` directives depends on calculating the exact scope of the refactoring, i.e., the visibility of the program entity. This is solved at the level of the Program Repository, following the Visibility Rule defined in Chapter 5.

Another issue with file inclusion is that the user may choose to change a program entity that is defined in a library file, i.e., a header file which is not supposed or it is not possible to change. For example, a user should not be able to rename the function "getc()" defined in "stdio.h", because that will break all other programs that use that function.

This issue is solved by having the user list the read-only directories in the input to CRefactory (see Section 4.2). Then, individual refactorings should check in their preconditions that the scope of refactoring does not include a file that belongs to a read-only directory.

6.2 Handling Macros During Refactoring

While conditional directives are the hardest to handle at processing time (when the program is being loaded), macros are the hardest to handle at refactoring time, i.e., there are many problems raised by macros that may invalidate a refactoring.

The first five subsections discuss issues that appear from refactoring C code that has macro calls. Most of these issues cannot be handled by the preconditions of the refactorings but are instead analyzed during the transformation of the AST. If a problem is found at that point that invalidates the refactoring, the changes are rolled back.

The last three subsections talk about the problems of refactoring macro definitions.

6.2.1 Scope of refactoring

When a refactoring is applied to a C language entity, macro definitions may change if their body refers to that entity. Therefore, the scope of refactoring must be extended with the body of all macro definitions that are called in that scope.

Conversely, if there is a macro definition in the scope of refactoring but there is no call to this macro, its body is not included in the refactoring because it is not possible to recognize if the macro body is actually referencing the changing entity.

This issue is handled during the transformation itself, by checking on the AST nodes that require changes, if they have a macro call label (i.e., if they come from macro expansion). If that is the case, a change is required either on the macro definition (if its body refers to the changing entity) or on the macro call (if the changing entity was an argument of the macro call).

6.2.2 Different contexts calling the same macro

There may be problems when a macro definition refers directly to C program entities, i.e., when instead of using parameters, the macro has unbound references inside its body. One of the problems that this may cause happens when a refactoring is applied to a program entity E, and the body of a macro M is affected by the refactoring (because M had a reference to E is its body). If there are calls to M from different scopes, and each scope has a different definition of E, it is not possible to apply refactoring to a single definition of E.

Figure 6.1 shows an example with two functions, `f1()` and `main()`, each one declaring a variable '`errstatus`' locally and calling macro '`ER1`', which refers to `errstatus`. Each macro expansion will refer to a different `errstatus`. If the variable `errstatus` defined in `f1()` is renamed but the body of macro `ER1` is not modified, the call to `ER1` in `f1()` will cause an error of undefined variable in the next compile. On the contrary, if the macro body is modified renaming `errstatus`, the call to `ER1` in function `main()` will cause the error of undefined variable.

```
#define ER1   errstatus = 1

int f1() {
  int errstatus;
  ...
  if (bottom < 0)
      ER1;
  ...
}

int main() {
  int errstatus;
  ...
  if (input == 0)
      ER1;
  ...
}
```

Figure 6.1: Macro referring to different definitions of a variable

Therefore, when a refactoring is applied to a program entity E, and the body of a macro M is affected by the refactoring (because M had a reference to E in its body), the scopes of all other calls to M in the entire program should be inspected. If there

is another scope in which M is called that has a different definition of E, then the refactoring cannot proceed.

6.2.3 A macro referring to different uses of the same name

Another problem caused by macros that refer directly to program entities is that different calls to the macro may refer to entities with the same name but in different *name spaces*. In C, the same name may refer to different entities, even in the same scope and under the same guarding condition, if the references are in different name spaces [22]. The name spaces in C are:

1. statement labels;
2. structure, union and enumeration tags;
3. fields of structures and unions;
4. all other names (variables, functions, user-defined types and enumeration constants).

For example, Figure 6.2 shows a variation of the code in Figure 6.1, where there is a structure declaration for '`Buffer`' that has '`errstatus`' as a field. The function `f1()` declares a variable `buff` of type `Buffer` and calls macro `ER1` twice. In the first call, `ER1` refers to the *variable* `errstatus`, while in the second call, `ER1` refers to the *field* `errstatus` of structure `buff`. Therefore, if for example variable `errstatus` is renamed, changing `ER1` would give an error in the second macro call, and leaving it unchanged would give an error in the first macro call. The refactoring is unsafe.

Therefore, when a refactoring is applied to a program entity E, and the body of a macro M is affected by the refactoring (because M had a reference to E in its body), all other calls to M in the entire program should be inspected. If any call to M refers to a different use of E, i.e., to a different overloading class of E, then the refactoring cannot proceed.

```
#define ER1   errstatus = 1

typedef struct {
   ...
   int errstatus;
} Buffer;

int f1() {
  int errstatus;
  Buffer buff;
  ...
  if (bottom < 0)
      ER1;
  ...
  if (nelems > N)
     buff.ER1;
  ...
}
```

Figure 6.2: Macro referring to different uses of a name

6.2.4 Use of concatenation in macro bodies

There are two special operators that can be used in the replacement text of macros: '#' and '##'. We did not find any instance in which '#' would cause a violation of correctness in refactoring. However, the operator '##' can cause a replacement text to refer indirectly to a C language entity by concatenating two tokens.

Therefore, when renaming is applied to an entity E, and there is a macro M called in the scope of refactoring that refers to E indirectly through concatenation of substrings of the name of E, the refactoring cannot proceed. The reason is that there is no safe way of modifying M so that it refers to the new name of E without affecting other calls to M and other results of the concatenation. Figure 6.3 shows an example where variable `errstatus` cannot be renamed.

```
#define ST(VAR)  VAR##Status

int main() {
   int errStatus;
   ...
   switch (x)
      case 0: ST(complete) = 1;
      case 1: ST(err) = 1;
  ...
}
```

Figure 6.3: Macro using concatenation prevents renaming

6.2.5 Macros affecting code movement

Macros can be defined, undefined and redefined at any point. Therefore, when a piece of code is moved, for example during Extract Function, the refactoring preconditions must include checking that for all the macros called from that piece of code, the definitions are the same in the new location, i.e., the binding of calls to macro definitions remains the same.

The next three sections describe problems during refactoring of macro definitions. Although the refactorings on macro definitions are similar to those that apply for function definitions, there are important differences as outlined below.

6.2.6 Scope of refactoring of a macro definition

The scope for any refactoring on the signature of a function is generally global to the application. Instead, macros can be undefined through `#undef` directives, which restricts the scope of refactoring to the code from the `#define` up to the `#undef`. Some compilers do not even require a `#undef` before a new definition of a given macro.

The Program Repository takes care of computing the correct scope for a macro (see Section 5.3).

6.2.7 Replacement of macro parameters

In the semantics of the C language, the definition of a function parameter binds to all uses of parameter name that are in the same name-space, e.g., it does not bind to a structure field with the same name.

The semantics of Cpp with macro parameters is different. All instances of the name of a macro parameter in the body of the macro are bound to the parameter.

Cpp does not know about name-spaces, it just performs string substitution. For example, in a macro definition like:

```
#define M1(x, st)    x = st.x
```

both instances of '`x`' in the macro body are bound to the macro parameter '`x`'.

6.2.8 Parentheses around macro arguments

When a piece of code is replaced by a call to a macro (in refactorings like 'Extract Macro' or 'Replace Code with Macro Call'), errors can be caused by the presence or absence of parentheses around a macro parameter in the body of the macro, or around an argument in the macro call. For example, Figure 6.4 shows a definition for a macro `SQUARE` and an expression below it that is to be replaced by a call to `SQUARE`. The correct substitution is the one that encloses the macro argument with extra parentheses. Behavior would not be preserved otherwise.

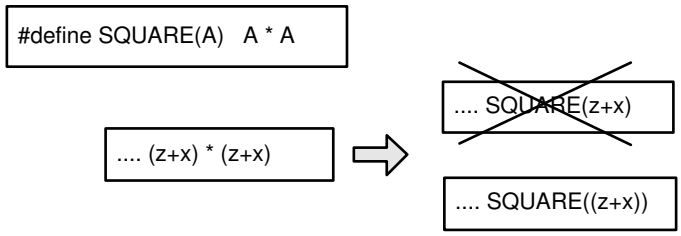

Figure 6.4: Parenthesis needed around a macro argument to preserve behavior

Unfortunately, the problem cannot always be solved by adding extra parentheses to the arguments in the macro call. Sometimes the presence of parentheses in the body of the macro can make the refactoring 'Replace Code with Macro Call' to fail. Figure 6.5 shows a case in which the piece of code on the right of the figure cannot be replaced by a call to the macro on the left, because a statement cannot be in between parentheses (it would cause a compiler error).

Moreover, replacing a piece of code by a call to a macro may not only cause a compiler error but may also change behavior. An example is given in Figure 6.6. If

Figure 6.5: Presence of parenthesis prevents substitution with macro call

the expression '`*b++`' is replaced by a call '`INC(*b)`', the expansion of the macro call would yield `(*b)++`, which increments the contents pointed to by `b` instead of the address of `b` as originally intended.

Figure 6.6: Example where introducing a macro call changes program behavior

Last but not least, the presence of a comma in a piece of code can make it impossible to replace the code by a macro call. Figure 6.7 shows a definition for macro `DECL(X)` on the left and a declaration on the right that cannot be replaced by a call to `DECL(X)`. The reason is that, if parenthesis are not added to the argument in the macro call, preprocessing would fail since it would appear as a call with two arguments. Conversely, if parenthesis are added to the argument in the macro call, the expansion of the macro would yield a compiler error.

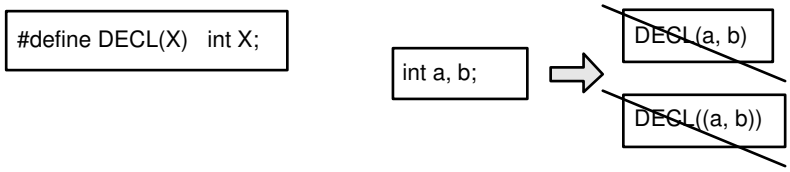

Figure 6.7: Presence of comma prevents substitution with macro call

6.3 Handling Conditional Directives During Refactoring

This section presents two issues that arise during refactoring of C code with conditional directives. As we have stated before, a refactoring is considered correct if and only if it is correct for all possible system configurations.

6.3.1 Multiple definitions for a program entity

When a program entity has multiple definitions under different configurations, some refactorings may apply to a single definition while other must be applied to all definitions to preserve behavior, depending on the type of refactoring and on the degree of intersection between definitions.

On the one hand, there are refactorings like 'Replace Type' that make sense when applied to a single definition. On the other hand, a refactoring like 'Rename' can only be applied to a single definition if the uses of the definition do not bind to other definitions, i.e., if there is no intersection between the set of uses of different definitions. Yet there are other refactorings like 'Delete Unreferenced Variable', which apply to a subset of definitions of the variable, all those that have no uses.

6.3.2 Conditionals affecting code movement

When moving code to a new function or a new file, there are two important considerations:

1. If the code being moved includes one or more conditional directives, the code must include the whole Cpp conditional construct(s) to which the individual directives belong.

2. The code being moved should be placed under the same condition that it was before.

For example, if in the code of Figure 6.8, a user selects lines 6 to 9 to be extracted into a new function, it would break the Cpp conditional construct that starts in line 7 and ends in line 12. On the other hand, if the user selects lines 8 and 9, the new function must be enclosed by Cpp conditional with a branch for '`defined _X1`' surrounding those two lines. Note that variable `q` is only defined if '`defined _X1`'.

```
1  #ifdef _X1
2    int q;
3  #endif
4
5  int f1() {
6     nelems++;
7  #ifdef _X1
8     q+= j;
9     nelems-= q;
10 #else
11    nelems*= j;
12 #endif
13 }
```

Figure 6.8: Extract function with conditional directives

6.4 Refactorings on C Code

In a previous work [16], we have proposed a catalog of refactorings applicable to C code. The refactorings are grouped into four major categories. Table 6.1 shows the list of refactorings with minor changes to the original version. The description of each refactoring can be found in [16].

The next three subsections describe precisely the enhanced preconditions and transformation steps of three representative refactorings in the catalog. Afterwards, some ideas are presented on how to handle the other refactorings in the catalog.

Table 6.1: Refactorings for the C language

1.Adding a program entity	Add variable Add parameter to function Add typedef definition Add field to structure Add pointer to variable
2.Removing a program entity	Delete unreferenced variable Delete unreferenced parameter Delete unreferenced function
3.Changing a program entity	Rename variable Rename user-defined type Rename structure field Rename function Replace type Contract variable scope Extend variable scope Replace value with constant Replace expression with variable Change to pointer Change from pointer Convert global variable into parameter Reorder function arguments
4.Complex refactorings	Group variables in new structure Move variable into structure Extract function Inline function Consolidate conditional expression For into while While into for

6.4.1 Delete unreferenced variable

After a program has been changed a number of times, some variables may become unused. However, programmers may resist deleting their definitions if they are unsure whether the variable is used in some particular configuration. This refactoring allows deleting only those definitions of the variable that have no uses under any possible configuration.

Input values:

1. The name of the variable to be deleted (`varName`).

2. The position interval (`interval`) in a file (`filename`) that contains the user selected reference to `varName`.

Preconditions:

Opdyke [2] describes the precondition of this refactoring as:

$$referencesTo(varName) = \emptyset$$

When multiple configurations of a program are considered, `varName` can have more than one definition. CRefactory allows deleting those definitions of `varName` that have no references, even when other definitions of `varName` are used in other configurations. In terms of CRefactory's analysis functions, this precondition is expressed as shown in Figure 6.9.

```
preconditions
  scopeOfSelection := CProgramDB contextForPosition: interval first inFile: filename.
  ^(((CProgramDB fileExistsInIDG: filename)
   and: [CProgramDB fileIsWriteable: filename])
    and: [CProgramDB containsReferenceTo: varName interval: interval inFile: filename])
     and: [(CProgramDB allVariablesNamed: varName visibleIn: scopeOfSelection)
                      contains: [:entry | entry uses isEmpty]
```

Figure 6.9: Preconditions of Delete Variable

Figure 6.9 shows the Smalltalk code for method `preconditions` in class CDelete-VariableRefactoring. The first line of the method assigns to the instance variable `scopeOfSelection` the result of sending the message '`contextForPosition:inFile:`' to the object in CProgramDB (the only instance of class CRProgramRepository). The first argument of the message is '`interval first`' and the second argument is '`filename`'. The '`^`' in the following line returns the result of the statement that follows, which will be a Boolean value. That Boolean value is obtained by sending the subsequent messages to the CProgramDB and applying the conjunction of the results

with the message `and:`. The message `and:` expects a *block* as argument. A block in Smalltalk is a closure, a piece of code with deferred execution, specified between square brackets. A block may have parameters, specified between ':' and '|' inside the block. An example is the parameter `entry` in the block that is the argument to the message `contains:` in the last line. That parameter will take the value of each element in the collection returned by the message '`allVariablesNamed:visibleIn:`' of the previous line. The message `contains:` evaluates the block on each element and returns `true` if there is any element that complies with the block. In this case, it will return `true` if there is a definition of `varName` as a variable that has no uses.

Transformation:

If `varName` has more than one definition under different branches of a Cpp conditional, the refactoring must check which of those definitions have no references and can be safely removed. Figure 6.10 shows the pseudo-code for the mechanics of this refactoring.

```
performRefactoring
   defCol := (CProgramDB allVariablesNamed: varName visibleIn: scopeOfSelection)
                  select: [:entry | entry uses isEmpty].
   guardingConditions := defCol collect: [:def | def guardingCondition].
   scope := CProgramDB unionOfScopesAffectedBy: defCol.
   self delete: (self getAllDeclaratorNodes).
   self createChangeObjects

delete: allDeclarators
   allDeclarators do:
      [: declarator |
       self deleteDeclaratorOrDeclaration: declarator]
```

Figure 6.10: Performing Delete Variable

The refactoring first gathers in `defCol` all definitions of `varName` to be deleted. The next statement returns the collection of guarding conditions of each definition in `defCol`. The scope affected by the refactoring is then calculated by sending the message `unionOfScopesAffectedBy:` to the CProgramDB (see Section 5.4). The keyword `self` in the next line refers to the same receiver of the current message, that is, the instance of CDeleteVariableRefactoring that is being performed. Method

`getAllDeclaratorNodes` is first invoked to find all declarators for `varName` in the scope, and this result becomes the argument of the message `delete:`. Method `delete:` iterates through all the declarators for `varName` and deletes each one by calling `deleteDeclaratorOrDeclaration:`, which may just delete the declarator or, if it was the only one in the declaration, delete the whole declaration.

To find all declarators for `varName`, method `getAllDeclaratorNodes` constructs a CRParseTreeSearchAll that, given an AST, is able to find all declarators of `varName` under any of the conditions in the set `guardingConditions`. The CRParseTreeSearchAll is then requested to find all declarators in each of the ASTs of the scope.

Going back to `performRefactoring`, after the ASTs have been replaced, the message `createChangeObjects` creates an instance of CDeleteVariableChange to update the Program Repository. This change object will first remove the corresponding definitions of `varName` from the symbol tables and then create a CRASTInfoModifier to modify the positions of every other element in the Program Repository.

This refactoring may leave a Cpp conditional branch empty, if the deleted definition or declaration of `varName` was the only thing in the branch. We leave the decision to the user to apply the refactoring "Delete empty branches of Cpp conditional", which, given a Cpp conditional, removes the branches that are left empty.

6.4.2 Rename variable

Renaming a variable is probably the best known and used refactoring. In CRefactory, there are two versions of Rename Variable: *Single Rename* and the *Multiple Rename*. If the user chooses the Single Rename, it means she wants to rename a *single* definition of the variable. If instead she chooses the Multiple Rename, the refactoring will rename all definitions of the variable that are compatible with the user selection (as previously, the use of the word "compatible" here means compatible guarding conditions).

Input values:

1. The name of the variable to be renamed (`oldName`).

2. The position interval (`interval`) in a file (`filename`) that contains the user-selected reference to `oldName`. This reference may be a use of the variable or part of the variable definition.

3. The new name for the variable (`newName`).

Preconditions:

Figure 6.11 shows the preconditions for the Single Rename and Multiple Rename versions of Rename Variable.

```
preconditionsSingleRename
  ((((CProgramDB fileExistsInIDG: filename) and: [CProgramDB fileIsWriteable: filename])
  and: [CProgramDB containsReferenceTo: oldName interval: interval inFile: filename])
  and: [CProgramDB isUniqueDefinitionOfVariable: oldName inFile: filename inInterval: interval])
  and: [(CProgramDB isDefinitionOfSymbol: newName visibleInFile: filename at: interval first) not]

preconditionsMultipleRename
  (((CProgramDB fileExistsInIDG: filename) and: [CProgramDB fileIsWriteable: filename])
  and: [CProgramDB containsReferenceTo: oldName interval: interval inFile: filename])
  and: [(CProgramDB isDefinitionOfSymbol: newName visibleInFile: filename at: interval first) not]
```

Figure 6.11: Preconditions of Rename Variable

The first two lines of the preconditions are the same in both cases, and are self explanatory. The last line is also the same in both cases, and represents the standard precondition of Rename Variable, which is that the new name does not clash with any other symbol in the scope [2].

The added precondition in the case of Single Rename is that the definition of `oldName` that the user selected is unique. What this means is that either:

- there is a single definition of `oldName` in the scope, or

- there are multiple definitions of `oldName` but the selected definition for rename does not share any uses with the other definitions (i.e., the set of uses does not intersect with the other sets of uses).

The issues discussed in Section 6.2 that appear with macros are not addressed in the preconditions but during the transformation, as it is described in the next subsection. If the refactoring is found to be incorrect at that point, changes made so far are rolled back.

Transformation:

Figure 6.12 shows the code for method `performRefactoring` in the case of Single Rename.

```
performRefactoring
  selectedDef := CProgramDB definitionOfVariable: oldName inFile: filename at: interval first.
  guardingConditions := OrderedCollection with: selectedDef guardingCondition.
  scope := CProgramDB scopeAffectedBy: selectedDef.
  self renameIn: scope.
  self checkAllMacroCallsChanged.
  self createChangeObjectsAndReplaceMacro

renameIn: aPgrmRepElem
  | astReplacer |
  astReplacer := self replacer.
  aPgrmRepElem isFileSet
    ifFalse: [astReplacer executeTree: aPgrmRepElem ast]
    ifTrue: [aPgrmRepElem files do: [:file| astReplacer executeTree: file ast]].

replacer
  ^(CRParseTreeRewriter new)
      replace: oldName
      withValueFrom:
        [:aNode |
        | macroCall |
        aNode token isMacroDerived
          ifTrue:
             [((macroCall := aNode token macroCall) argumentsReferTo: oldName)
                  ifTrue: [macroCall replaceReferenceInArg: oldName with: newName]
                  ifFalse:
                    [(macroCall anyDefinitionUsesConcatOn: oldName)
                        ifTrue:
                            [^self refactoringErrorNeedsRollback: 'Macro uses ## on ',oldName].
                     self addChangedMacro: macroCall.
                     aNode]]
          ifFalse:
             [CRIdentifierNode value: ((aNode token) value: newName; start: nil)]].
      when:
        [:aNode |
        (aNode isField not and: [aNode isLabel not])
            and: [selectedDef isInInterval: (aNode startPosition to: aNode stopPosition)]];
      avoidRedefinitionsOf: oldName
```

Figure 6.12: Performing Rename Variable

The preconditions already checked that there is a unique definition of `oldName` selected by the user, so the first step in `performRefactoring` is to get this defini-

tion. The guarding condition of the selected definition is stored and then it calculates the affected scope. Method `renameIn:` is called with the affected scope as parameter. That method creates a "replacer" of the AST (returned by method `replacer`) and, depending on whether there is a single AST or more, it sends the message `executeTree:` to the replacer on the single AST of the scope or on each of the ASTs, which will perform the changes. The next step in `performRefactoring` is to check that the replacer visited all calls to the macros that needed changes, thus addressing the problems described in Section 6.2.2 and Section 6.2.3. If there were unvisited macro calls, an error is raised declaring the refactoring as invalid and changes to the ASTs are rolled back. Otherwise, change objects are created in the last step to change the affected macros definitions and update the Program Repository from the new ASTs. The change objects associated with Rename Variable are instances of CRenameVariableChange.

In the `replacer` method the parameter to the keyword "`when:`" is a block that tells which nodes should be visited. In this case, those representing an identifier for `oldName` that is not a a field of a structure nor a label, and that represent a reference to the selected definition. The parameter to the keyword "`withValueFrom:`" is a block that tells how are nodes replaced. If the node derives from a macro, it first checks if it was an argument of the macro. Otherwise, it checks if `oldName` comes from the use of concatenation in the macro body, and in this case the refactoring is aborted and all changes performed to the ASTs so far are rolled back. This addresses the issues with macros discussed in Section 6.2.

The case for Multiple Rename is very similar, although there will not be a "selected definition" but a set of them, and the affected scope will be the union of scopes, like in the implementation of Delete Variable.

6.4.3 Move variable into structure

A variable defined outside any structure is moved inside one, so that it becomes a field of the structure. This refactoring is useful when creating a structure out of global variables. The next steps are to add a pointer reference to this structure and to pass the pointer as argument to the functions, thus reducing the use of global variables in the program.

Input values:

1. The name of the variable to be moved (`varName`).
2. The interval (`varInterval`) in a file (`varFile`) that contains the user-selected reference to `varName`.
3. The name of the structure (`structName`) where the variable is to be moved.

Preconditions:

Figure 6.13 shows the preconditions for this refactoring. Once the input values are checked as in previous refactorings, the preconditions are split in two cases: 1. There is a single definition of `varName`, and 2. There are multiple definitions of `varName`.

In the case when there is a single definition of the variable, there must also be a single definition of the structure, and they either must have the same guarding condition or the guarding condition of the structure must be a TrueCondition. Moreover, message `checkNotFieldAndInnerScopeOf:in:` checks that `varName` is not already a field in the structure and that its scope is included in the scope of the structure. The message `checkStructVarForEachVarUse` checks that at each use of `varName`, there is a way to refer to an instance of the structure so that:

$$\begin{aligned}\forall u_i \in uses(defVar) : \exists SVar : (refers(type(SVar) = defSt)) \\ \wedge (guardingCondition(SVar) = guardingCondition(u_i)) \\ \wedge (scope(SVar) \supseteq scope(defVar))\end{aligned}$$

```
preconditions
   scopeOfSelection := CProgramDB contextForPosition: varInterval first inFile: varFile.
   ((((CProgramDB fileExistsInIDG: varFile) and: [CProgramDB fileIsWriteable: varFile])
    and: [CProgramDB containsReferenceTo: varName interval: varInterval inFile: varFile])
    and: [(CProgramDB allStructsNamed: structName visibleIn: scopeOfSelection) notEmpty])
        ifFalse: [^false].
   (CProgramDB isUniqueDefinitionOfVariable: varName inFile: varFile inInterval: varInterval)
        ifTrue: [^self preconditionsSingleDefVar]
        ifFalse: [^self preconditionsMultDefsVar]

preconditionsSingleDefVar
   | defVar defSt |
   (CProgramDB isUniqueDefinitionOfStruct: structName visibleIn: scopeOfSelection)
      ifFalse: [^false].
   defVar := CProgramDB definitionOfVariable: varName inFile: varFile at: varInterval first.
   defSt := CProgramDB definitionOfStruct: structName visibleIn: scopeOfSelection.
   ^((defVar guardingCondition = defSt guardingCondition) or: [defSt guardingCondition isTrue])
     and: [self checkNotFieldAndInnerScopeOf: defVar in: defSt]

preconditionsMultDefsVar
   | defsVar defSt defsSt |
   defsVar := CProgramDB allVariablesNamed: varName visibleIn: scopeOfSelection.
   (CProgramDB isUniqueDefinitionOfStruct: structName visibleIn: scopeOfSelection)
      ifTrue:
         [defSt := CProgramDB definitionOfStruct: structName visibleIn: scopeOfSelection.
         ^(defSt guardingCondition isTrue) and: [defsVar do:
            [: eachVar | (self checkNotFieldAndInnerScopeOf: eachVar in: defSt)
                           ifFalse: [^false]]]]
      ifFalse:
         [defsSt := CProgramDB allStructsNamed: structName visibleIn: scopeOfSelection.
         ^(defsVar size = defsSt size) and: [defsVar do:
            [: eachVar |
            defsSt detect: [:st | st guardingCondition = eachVar guardingCondition
                                  and: [self checkNotFieldAndInnerScopeOf: eachVar in: st]]
                  ifNone: [^false]]]].
   ^true

checkNotFieldAndInnerScopeOf: defVar in: defSt
   scopeVar := CProgramDB scopeAffectedBy: defVar.
   scopeSt := CProgramDB scopeAffectedBy: defSt.
   ^(defSt includesField: varName) not]) and: [scopeSt includesScope: scopeVar])
       and: [self checkStructVarForEachVarUse]
```

Figure 6.13: Preconditions of Move Variable Into Structure

In the case when there are multiple definitions of `varName`, there are again two cases:

- There is a single definition of `structName` and for each definition of `varName`, the checks in `checkStructVarForEachVarUse` hold, or
- There are as many definitions of `varName` as there are of `structName`, and the checks in `checkStructVarForEachVarUse` hold for each pair of definitions with the same condition.

Transformation:

Figure 6.14 shows how this refactoring is performed in the case when there is a single definition of `varName`.

```
performSingleDefVar
   defVar := CProgramDB definitionOfVariable: varName inFile: varFile at: varInterval first.
   defSt := CProgramDB definitionOfStruct: structName visibleIn: scopeOfSelection.
   self moveVar: defVar intoStruct: defSt


moveVar: defVar intoStruct: defSt
   | nodeVar nodeSt |
   scope :=  CProgramDB scopeAffectedBy: defSt.
   nodeVar := self searchForVar: defVar inASTOf: scope.
   nodeSt := self searchForStruct: defSt inASTOf: scope.
   (defSt guardingCondition = defVar guardingCondition)
      ifTrue: [self moveNode: nodeVar into: nodeSt]
      ifFalse:
        ["means guardingCondition of defSt is true"
        self insertCondIf: (defVar guardingCondition) into: nodeSt.
        self moveNode: nodeVar into: nodeSt.
        self insertEndifInto: nodeSt].
   self replaceUsesOf: defVar withAccessTo: defSt
```

Figure 6.14: Performing Move Variable Into Structure

In the code of Figure 6.14, message `searchForVar:inASTOf:` constructs a CRParseTreeSearcher that finds the node representing the declaration of `defVar` in the AST(s) of the scope. Similarly, the message `searchForStruct:inASTOf:` returns the node representing the declaration of `defSt`. When message `moveNode:into:` is called, the node for the declaration of `varName` is moved as the last field of the node for `defSt`. In the case when the guarding condition of `defSt` is a TrueCondition, `nodeVar` is surrounded by a Cpp conditional with one branch for the guarding condition of `defVar`.

The message `replaceUsesOf:withAccessTo:` will replace every use of `varName` by a reference to "`structVar.varName`", where `structVar` represents the way to refer to an instance of `defSt` in the scope (that should be unique). Note that `structVar` may be a variable or a parameter of type '`struct structName`' or '`struct *structName`', etc.

The case when there are multiple definitions of `varName` will just consist of calling

`moveVar:intoStruct:` several times, for each appropriate combination of `defVar` and `defSt`.

6.4.4 Other refactorings on C code

This section presents, for each category of refactorings in Table 6.1, a high-level discussion of the issues that Cpp may introduce and that should be addressed in the preconditions or transformations of the refactorings.

1. **Adding a program entity.** This category of refactorings is probably the easiest to implement even in the presence of Cpp directives, since the entities introduced are new to the program. The only consideration is, like in the case of 'Rename variable' (Section 6.4.2), making sure that the name of the new entity is not used in the scope, which may include more than one file and a particular configuration, and also includes macro definitions.

2. **Removing a program entity.** Section 6.4.1 describes the refactoring 'Delete unreferenced variable' in detail. The other two refactorings in this category should look very similar. In the case of 'Delete unreferenced parameter', a further step is necessary to transform all calls to the function in which the parameter is deleted, removing the corresponding actual argument.

3. **Changing a program entity.** Section 6.4.2 describes the first refactoring of this category, 'Rename variable', in detail. The other rename refactorings look very similar. The only difference in 'Rename field' is that it requires the parse tree to be typed-checked before applying the transformation. The reason is that every structure creates a separate name space, and the refactoring should only apply to the selected structure. In other words, fields with the same name of the one being renamed but in a different structure, should not be renamed.

The refactoring ‘Replace type’ is discussed in [2] under the name ‘change_type’. Basically, the preconditions ensure type safe assignments. The only additional remark is that assignments should be checked for type safety under all possible configurations.

The refactorings ‘Contract variable scope’ and ’Extend variable scope’ involve code movement (the movement of the variable definition). Therefore, the issues discussed in Section 6.2.5 about macros affecting code movement, and those in Section 6.3.2 about conditionals affecting code movement, should be addressed to implement these refactorings.

Refactorings that add or remove a level of indirection: ‘Change to pointer’ and ‘Change from pointer’ respectively, require some specific checks on the macros involved in the transformation, if any. On the one hand, these refactorings should check that all macro calls are changed in the same way, just like in ‘Rename variable’, to address the issues discussed in Sections 6.2.2 and 6.2.3. On the other hand, parentheses are necessary when an address operator (&) or indirection operator (*) is added, and if macros are involved, they may bring the issues discussed in Section 6.2.8 about the use of parentheses in macros.

The other refactorings in this category should not bring further issues that the ones already discussed.

4. **Complex refactorings.** The first refactoring in this category, ‘Group variables in new structure’, is similar to the second, ‘Move variable into structure’, which was presented in detail in Section 6.4.3. Nevertheless, the first is easier to implement because it adds a new structure definition and variable(s), instead of having to deal with existing, and possibly multiple, structure definitions.

   Many of the issues that may arise during ‘Extract function’ are discussed in the first three sections of this chapter. Moreover, ‘Inline function’ and ‘Consolidate

conditional expression' (which joins adjacent cases in a switch) involve code movement and therefore share similar issues.

The last two refactorings, 'For into while' and 'While into for' should not bring additional complications.

6.5 Refactorings on Cpp Directives

This section presents new refactorings that apply on `#include`, `#define` and conditional directives. In general, the refactorings in this section are simpler than the ones in the previous section, because they mostly deal with the language of Cpp, whereas the refactorings on C code need to deal with two languages: C and Cpp.

Table 6.2 shows a list of refactorings that we propose for preprocessor directives.

Table 6.2: Refactorings on preprocessor directives

Add file and #include
Remove #include
Rename macro
Rename macro parameter
Add macro parameter
Remove macro parameter
Extract macro
Inline macro
Remove condition
Complete Cpp conditional
Move common code outside Cpp conditional

The following subsections describe two refactorings for each Cpp directive. Since these refactorings are simpler, their description is not as detailed as in the previous section. Their correctness follows easily from observing that the output of Cpp is the same before and after applying the refactoring.

6.5.1 Add file and #include

The user selects a piece of code to be moved into a new file. A `#include` directive of the new file in inserted at the place of the selection.

Input values:

1. Text to be moved into a new file (`text`).
2. The interval (`interval`) in a file (`filename`) that contains the `text`.
3. The name of the new file (`newFilename`).

Preconditions:

The name `newFilename` should not be used already by another file in the same directory. Moreover, the selected piece of code should be a complete syntactical unit. This precondition assures that the end result is still a valid input for P-Cpp, which requires each file to be parseable on its own (see Section 4.2).

Transformation:

The refactoring cuts the selected `text` out of file `filename` and in its place adds a line: `#include ‘‘newFilename’’`. Then the refactoring creates a new file with name `newFilename`, in the same directory as `filename` and pastes the selected text in it. Finally, two changed objects will be created: one to re-process `filename` and re-create its symbol table, and the other to add `newFilename` to the IDG, process it and create its symbol table.

This refactoring preserves behavior because the resulting code after preprocessing will be the same than before the refactoring was applied. That is, Cpp will replace the new `#include` line with the code of file `newFilename`, leaving the source code exactly as it was before the refactoring. The precondition also ensures that there are no side effects of `newFilename` clashing with an existing file name.

6.5.2 Remove file and #include

A file is removed from the set of program files and each `#include` directive for that file is replaced by its text.

Input values:

1. The complete name of the file to be removed (`filename`).

Preconditions:

The only precondition is that a file with name `filename` is part of the program, i.e., exists in the IDG.

Transformation:

The first step is to search for the Program File with name `filename` in the IDG (let us call it PF). Its direct successors in the IDG are the files that have a `#include` directive to include it, so the next step is to visit the ASTs of all direct successors of PF and replace the node representing the `#include` line by the AST of PF. Then PF is removed from the IDG. Change objects are finally created for each modified file, to re-process the file entirely.

Applying this refactoring is like preprocessing the #include directive. Therefore, the code before and after this refactoring is equivalent at parsing time, and so there is no change of behavior.

6.5.3 Rename macro

In the same way that a variable or a function can be renamed, so a macro can be renamed. Moreover, there are also two versions of this refactoring: *Single Rename* and *Multiple Rename*, depending on whether the user chooses to rename a single definition of the macro or all definitions of the macro.

Input values:

1. The name of the macro to be renamed (`oldName`).
2. The position interval (`interval`) in a file (`filename`) that contains the user-selected reference to `oldName`.
3. The new name for the macro (`newName`).

Preconditions:

The preconditions of this refactoring are very similar to the preconditions of 'Rename variable'. The only difference is that, in the case of Single Rename, instead of calling the method isUniqueDefinitionOfVariable:inFile:inInterval:, it will be calling the method isUniqueDefinitionOfMacro:inFile:inInterval:

Transformation:

The transformation is also similar to 'Rename variable', although the Parse Tree Rewriter differs. The Parse Tree Rewriter will not be searching for Identifier nodes in this case, because uses of the macro are not represented as nodes but as node labels. Furthermore, uses of the macro (i.e., macro calls) do not store the name of the macro but instead obtain the name from the macro definition to which they bound.

Therefore, changing the macro definition is enough, except for one case: if the macro name is present in the body of another macro, it is necessary to change the text of the other macro. Consequently, the Parse Tree Rewriter searches for macro definition nodes and replaces their names or their body.

6.5.4 Rename macro parameter

This refactoring replaces the name of a formal parameter in a single macro definition. If other macro definitions with the same name appear under other configurations, they are not affected in the refactoring.

Input values:

1. The name of the parameter to be renamed (`oldName`).
2. The position interval (`interval`) in a file (`filename`) that contains the user-selected reference to `oldName`.
3. The new name for the parameter (`newName`).

Preconditions:

The preconditions are very simple in this case, because the scope is confined to the macro definition. The preconditions are that the input values are correct and that the `newName` is not used in the macro definition (by another parameter or in the replacement text).

Transformation:

There is a single node to replace in this case, the node for the macro definition to which the parameter belongs. The macro definition is transformed by replacing the parameter name in the parameter list and replacing all occurrences of the name `oldName` in the body of the macro. Note that this differs from function parameter renaming, where a different use of `oldName` would remain unmodified.

The argument for behavior preservation is the same as in the previous refactoring, as changing the name of a parameter does not change the macro's behavior or the C preprocessor output.

6.5.5 Remove condition

When a configuration is discontinued, this refactoring allows removing all Cpp conditional branches for certain condition that identifies the discontinued configuration.

Input values:

1. The text for the condition to be removed from the Cpp conditionals of the program (`cond`).

Preconditions:

None, but the user is responsible for making sure that the configuration will not be used in the future.

Transformation:

All ASTs of the program will be searched for nodes representing a conditional directive with condition `cond`. The branches created by those conditional directives should be removed from the Cpp conditional construct in which they appear.

Figure 6.15 uses Maude's rewrite rules to describe how a Cpp conditional is replaced when the branch with condition '`X`' is eliminated. The module in Fig. 6.15 uses the sort TOKEN defined in Chapter 3.

```
mod RULES-REM-COND is
  pr TOKEN .
  op <_> : TokenSequence -> State [ctor] .
  vars X Y TS1 TS2 TS3 : TokenSequence .

  rl [ifChangeElse] : < '#if X TS1 '#else TS2 '#endif > => < '#if !X TS2 '#endif > .
  rl [ifChangeElif] : < '#if X TS1 '#elif Y TS2 '#endif> => < '#if Y TS2 '#endif > .
  rl [elseRem]      : < '#if !X TS1 '#else TS2 '#endif > => < '#if !X TS1 '#endif > .
  rl [elifRem]      : < '#if Y TS1 '#elif X TS2 '#elif TS3 '#endif >
                    => < '#if Y TS1 '#elif TS3 '#endif > .
  rl [elifRemElse]  : < '#if Y TS1 '#elif X TS2 '#else TS3 '#endif >
                    => < '#if Y TS1 '#else TS3 '#endif > .
  rl [ifRem]        : < '#if X TS1 '#endif > => < nil > .
endm
```

Figure 6.15: Rules for Remove Condition

6.5.6 Complete Cpp conditional

With this refactoring, a Cpp conditional is transformed so that its branches are complete syntactical units.

Input values:

1. Location of the Cpp conditional to be completed.

Preconditions:

None.

Transformation:

This refactoring "externalizes" the internal transformation that P-Cpp does on Cpp conditionals when running the Conditional Completion Algorithm (see Section 4.5.5). Since Cpp conditionals are already completed in the abstract syntax tree, this refactoring only requires changing the labels assigned by the Conditional Completion Algorithm to the nodes in the tree. All nodes inside the Cpp conditional should be labelled as '**not-moved**', so the pretty-printer will print them at the current place.

Chapter 7

Quantitative Evaluation of CRefactory

CRefactory is implemented in VisualWorks Smalltalk^{TM}. The refactoring engine mimics the design of the Smalltalk Refactoring Browser [7]. This chapter provides a quantitative analysis of the program representations built by CRefactory when loading some open-source packages. These metrics show that, in practice, conditional completion does not appear to produce an exponential growth of the representation of programs.

All these tests were run on a Linux platform with a 2.4GHz processor.

7.1 Results on rm

The source code for rm is contained in a single source file: "rm.c". When this source file was loaded in CRefactory, it included 94 header files, although only 20 of them belong to the same package and are therefore modifiable (the others are GCC library headers). Note that all these headers are included when considering all possible configurations, only excluding false conditions (which account to 12 conditions, mostly to disallow GCC extensions that CRefactory still does not support, such as variable

and function attributes [66]).

Table 7.1 shows the results on all program files and on the 20 files in the rm package separately. It took 12 seconds to load the whole program in CRefactory, which includes 9 seconds of pseudo-preprocessing time.

Table 7.1: Metrics on rm

	All files	**'rm' package**
Number of files	94	20
Size	556 Kb	104 Kb
Number of Cpp conditionals	1182	262
Number of Cpp conditionals introduced by macro expansion	41	30
Number of incomplete Cpp conditionals	41	30
Perc. of code growth after conditional completion	3%	18%
Maximum level of nesting of Cpp conditionals	23	3
Percentage of conditional definitions	48%	24%
Number of macros defined	2131	516
Number of macros with 1 definition	2028	497
Number of macros with 2 definitions	94	19
Number of macros with 3 definitions	7	0
Number of macros with 6 and with 7 definitions	1	0
Number of symbols defined as macros and functions	66	2
Number of symbols defined as macros and variables	6	0

There is one header file in the package: `system.h`, that alone contains 166 out of the 262 Cpp conditionals. It contains many macro definitions (31% of all macro definitions in the package). We can infer that this file is highly configurable and so, difficult to maintain.

From the total of 262 Cpp conditionals in the rm package, 30 were introduced by P-Cpp due to macro expansion (i.e., because of calls to macros with more than one definition). All of these Cpp conditionals and only those were incomplete. Therefore, all Cpp conditionals present in the source code of the rm package are complete, which speaks very well of the readability of the source code. Moreover, 27 out of the 30 Cpp conditionals introduced by P-Cpp appear in a single file: `rm.c`. Since all of these

conditionals had to be completed, the tokenized representation of `rm.c` grew 57% after conditional completion. Regarding standard library files, all Cpp conditionals in their source code were also complete.

The depth of nesting of Cpp conditionals in the rm package is low, so the source code is not complex in that sense. However, the percentage of symbol definitions that depend on configuration variables (i.e., conditional definitions) is rather high. Considering all possible configurations simultaneously is therefore very important to be able to modify this code.

The macro that has 6 definitions in the program is `LONG_BIT` and the one with 7 definitions is `WORD_BIT`, both defined in file '`/usr/include/bits/xopen_lim.h`'. Although it would be complex to expand and complete a conditional with 6 or 7 branches inside a possible complicated expression, these macros are not used or referenced in the program.

One of the symbols defined as a function under one configuration and as a macro under another configuration in a different header file is `getopt`. Refactoring this symbol or the code that uses it would be difficult without a tool that can spot these double definitions and check for possible problems.

7.2 Results on Flex

The source code for Flex is contained in 13 source files. However, two of them are automatically generated from grammar specifications. Loading Flex involved loading 54 files, but Table 7.2 shows the results obtained on the 11 source files that are not auto-generated plus the 4 headers in the Flex package (i.e., all the modifiable files).

Table 7.2 shows that the Flex package is not too complex in terms of Cpp conditionals. As with the previous case, all incomplete Cpp conditionals were introduced by P-Cpp due to macro expansion.

Table 7.2: Metrics on Flex

Number of files	15
Size	245 Kb
Number of Cpp conditionals	36
Number of Cpp conditionals introduced by macro expansion	9
Number of incomplete Cpp conditionals	9
Perc. of code growth after conditional completion	1%
Maximum level of nesting of Cpp conditionals	1
Percentage of conditional definitions	2%
Number of macros defined	134
Number of macros with 1 definition	134
Number of symbols defined as macros and functions	0
Number of symbols defined as macros and variables	0

The files that have the most conditional directives are `flexdef.h`, `main.c` and `misc.c`. File `flexdef.h` has 15 Cpp conditionals, all present in the source code and all complete. In the case of `misc.c`, 5 out of 7 Cpp conditionals come from macro expansion. The tokenized representation of `misc.c` grew 4% after conditional completion. The file that grew the most was `yylex.c`, with a growth of 7%.

There is no nesting of the Cpp conditionals in the Flex package, although the maximum level of nesting is 23 when counting library files.

There is one unreferenced variable in the package: `copyright`, defined under condition $\neg defined(lint)$.

7.3 Results on linux/init/main.c

In this test case, the file '`init/main.c`' of the Linux Kernel distribution version 2.6.7, was the only source file loaded, although it included a total of 227 header files. The configuration that loaded it into CRefactory had a total of 22 false conditions, again most of them to disallow GCC extensions. The configuration also had 6 pairs of incompatible conditions. It took 3 minutes to preprocess and another 3 minutes to

parse and populate the Program Repository with complete def-uses chains.

Table 7.3 shows the quantitative analysis on '`init/main.c`' and all included files.

Table 7.3: Metrics on init/main.c

Number of files	228
Size	1.17 Mb
Number of Cpp conditionals	1169
Number of Cpp conditionals introduced by macro expansion	364
Number of incomplete Cpp conditionals	351
Perc. of code growth after conditional completion	17%
Maximum level of nesting of Cpp conditionals	8
Percentage of conditional definitions	38%
Number of macros defined	6653
Number of macros with 1 definition	6486
Number of macros with 2 definition	149
Number of macros with 3 definition	14
Number of macros with 4 definition	2
Number of macros with 7 definition	1
Number of macros with 20 definition	1
Number of symbols defined as macros and functions	37
Number of symbols defined as macros and variables	6

It is remarkable that the 94 files of the `rm` package (which includes 74 standard library header files) are as much or even more complex in terms of Cpp conditionals as the 227 header files loaded with '`init/main.c`'. For example, the `rm` package has 1182 Cpp conditionals, while this Linux program has 1169. Also, the percentage of conditional definitions in `rm` was 48%, while in this case is 38%. Moreover, the maximum level of nesting of Cpp conditionals is 23 in `rm` and 8 here.

On the contrary, '`init/main.c`' and its headers have about 3 times more macro definitions than the 'rm' package. That creates a larger number of Cpp conditionals introduced by macro expansion and consequently, a larger number of incomplete Cpp conditionals.

The macro that has 7 definitions is `SHIFT_HZ`. There is at least one statement in

file '`include/linux/time.h`' that calls this macro 5 times. When completing this 5 conditionals in the single statement, the conditionals have to be combined. The number of conditional branches would be huge (7^5) if our conditional completion algorithm was not checking that the conditionals really come from the same macro. With that optimization the number of branches becomes 7. Nevertheless, the file 'time.h' calls the macro `SHIFT_HZ` several times (11 to be exact), and that causes the file to grow 261% after conditional completion. The rest of the files do not grow much. The average file growth is 5%.

The macro with 20 definitions is `MODULE_PROC_FAMILY`, and all definitions are in file '`include/asm/module.h`', in different branches of a single Cpp conditional. This macro could have caused a large growth percentage, although there are no uses of this macro.

During this case study, we realized the importance of reusing previously generated representations. The file '`include/linux/config.h`' gets re-included 69 times. Another file is re-included 34 times, and so on. Moreover, while testing an earlier version of CRefactory, we realized that calculating the binding of uses to definitions had to be done after the symbol tables are constructed. In that earlier version, def-use chains were updated at the same time that the symbol tables were constructed, and that required a file to be revisited by a CRASTUsesModifier every time the file was included. Processing the program took about 1 hour. The new version of CRefactory goes once through each file in the Include Dependencies Graph to populate the symbol tables with local symbol definitions and uses, and a second time to update the uses of global symbols. The current version runs 10 times faster.

7.4 Results on Directory linux/init/

For this test case, we loaded all the source files in the '`init/`' directory of the Linux distribution. There are 8 source files in that directory, and when we loaded them in CRefactory, the number of program files reached 321. There were a total of 29 false conditions needed to load the program, together with the same incompatible condition pairs than in the case of '`init/main`'. It took 15 minutes to load the program, which includes almost 11 minutes of pseudo-preprocessing time. Table 7.4 shows the quantitative results of this test case.

Table 7.4: Metrics on linux-2.6.7/init/

Number of files	321
Size	1.68 Mb
Number of Cpp conditionals	1988
Number of Cpp conditionals introduced by macro expansion	928
Number of incomplete Cpp conditionals	918
Perc. of code growth after conditional completion	13%
Maximum level of nesting of Cpp conditionals	8
Percentage of conditional definitions	35%
Number of macros defined	8805
Number of macros with 1 definition	8613
Number of macros with 2 definition	173
Number of macros with 3 definition	15
Number of macros with 4 definition	2
Number of macros with 7 definition	1
Number of macros with 20 definition	1
Number of symbols defined as macros and functions	60
Number of symbols defined as macros and variables	8

It is interesting to compare growth percentages from the previous test case where only the file '`main.c`' of directory '`init/`' was loaded. Now that the 8 source files in that directory were loaded, the number of files in the IDG (Include Dependencies Graph) grew 41% and the size of the program grew 43%. With that growth in program

size, the number of macro definitions grew 32%, about the same growth as the number of files. However, the number of Cpp conditionals grew 70% and the number of incomplete Cpp conditionals, all coming from the expansion of macro calls, grew 161%. While the increase in the number of incomplete Cpp conditionals, together with the increase in program size, causes pseudo-preprocessing to take considerably longer, it does not cause a raise in the percentage of code growth after conditional completion. In the previous test case, this percentage was 17%, while now it is 13%. The reason is that the file that grows the most, '`include/linux/time.h`', was already included in the previous test case, and the additional files in this case grow an average of 3%, which spreads the higher percentages among a larger number of files.

Chapter 8

Conclusions

8.1 Summary of Contributions

Integrating preprocessor directives during refactoring is hard, for both the analysis of the code required before refactoring and the transformation functions themselves. This is a major factor that hinders the development of refactoring tools for C and C++ code. The main contribution of this thesis is that it demonstrates that complete integration between Cpp and C is possible.

Although CRefactory focuses on C and Cpp, our work applies to other languages that use Cpp, like C++ and Fortran, and probably to other preprocessors with similar directives, whose syntax is independent of the syntax of the underlying language (for example, C#, which has conditional directives). Moreover, analysis tools should be able to apply the same ideas for parsing and representation of Cpp directives. In fact, the formal semantics that we have given to Cpp is programming language independent. For this reason, the Cpp refactoring rules are programming language generic, i.e., they will be applicable to any programming language using Cpp and will also be correct for that language.

The main contributions of this thesis are:

- A formal executable specification of the C preprocessor.

- A new method for preprocessing that does not remove preprocessor directives but prepares the source code for parsing. This method includes the Conditional Completion Algorithm.
- Design of program representations that integrate C and Cpp.
- The identification of problems posed by Cpp in refactoring.
- Specification of the new preconditions and transformation rules for some refactorings, which solve the problems identified.
- A catalog of refactorings for Cpp directives.
- A refactoring tool for C: CRefactory.
- Measurements on the program representations when loading a few case studies.

8.2 Lessons Learned by Implementing CRefactory

The implementation of a refactoring tool has been essential to discover flaws or limitations of our approach and correct them. It also gave us the opportunity to test our ideas on real programs and to measure the performance of our algorithms and program representations. Some of the lessons we have learned on the road towards implementing CRefactory have been:

1. *Handling multiple configurations at a time is the hardest problem.*

   At the early stages of this research, we believed that handling macros was the main difficulty to be able to support refactoring on C + Cpp. While it is true that macros invalidate many refactorings and complicate the transformations, being able to parse conditional directives and to build a program representation of all possible program configurations has been the most challenging aspect of this work.

The first obstacle was to figure out how to parse multiple configurations at a time without restricting the placement of conditional directives. In a personal communication with Ira Baxter, one of the directors of the DMS project, he warned us of this problem and advised us that parsing in multiple passes was unwise, the reasons being the exponential time and complexity of that approach. That is the reason for designing and implementing the Conditional Completion Algorithm, whose novelty makes it one of the main contributions of this work.

Once parsing of multiple configurations was solved, the second obstacle was being able to expand a macro with multiple definitions. The solution was to expand the macro as a Cpp conditional construct. This added more complexity to our Conditional Completion Algorithm, since more than one Cpp conditional could break the same statement, and those conditionals had to be combined.

Supporting multiple configurations also required symbol tables to be enhanced, and with that, a new definition of visibility and binding of symbol uses to definitions was necessary.

Last but not least, pretty-printing the completed conditionals back to their original, un-completed version, led us to design the complex labelling of tokens that CRefactory currently has.

2. *It is common for conditional directives to appear inside structures, enumeratives and initializers.*

   An early version of CRefactory did not allow conditional directives to appear in between structure fields, enumerator values or array initializers. During testing of the Linux Kernel code, there was an exponential growth of code size and processing time to generate a different array initializer, structure declaration or enumerative for each possible combination of conditionals inside them. For example, the initializer of array `kern_table` in file '`/include/linux/sysctl.h`'

has 14 Cpp conditionals inside, which would generate 2^{14} different initializers. Allowing conditional directives to appear inside these constructs has solved the problem.

3. *Exact pretty-printing is difficult but necessary.*

   Developers demand exact or nearly exact pretty printing in any usable tool. That is why it was important and necessary to reverse the completion of Cpp conditionals and macro expansion, to assign exact position to tokens and to preserve comments. Moreover, the pretty-printing algorithm is designed to preserve the original format of all code pieces that have not been affected by refactorings.

4. *Reuse of individual file representations makes the tool more efficient.*

   It is common for a header file to be included more than once in a program. Although multiple inclusion is usually prevented by using "include guards" (a combination of conditional directive and macro definition), they have to be ignored when preprocessing multiple source files and multiple configurations simultaneously. For this reason, it became important to represent each file separately, detaching its representation from the context in which it is included, and reusing its representation on subsequent includes. This requires each file to be a complete syntactical unit, parseable on its own. We believe that the restriction is worthwhile; furthermore, it is widely accepted and advisable to follow it.

   Moreover, an earlier version of CRefactory updated the def-use chains of each symbol while constructing the symbol tables. This required the ASTs of previously included files to be re-visited each time, to update the symbol tables already constructed. Changing this approach to instead update the symbol tables after they were constructed decreased the time to process the file '`/init/main.c`' of the Linux kernel from 1 hour to 6 minutes.

8.3 Limitations

In the following list, the first two items are limitations of the input that CRefactory accepts. The rest are limitations of the current implementation of CRefactory.

1. Except for conditional directives, all other Cpp directives must appear in between statements, declarations, structure fields, enumerator or initializer values. Note that directives may appear inside a compound statement, as long as they do not break any statement inside the compound statement. These are the only places we have found in open source code packages where programmers place all but conditional directives, so we believe that this assumption is not too restrictive in the kind of input that CRefactory accepts.

2. Each file in the program must be a complete syntactical unit, parseable on its own. As described in the previous section, while it is standard practice to have header be complete units, this is also a worthwhile restriction that makes CRefactory much more efficient.

3. While testing CRefactory on open-source packages, it became increasingly important to support some GCC extensions [66]. CRefactory currently supports statement expressions, assembler instructions, inline specifier, alternate keywords like `__inline__`, `__volatile__` and `__asm__`, and double-word integers. However, there are many extensions that CRefactory still does not support, like nested functions, empty structures, variadic macros, and variable and function attributes.

4. The Conditional Completion Algorithm currently supports 7 cases of conditionals and combination of conditionals. Although these cases cover all those that we have found in practice, the list is not exhaustive. See Chapter 4 for a list of supported and unsupported cases.

8.4 Future Work

The first step in future work should be to release CRefactory to the public, realizing a long-awaited contribution to the C community. For this purpose, the existent Emacs plugin for CRefactory ([71]) should be upgraded to match the new features of the current version.

In future releases, CRefactory could be extended to outgrow its limitations, i.e., adding all GCC extensions and supporting other cases of conditional completion.

Moreover, the set of supported refactorings could be extended. CRefactory currently supports renaming C entities and macros, deleting unreferenced variables, moving variables to a structure, creating a new structure from a set of variables, converting a variable to a pointer and viceversa, and moving a variable up in the scope. There are many other useful refactorings listed in our previous work that would be useful to implement [16]. The first one should be function extraction, which is already halfway implemented.

We believe that our approach is directly applicable to C++. It would be interesting to demonstrate this idea. Firstly, the C++ grammar should be extended with Cpp directives, finding the places where directives commonly appear in C++ constructs. Secondly, the first pass of the Conditional Completion Algorithm should be extended to recognize the beginning and end of the new constructs and consequently be able to identify incomplete conditionals. Thirdly, the interaction of macros with C++ refactorings should be studied to discover potential new problems. Nevertheless, we do not foresee large obstacles since macros are not heavily used in C++. The language itself includes features that limit the need for macros (e.g., templates).

Another area of future work would be to formally prove that pseudo-preprocessing, and specially, the Conditional Completion Algorithm, preserves program behavior. This could be accomplished by completing the Maude specification of P-Cpp and using Maude's theorem prover to show that Cpp's semantics are equally observed by

(P-Cpp + Cpp)'s semantics. That is, if P-Cpp is applied on a certain input, and the output of P-Cpp is flattened in a way that Cpp can subsequently process it, the output would be the same as using Cpp alone on the same input and with the same values for configuration variables. This algebraic simulation technique is described by Martí-Oliet et.al [72].

Another interesting theoretical area would be to study whether refactorings can be specified with Maude's rewrite rules. For this purpose, it would be necessary to devise a way to carry context information to a rewrite rule, i.e., passing the data stored in the program repository from rule to rule. If Maude is found suitable for this endeavor, it would make a large impact on the refactoring community, since proving the behavior preservation of each refactoring would be relatively straightforward. Furthermore, proving the correctness of some of the refactorings on Cpp directives (those that do not need context information, like the one in Figure 6.15) should be easier to achieve.

Appendix A

Maude Specification of Cpp

The following is the Maude specification of the syntax of Cpp.

```
--- ------------- ---
--- SYNTAX OF CPP ---
--- ------------- ---

fmod IDENTIFIER is
  pr QID .
  sort Identifier IdentifierList IdentifierListP .
  subsorts Qid < Identifier < IdentifierList .
  op `(` : -> IdentifierListP .
  op `(_` : IdentifierList -> IdentifierListP .
  op _,_ : IdentifierList IdentifierList -> IdentifierList [assoc] .
  op size : IdentifierListP -> Nat .
  op _in_ : Identifier IdentifierListP -> Bool .
  op pos : Identifier IdentifierListP -> Nat .
  op elemAt : Nat IdentifierListP -> Identifier .
  op cons : Identifier IdentifierListP -> IdentifierListP .

  vars I I' : Identifier . var IL : IdentifierList . var N : Nat .
  var ILP : IdentifierListP .
  eq size(()) = 0 .
  eq size((I, IL)) = 1 + size((IL)) .
  eq size((I)) = 1 .
  eq I in () = false .
  eq I in (I', IL) = (I == I') or (I in (IL)) .
  eq I in (I') = (I == I') .
  eq pos(I, ()) = 0 .
  ceq pos(I, (I', IL)) = 1 if (I == I') .
  ceq pos(I, (I', IL)) = 1 + pos(I, (IL)) if (I =/= I') and (I in (IL)) .
  ceq pos(I,(I')) = 1 if (I == I') .
  eq pos(I, (IL)) = 0 [owise] .
  eq elemAt(1, (I, IL)) = I .
  eq elemAt(s(N), (I, IL)) = elemAt(N, (IL)) .
  eq elemAt(1, (I)) = I .
  eq cons(I,()) = (I) .
  eq cons(I,(IL)) = (I,IL) .
endfm

fmod TOKEN is
  pr IDENTIFIER .
  sorts Token TokenSequence .
  subsort Identifier < Token < TokenSequence .
  op nil : -> TokenSequence .
```

```
  op __ : TokenSequence TokenSequence -> TokenSequence [assoc id: nil] .
  op _inTS_ : Token TokenSequence -> Bool .
  vars T T' : Token . var TS : TokenSequence .
  eq T inTS nil = false .
  eq T inTS (T' TS) = (T == T') or (T inTS TS) .
endfm

--- --- --- --- --- --- --- --- --- ---
--- Definition of expresions: CondExp ---

fmod COND-EXP-SYNTAX is
  pr TOKEN . pr INT .
  sort CondExp .
  subsort Identifier < CondExp .
  op e : Int -> CondExp .   --- had to do this instead so I can modify attributes
  op tokenize : CondExp -> TokenSequence .
endfm

fmod MACRO-CALL-SYNTAX is ex COND-EXP-SYNTAX .
  sorts ArgList ArgListP MacroCall .
  subsort CondExp < ArgList .
  subsort IdentifierList < ArgList .
  subsort IdentifierListP < ArgListP .
  subsort Identifier < MacroCall .
  subsort MacroCall < CondExp .
  op ((_)) : ArgList -> ArgListP .
  op _,_ : ArgList ArgList -> ArgList [ditto] .
  op __ : Identifier ArgListP -> MacroCall [prec 30] .
  op name : MacroCall -> Identifier .
  var I : Identifier . var ALP : ArgListP .
  eq name(I)  = I .
  eq name(I ALP) = I .
endfm

fmod DEF-COND-SYNTAX is ex COND-EXP-SYNTAX .
  op defined_ : Identifier -> CondExp .
endfm

fmod ARITH-EXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _+_ : CondExp CondExp -> CondExp [prec 40 gather(e E)] .
  op _-_ : CondExp CondExp -> CondExp [prec 40 gather(e E)] .
  op _*_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
  op _/_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
  op _%_ : CondExp CondExp -> CondExp [prec 35 gather(e E)] .
endfm

fmod BIT-EXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _<<_ : CondExp CondExp -> CondExp [prec 42] .
  op _>>_ : CondExp CondExp -> CondExp [prec 42] .
  op _&_ : CondExp CondExp -> CondExp [prec 46] .
  op _^_ : CondExp CondExp -> CondExp [prec 46] .
  op _|_ : CondExp CondExp -> CondExp [prec 46] .
endfm

fmod REXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _<_  : CondExp CondExp -> CondExp [prec 44] .
  op _<=_ : CondExp CondExp -> CondExp [prec 44] .
  op _>_  : CondExp CondExp -> CondExp [prec 44] .
  op _>=_ : CondExp CondExp -> CondExp [prec 44] .
  op _==_ : CondExp CondExp -> CondExp [prec 45] .
  op _!=_ : CondExp CondExp -> CondExp [prec 45] .
endfm

fmod BEXP-SYNTAX is ex COND-EXP-SYNTAX .
  op !_ : CondExp -> CondExp [prec 30] .
  op _&&_ : CondExp CondExp -> CondExp [prec 49] .
  op _||_ : CondExp CondExp -> CondExp [prec 51] .
endfm
```

```
fmod CEXP-SYNTAX is ex COND-EXP-SYNTAX .
  op _?_:_ : CondExp CondExp CondExp -> CondExp [prec 55] .
endfm

fmod ALL-COND-EXP-SYNTAX is
  pr COND-EXP-SYNTAX .
  pr MACRO-CALL-SYNTAX .
  pr DEF-COND-SYNTAX .
  pr ARITH-EXP-SYNTAX .
  pr BIT-EXP-SYNTAX .
  pr REXP-SYNTAX .
  pr BEXP-SYNTAX .
  pr CEXP-SYNTAX .
endfm

--- CppDirectives

fmod CPP-DIR-SYNTAX is
  sort CppDirective .
endfm

fmod DEFINE-SYNTAX is ex CPP-DIR-SYNTAX .
  pr TOKEN .
  sorts MacroDefDir MacroUndefDir .
  subsort MacroDefDir MacroUndefDir < CppDirective .
  op #define__cr : Identifier TokenSequence -> MacroDefDir .
  op #define___cr : Identifier IdentifierListP TokenSequence -> MacroDefDir .
  op #undef_cr : Identifier -> MacroUndefDir .
endfm

fmod INCLUDE-SYNTAX is ex CPP-DIR-SYNTAX .
  pr IDENTIFIER . pr MACRO-CALL-SYNTAX .
  sorts IncludeDir FileName .
  subsort IncludeDir < CppDirective .
  op <_> : Identifier -> FileName .
  op #include_cr : FileName -> IncludeDir .
  op #include_cr : String -> IncludeDir .
  op #include_cr : MacroCall -> IncludeDir .
endfm

fmod COND-DIR-SYNTAX is ex CPP-DIR-SYNTAX .
  pr IDENTIFIER .
  pr ALL-COND-EXP-SYNTAX .
  sort CondDir .
  subsort CondDir < CppDirective .
  op #if_cr : CondExp -> CondDir .
  op #ifdef_cr : Identifier -> CondDir .
  op #ifndef_cr : Identifier -> CondDir .
  op #elif_cr : CondExp -> CondDir .
  op #else`cr : -> CondDir .
  op #endif`cr : -> CondDir .
endfm

fmod LINE-SEQ-SYNTAX is
  pr CPP-DIR-SYNTAX .
  pr TOKEN .
  sorts Line LineSeq .
  subsorts CppDirective < Line < LineSeq .
  op nilLS : -> LineSeq .
  op __ : LineSeq LineSeq -> LineSeq [assoc id: nilLS] .
  op _cr : TokenSequence -> Line .
  op _\`cr_ : TokenSequence Line -> Line .
  vars TS1 TS2 : TokenSequence .
  eq TS1 \ cr TS2 cr = (TS1 TS2) cr .
endfm

fmod CPP-SYNTAX is
  pr TOKEN .
```

```
  pr ALL-COND-EXP-SYNTAX .
  pr DEFINE-SYNTAX .
  pr INCLUDE-SYNTAX .
  pr COND-DIR-SYNTAX .
  pr LINE-SEQ-SYNTAX .
endfm
```

The Maude module QID defines quoted identifiers. By defining the sort Identifier as a super-sort of the sort Qid, an identifier like `var` is denoted as `'var`. An example of a TokenSequence would be `'return 'a '+ 'b ';`.

The following is the Maude specification of the semantics of Cpp.

```
in cpp-syntax.maude
--- ---------------- ---
--- SEMANTICS OF CPP ---
--- ---------------- ---

fmod STRINGS is  pr STRING .
  sort StringSet .
  subsort String < StringSet .
  op empty : -> StringSet .
  op __ : StringSet StringSet -> StringSet [assoc comm id: empty] .
endfm

fmod TOKEN-TO-ARG is pr TOKEN .
  sort TokenSeqList .
  subsort TokenSequence < TokenSeqList .
  op nilTSL : -> TokenSeqList .
  op _;_ : TokenSeqList TokenSeqList -> TokenSeqList [assoc id: nilTSL] .
  op size : TokenSeqList -> Nat .
  op elemAtTS : Nat TokenSeqList -> TokenSequence .
  op toTokenSeqList : TokenSequence -> TokenSeqList .
  var T : Token . vars TS1 TS2 TSA : TokenSequence . var TSL : TokenSeqList . var N : Nat .
  eq size(nilTSL) = 0 .
  eq size(TS1 ; TSL) = 1 + size(TSL) .
  eq elemAtTS(1, (TS1 ; TSL)) = TS1 .
  eq elemAtTS(s(N), (TS1 ; TSL)) = elemAtTS(N, TSL) .
  ceq toTokenSeqList(TS1) = TS1 if not ('`, inTS TS1) .
  ceq toTokenSeqList(TS1 '`, TS2) = TS1 ; toTokenSeqList(TS2)
    if not ('`, inTS TS1) and ('`, inTS TS2) .
  ceq toTokenSeqList(TS1 '`, TS2) = TS1 ; TS2 if not ('`, inTS TS1) and not ('`, inTS TS2) .
endfm

fmod MACRO-DEF is
  pr TOKEN . pr STRING . pr TOKEN-TO-ARG .
  pr MACRO-CALL-SYNTAX .
  sort MacroDef .
  op name_replText_ : Identifier TokenSequence -> MacroDef .
  op name_params_replText_ : Identifier IdentifierListP TokenSequence -> MacroDef .
  op name : MacroDef -> Identifier .
  op hasArgs : MacroDef -> Bool .
  op expand : MacroDef -> TokenSequence .
  op expandWithArgs : MacroDef ArgListP -> TokenSequence .
  op ex-rec : IdentifierListP TokenSequence ArgListP -> TokenSequence .
  op dquote : -> Qid .

  var N : Identifier . var TS : TokenSequence . vars T T2 : Token .
  var PL : IdentifierListP . var Args : ArgListP .
  eq name(name N replText TS) = N .
  eq name(name N params PL replText TS) = N .
  eq hasArgs(name N replText TS) = false .
  eq hasArgs(name N params PL replText TS) = true .
```

```
  eq expand(name N replText TS) = ex-rec(‘(‘), TS, ‘(‘)) .
  ceq expandWithArgs(name N params PL replText TS, Args) = nil
    if ( size(PL)  =/= size(Args) ) .
  eq expandWithArgs(name N params PL replText TS, Args) = ex-rec(PL, TS, Args) [owise] .
  eq ex-rec(PL, nil, Args) = nil .
  eq ex-rec(PL, '# T TS, Args) = dquote elemAt(pos(T, PL), Args) dquote
                        ex-rec(PL, TS, Args) .
  ceq ex-rec(PL, T '## T2 TS, Args) = qid(string(T) + string(T2)) ex-rec(PL, TS, Args)
    if not(T in PL) and not(T2 in PL) .
  ceq ex-rec(PL, T '## T2 TS, Args) = qid(string(elemAt(pos(T, PL), Args)) + string(T2))
        ex-rec(PL, TS, Args) if (T in PL) and not(T2 in PL) .
  ceq ex-rec(PL, T '## T2 TS, Args) = qid(string(T) + string(elemAt(pos(T2, PL), Args)))
        ex-rec(PL, TS, Args) if not(T in PL) and (T2 in PL) .
  eq ex-rec(PL, T '## T2 TS, Args) = qid(string(elemAt(pos(T, PL), Args)) +
        string(elemAt(pos(T2, PL), Args))) ex-rec(PL, TS, Args) [owise] .
  ceq ex-rec(PL, T TS, Args) = T ex-rec(PL, TS, Args) if not(T in PL) .
  eq ex-rec(PL, T TS, Args) = tokenize(elemAt(pos(T, PL), Args)) ex-rec(PL, TS, Args) [owise] .

  op expandWithTSArgs : MacroDef TokenSeqList -> TokenSequence .
  op ex-recTS : IdentifierListP TokenSequence TokenSeqList -> TokenSequence .

  var TSL : TokenSeqList .
  ceq expandWithTSArgs(name N params PL replText TS, TSL) = nil
    if ( size(PL)  =/= size(TSL) ) .
  eq expandWithTSArgs(name N params PL replText TS, TSL) = ex-recTS(PL, TS, TSL) [owise] .
  eq ex-recTS(PL, nil, TSL) = nil .
  eq ex-recTS(PL, '# T TS, TSL) = dquote elemAtTS(pos(T, PL), TSL) dquote
                        ex-recTS(PL, TS, TSL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(T) + string(T2)) ex-recTS(PL, TS, TSL)
    if not(T in PL) and not(T2 in PL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(elemAtTS(pos(T, PL), TSL)) + string(T2))
        ex-recTS(PL, TS, TSL) if (T in PL) and not(T2 in PL) .
  ceq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(T) + string(elemAtTS(pos(T2, PL), TSL)))
        ex-recTS(PL, TS, TSL) if not(T in PL) and (T2 in PL) .
  eq ex-recTS(PL, T '## T2 TS, TSL) = qid(string(elemAtTS(pos(T, PL), TSL)) +
        string(elemAtTS(pos(T2, PL), TSL))) ex-recTS(PL, TS, TSL) [owise] .
  ceq ex-recTS(PL, T TS, TSL) = T ex-recTS(PL, TS, TSL) if not(T in PL) .
  ceq ex-recTS(PL, T TS, TSL) = elemAtTS(pos(T, PL), TSL) ex-recTS(PL, TS, TSL) if (T in PL) .

endfm

fmod MACRO-TABLE is
  pr MACRO-DEF .
  sort MacroTable .
  op empty : -> MacroTable .
  op [_:_] : Identifier MacroDef -> MacroTable .
  op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
  op _[_] : MacroTable Identifier -> MacroDef .
  op _[_<-_] : MacroTable Identifier MacroDef -> MacroTable .
  op isMacro : Identifier MacroTable -> Bool .
  op isMacroWithArgs : Identifier MacroTable -> Bool .
  op isMacroWithoutArgs : Identifier MacroTable -> Bool .
  op remove : MacroDef MacroTable -> MacroTable .

  vars N N' : Identifier .  vars M M' : MacroDef . var MT : MacroTable .
  eq ([N : M] MT)[N] = M .
  eq ([N : M'] MT)[N <- M] = [N : M] MT .
  eq MT[N <- M] = MT [N : M] [owise] .
  eq isMacro(N, empty) = false .
  eq isMacro(N, ([N' : M] MT)) = (N == N') or isMacro(N, MT) .
  eq isMacroWithArgs(N, MT) = isMacro(N, MT) and hasArgs(MT[N]) .
  eq isMacroWithoutArgs(N, MT) = isMacro(N, MT) and not hasArgs(MT[N]) .
  eq remove(M, empty) = empty .
  eq remove(M, ([N : M] MT)) = remove(M, MT) .
  ceq remove(M, ([N : M'] MT)) = [N : M'] remove(M, MT) if M =/= M' .
endfm
```

```
fmod COND-EXP-SEMANTICS is
  pr COND-EXP-SYNTAX . pr MACRO-TABLE .
  op evalB : CondExp MacroTable -> Bool .
  op evalA : CondExp MacroTable -> Int .
  op toCondExp : TokenSequence -> CondExp .
  var X : Int . var MT : MacroTable .
  var N : Identifier . var MD : MacroDef .
  ceq evalB(e(X), MT) = true if X =/= 0 .
  eq evalB(e(0), MT) = false .
  ceq evalB(N, MT) = false if not isMacro(N, MT) .
  eq evalA(e(X), MT) = X .
  eq tokenize(N) = N .
  eq tokenize(e(X)) = 'X .
endfm

fmod DEF-COND-SEMANTICS is pr DEF-COND-SYNTAX .
  ex COND-EXP-SEMANTICS .
  var N : Identifier . var MT : MacroTable .
  eq evalB(defined N, MT) = isMacro(N, MT) .
  eq tokenize(defined N) = 'defined N .
endfm

fmod ARITH-EXP-SEMANTICS is pr ARITH-EXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  var X : Int . vars T T2 : Token .
  eq evalA(E + E', MT) = evalA(E, MT) + evalA(E', MT) .
  eq evalA(E - E', MT) = evalA(E, MT) - evalA(E', MT) .
  eq evalA(E * E', MT) = evalA(E, MT) * evalA(E', MT) .
  eq evalA(E / E', MT) = evalA(E, MT) quo evalA(E', MT) .
  eq evalA(E % E', MT) = evalA(E, MT) rem evalA(E', MT) .
  eq tokenize(E + E') = tokenize(E) '+ tokenize(E') .
  eq tokenize(E - E') = tokenize(E) '- tokenize(E') .
  eq tokenize(E * E') = tokenize(E) '* tokenize(E') .
  eq tokenize(E / E') = tokenize(E) '/ tokenize(E') .
  eq tokenize(E % E') = tokenize(E) '% tokenize(E') .
endfm

fmod BIT-EXP-SEMANTICS is pr BIT-EXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  eq evalA(E << E', MT) = evalA(E, MT) << evalA(E', MT) .
  eq evalA(E >> E', MT) = evalA(E, MT) >> evalA(E', MT) .
  eq evalA(E & E', MT) = evalA(E, MT) & evalA(E', MT) .
  eq evalA(E ^ E', MT) = evalA(E, MT) xor evalA(E', MT) .
  eq evalA(E | E', MT) = evalA(E, MT) | evalA(E', MT) .
  eq tokenize(E << E') = tokenize(E) '<< tokenize(E') .
  eq tokenize(E >> E') = tokenize(E) '>> tokenize(E') .
  eq tokenize(E & E') = tokenize(E) '& tokenize(E') .
  eq tokenize(E ^ E') = tokenize(E) '^ tokenize(E') .
  eq tokenize(E | E') = tokenize(E) '| tokenize(E') .
endfm

fmod REXP-SEMANTICS is pr REXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  eq evalB(E < E', MT) = (evalA(E, MT) < evalA(E', MT)) .
  eq evalB(E <= E', MT) = (evalA(E, MT) <= evalA(E', MT)) .
  eq evalB(E > E', MT) = (evalA(E, MT) > evalA(E', MT)) .
  eq evalB(E >= E', MT) = (evalA(E, MT) >= evalA(E', MT)) .
  eq evalB(E == E', MT) = (evalA(E, MT) == evalA(E', MT)) .
  eq evalB(E != E', MT) = (evalA(E, MT) =/= evalA(E', MT)) .
  eq tokenize(E < E') = tokenize(E) '< tokenize(E') .
  eq tokenize(E <= E') = tokenize(E) '<= tokenize(E') .
  eq tokenize(E > E') = tokenize(E) '> tokenize(E') .
  eq tokenize(E >= E') = tokenize(E) '>= tokenize(E') .
  eq tokenize(E == E') = tokenize(E) '== tokenize(E') .
  eq tokenize(E != E') = tokenize(E) '!= tokenize(E') .
```

```
endfm

fmod BEXP-SEMANTICS is pr BEXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars E E' : CondExp . var MT : MacroTable .
  eq evalB(! E, MT) = not evalB(E, MT) .
  eq evalB(E && E', MT) = evalB(E, MT) and evalB(E', MT) .
  eq evalB(E || E', MT) = evalB(E, MT) or evalB(E', MT) .
  eq tokenize(! E) = '! tokenize(E) .
  eq tokenize(E && E') = tokenize(E) '&& tokenize(E') .
  eq tokenize(E || E') = tokenize(E) '|| tokenize(E') .
endfm

fmod CEXP-SEMANTICS is pr CEXP-SYNTAX .
  ex COND-EXP-SEMANTICS .
  vars C E E' : CondExp . var MT : MacroTable .
  eq evalB(C ? E : E', MT) = if evalB(C, MT) then evalB(E, MT) else evalB(E', MT) fi .
  eq tokenize(C ? E : E') = tokenize(C) '? tokenize(E) ': tokenize(E') .
endfm

fmod MACRO-CALL-SEMANTICS is pr MACRO-CALL-SYNTAX .
  ex COND-EXP-SEMANTICS . pr MACRO-TABLE .
  var N : Identifier . var MT : MacroTable . var AP : ArgListP . var A : ArgList .
  ceq evalB(N, MT) = evalB(toCondExp(expand(MT[N])), MT) if isMacroWithoutArgs(N, MT) .
  ceq evalB(N AP, MT) = evalB(toCondExp(expandWithArgs(MT[N], AP)), MT) if isMacroWithArgs(N, MT) .
  ceq evalA(N, MT) = evalA(toCondExp(expand(MT[N])), MT) if isMacroWithoutArgs(N, MT) .
  ceq evalA(N AP, MT) = evalA(toCondExp(expandWithArgs(MT[N], AP)), MT) if isMacroWithArgs(N, MT) .
  var E : CondExp .
  op tokenize : ArgListP -> TokenSequence .
  op tokenize : ArgList -> TokenSequence .
  eq tokenize(()) = nil .
  eq tokenize(E, A) = tokenize(E) tokenize(A) .
endfm


fmod ALL-COND-EXP-SEMANTICS is
  pr DEF-COND-SEMANTICS .
  pr ARITH-EXP-SEMANTICS .
  pr BIT-EXP-SEMANTICS .
  pr REXP-SEMANTICS .
  pr BEXP-SEMANTICS .
  pr CEXP-SEMANTICS .
  pr MACRO-CALL-SEMANTICS .
endfm

--- --- --- ---
--- CPP-STATE ---

fmod CPP-STATE is
  pr MACRO-TABLE . pr TOKEN . pr STRINGS .
  sorts CppState CppStateAttribute .
  subsort CppStateAttribute < CppState .
  op empty : -> CppState .
  op _,_ : CppState CppState  -> CppState [assoc comm id: empty] .

  op includeDirs : StringSet -> CppStateAttribute .
  op macroTbl : MacroTable -> CppStateAttribute .
  op curMacroCalls : IdentifierListP -> CppStateAttribute .
  op skip : Bool -> CppStateAttribute .
  op nestLevelOfSkipped : Nat -> CppStateAttribute .
  op branchTaken : Bool -> CppStateAttribute .
  op outputStream : TokenSequence -> CppStateAttribute .
endfm

fmod HELPING-OPS is
  pr STRINGS . pr CPP-SYNTAX . pr CPP-STATE .
  sort MacroDefDirList .
  subsort MacroDefDir < MacroDefDirList .
```

```
  op nil : -> MacroDefDirList .
  op _;_ : MacroDefDirList MacroDefDirList -> MacroDefDirList [assoc id: nil] .

  op readFile : String -> LineSeq .
  --- This function reads in memory the source code of the file
  --- whose name is specified in the parameter and returns its contents

  op readFile : Identifier StringSet -> LineSeq .
  op readFile : String StringSet -> LineSeq .
  --- Idem previous except that the file is searched in the include
  --- directories specified in the second argument

  op initMacroTable : MacroDefDirList -> MacroTable .
  var M : Identifier . var TS : TokenSequence .
  var L : MacroDefDirList . var IdL : IdentifierList .
  eq initMacroTable(nil) = empty .
  eq initMacroTable((#define M TS cr) ; L) =
        [M : (name M replText TS)] initMacroTable(L) .
  eq initMacroTable((#define M ( IdL ) TS cr) ; L) =
        [M : (name M params ( IdL ) replText TS)] initMacroTable(L) .

  op initialCppState : StringSet MacroDefDirList -> CppState .
  var ID : StringSet .
  eq initialCppState(ID, L) = includeDirs(ID), macroTbl(initMacroTable(L)),
        curMacroCalls(‘(‘)), skip(false), nestLevelOfSkipped(0),
        branchTaken(false), outputStream(nil) .

  eq readFile("foo.c") =
    #include "foo.h" cr
    #define ’MAXTOKEN  ’100 cr
    #define ’INC(’X) (’X ’+ ’1) cr
    #define ’M3(’X, ’Y, ’Z) (’X ’> ’Y ’? ’X ’: ’Z) cr
    (’char ’token ’‘[ ’MAXTOKEN ’‘] ’;) cr
    (’int ’i ’= ’INC ’‘( ’i ’‘) ’;) cr
    (’float ’h ’= ’M3 ’‘( ’i ’‘, ’j ’‘, ’k ’‘) ’;) cr
    #include "foo.h" cr
    .
  eq readFile("foo.h", empty) =
    #if defined ’MAXTOKEN cr
    (’int ’max ’;) cr
    #else cr
    (’int ’min ’;) cr
    #endif cr
    .
endfm

--- --- --- ---
--- SEMANTICS ---

fmod CPP-DIR-SEMANTICS is pr CPP-DIR-SYNTAX .
  pr CPP-STATE .
  op state : CppDirective CppState -> CppState .
endfm

fmod INCLUDE-SEMANTICS is pr INCLUDE-SYNTAX .
  ex CPP-DIR-SEMANTICS . pr HELPING-OPS .
  pr MACRO-CALL-SEMANTICS .
  var FN : String . var S : CppState .
  var SS : StringSet . var I : Identifier . var MT : MacroTable .
  eq state(#include FN cr, (includeDirs(SS), S)) = state(readFile(FN, SS), (includeDirs(SS), S)) .
  eq state(#include < I > cr, (includeDirs(SS), S)) = state(readFile(I, SS), (includeDirs(SS), S)) .
  ceq state(#include I cr, (includeDirs(SS), macroTbl(MT), S))
    = state(readFile(string(expand(MT[I])), SS), (includeDirs(SS), macroTbl(MT), S))
    if isMacroWithoutArgs(I, MT) .
endfm

fmod DEFINE-SEMANTICS is pr DEFINE-SYNTAX .
  ex CPP-DIR-SEMANTICS .
```

```
  var I : Identifier . var TS : TokenSequence . var MT : MacroTable .
  var S : CppState . var IdL : IdentifierList .
  eq state(#define I TS cr, (macroTbl(MT), S))
    = macroTbl([I : (name I replText TS)] MT), S .
  eq state(#define I ( IdL ) TS cr, (macroTbl(MT), S))
    = macroTbl([I : (name I params (IdL) replText TS)] MT), S .
  ceq state(#undef I cr, (macroTbl(MT), S))
    = macroTbl(remove(MT[I], MT)), S if isMacro(I, MT) .
  eq state(#undef I cr, (macroTbl(MT), S)) = macroTbl(MT), S [owise] .
endfm

fmod COND-DIR-SEMANTICS is pr COND-DIR-SYNTAX .
  ex CPP-DIR-SEMANTICS .
  pr ALL-COND-EXP-SEMANTICS .
  var CE : CondExp . var N : Nat . var B : Bool . var AMT : MacroTable . var S : CppState .

  --- Case 1 of #if: Not skipping -> Not skipping
  ceq state(#if CE cr, (macroTbl(AMT), skip(false), branchTaken(false), S))
    = macroTbl(AMT), skip(false), branchTaken(true), S  if evalB(CE, AMT) = true .
  --- Case 2 of #if: Not skipping -> Skipping
  ceq state(#if CE cr, (macroTbl(AMT), skip(false), nestLevelOfSkipped(0), branchTaken(false), S))
    = macroTbl(AMT), skip(true), nestLevelOfSkipped(1), branchTaken(false), S
    if evalB(CE, AMT) = false .
  --- Case 3 of #if: Skipping -> Skipping
  eq state(#if CE cr, (skip(true), nestLevelOfSkipped(N), branchTaken(B), S))
    = skip(true), nestLevelOfSkipped(N + 1), branchTaken(false), S .

  --- Case 1, 2, 3 of #ifdef and #ifndef: idem

  --- Case 1 of #elif: Not skipping -> Skipping
  eq state(#elif CE cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S .
  --- Case 2 of #elif: Skipping -> Skipping
  ceq state(#elif CE cr, (macroTbl(AMT), skip(true), S))
    = macroTbl(AMT), skip(true), S  if evalB(CE, AMT) = false .
  --- Case 3 of #elif: Skipping -> Not skipping
  ceq state(#elif CE cr, (macroTbl(AMT), skip(true), nestLevelOfSkipped(1),
        branchTaken(false), S))
    = macroTbl(AMT), skip(false), nestLevelOfSkipped(0), branchTaken(true), S
        if evalB(CE, AMT) = true .

  --- Case 1 of #else: Not skipping -> Skipping
  eq state(#else‘cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S .
  --- Case 2 of #else: Skipping -> Skipping
  eq state(#else‘cr, (skip(true), nestLevelOfSkipped(N), branchTaken(true), S))
    = skip(true), nestLevelOfSkipped(N), branchTaken(true), S .
  --- Case 3 of #else: Skipping -> Not skipping
  eq state(#else‘cr, (skip(true), nestLevelOfSkipped(1), branchTaken(false), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(true), S .

  --- Case 1 of #endif: Not skipping -> Not skipping
  eq state(#endif‘cr, (skip(false), branchTaken(true), S))
    = skip(false), branchTaken(false), S .
  --- Case 2 of #endif: Skipping -> Skipping
  ceq state(#endif‘cr, (skip(true), nestLevelOfSkipped(N), S))
    = skip(true), nestLevelOfSkipped(N - 1), S   if N > 1 .
  --- Case 3 of #endif: Skipping -> Not Skipping
  eq state(#endif‘cr, (skip(true), nestLevelOfSkipped(1), branchTaken(true), S))
    = skip(false), nestLevelOfSkipped(0), branchTaken(false), S .
endfm


fmod LINE-SEQ-SEMANTICS is pr LINE-SEQ-SYNTAX .
  pr CPP-DIR-SEMANTICS . pr ALL-COND-EXP-SEMANTICS .
  op state : LineSeq CppState -> CppState .

  var L : Line .  var LS : LineSeq .  var S : CppState . var IL : IdentifierList .
```

```
  vars ILP ILP2 : IdentifierListP .
  vars T T2 : Token . vars TS O : TokenSequence . var MT : MacroTable . var I : Identifier .
  var MC : MacroCall . var AS : TokenSequence .
  eq state(nil cr, S) = S .
  eq state((’## TS) cr, (curMacroCalls( (I, IL) ), skip(false), S))
    = state(TS cr, (curMacroCalls( (IL) ), skip(false), S)) .
  eq state((’## TS) cr, (curMacroCalls( (I) ), skip(false), S))
    = state(TS cr, (curMacroCalls( () ), skip(false), S)) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), skip(false), outputStream(O), S))
    = state(TS cr, (macroTbl(MT), curMacroCalls(ILP), skip(false), outputStream(O T), S))
    if not(isMacro(T, MT)) or (T in ILP) .
  ceq state((T ’‘( AS ’‘) TS) cr, (macroTbl(MT), curMacroCalls(ILP), skip(false), S))
    = state((expandWithTSArgs(MT[T], toTokenSeqList(AS)) ’## TS) cr,
        (macroTbl(MT), curMacroCalls(cons(T, ILP)), skip(false), S))
    if isMacroWithArgs(T, MT) .
  ceq state((T TS) cr, (macroTbl(MT), curMacroCalls(ILP), skip(false), S))
    = state((expand(MT[T]) ’## TS) cr,
        (macroTbl(MT), curMacroCalls(cons(T, ILP)), skip(false), S))
    if isMacroWithoutArgs(T, MT) .
  eq state((T TS) cr, (skip(true), S)) = skip(true), S .

  eq state(nilLS, S) = S .
  eq state(L LS, (skip(false), S)) = state(LS, state(L, (skip(false), S))) .
  eq state(L LS, (skip(true), S)) = state(LS, state(L, (skip(true), S))) .
endfm

fmod CPP-SEMANTICS is
  pr CPP-SYNTAX . pr HELPING-OPS .
  pr INCLUDE-SEMANTICS . pr DEFINE-SEMANTICS . pr COND-DIR-SEMANTICS .
  pr LINE-SEQ-SEMANTICS .

  op preprocess : String StringSet MacroDefDirList -> TokenSequence .
  op returnOutput : CppState -> TokenSequence .

  var Name : String .
  vars IncludeDirs : StringSet .
  var ComLineMacros : MacroDefDirList .
  var O : TokenSequence . var S : CppState .
  eq preprocess(Name, IncludeDirs, ComLineMacros)
    = returnOutput(state(readFile(Name),
        initialCppState(IncludeDirs, ComLineMacros))) .
  eq returnOutput(outputStream(O), S) = O .
endfm
```

After loading the above specification, executing the line:

`red preprocess(‘‘foo.c’’, empty, nil) .`

at the command prompt in Maude, it make it “read” the file “foo.c” (specified in module HELPING-OPS), which exercises the different Cpp directives. The result is:

```
TokenSequence:
   ’int ’min ’;
   ’char ’token ’‘[ ’100 ’‘] ’;
   ’int ’i ’= ’i ’+ ’1 ’;
   ’float ’h ’= ’i ’> ’j ’? ’i ’: ’k ’;
   ’int ’max ’;
```

Appendix B

Maude Specification of P-Cpp

The following is the Maude specification of the semantics of P-Cpp.

```
in cpp-syntax.maude
--- ------------------ ---
--- SEMANTICS OF P-CPP ---
--- ------------------ ---

fmod CONDITIONS is
  pr ALL-COND-EXP-SYNTAX .
  sorts CppCondition CondStack CondStackStack CondSet CondPair CondPairSet .
  subsort CppCondition < CondStack .
  subsort CondStack < CondStackStack .
  subsort CppCondition < CondSet .
  subsort CondPair < CondPairSet .

  op condition : CondExp -> CppCondition [ctor] .
  op trueCondition : -> CppCondition [ctor] .
  op _and_ : CppCondition CppCondition -> CppCondition .
  op _isNegationOf_ : CppCondition CppCondition -> Bool .
  op compatible : CppCondition CppCondition CondPairSet -> Bool .
  op nil : -> CondStack .
  op _;_ : CondStack CondStack -> CondStack [assoc id: nil] .
  op nil : -> CondStackStack .
  op _;_ : CondStackStack CondStackStack -> CondStackStack [assoc id: nil] .
  op empty : -> CondSet .
  op __ : CondSet CondSet -> CondSet [assoc comm id: empty] .
  op _in_ : CppCondition CondSet -> Bool .
  op <_;_> : CppCondition CppCondition -> CondPair [ctor] .
  op empty : -> CondPairSet .
  op __ : CondPairSet CondPairSet -> CondPairSet [assoc comm id: empty] .
  op _in_ : CondPair CondPairSet -> Bool .
  op condFromStack : CondStack -> CppCondition .
  op condFromStackStack : CondStackStack -> CppCondition .
  op tokenize : CppCondition -> TokenSequence .

  vars C C' : CppCondition . var Incomp : CondPairSet . var S : CondStack .
  eq compatible(C, C', Incomp) = not (C isNegationOf C') and not (< C ; C' > in Incomp) .
  eq condFromStack(nil) = trueCondition .
  eq condFromStack(C ; nil) = C .
  eq condFromStack(trueCondition ; S) = condFromStack(S) .
  eq condFromStack(C ; S) = C and condFromStack(S) .
  var CE : CondExp .
  eq tokenize(condition(CE)) = tokenize(CE) .
endfm
```

```
fmod STRINGS is pr STRING .
  sorts StringSet StringList .
  subsort String < StringSet .
  subsort String < StringList .
  op nil : -> StringSet .
  op __ : StringSet StringSet -> StringSet [assoc comm id: nil].
  op nil : -> StringList .
  op _;_ : StringList StringList -> StringList [assoc id: nil].
endfm

fmod LOC is
  pr STRING . pr NAT .
  sort Location .
  op nilLoc : -> Location .  --- A nil location
  op file_offset_ : String Nat -> Location .
endfm

fmod TOKEN-TO-ARG is pr TOKEN .  --- exactly like Cpp
  sort TokenSeqList .
  subsort TokenSequence < TokenSeqList .
  op nilTSL : -> TokenSeqList .
  op _;_ : TokenSeqList TokenSeqList -> TokenSeqList [assoc id: nilTSL] .
  op size : TokenSeqList -> Nat .
  op elemAtTS : Nat TokenSeqList -> TokenSequence .
  op toTokenSeqList : TokenSequence -> TokenSeqList .
  var T : Token . vars TS1 TS2 TSA : TokenSequence . var TSL : TokenSeqList . var N : Nat .
  eq size(nilTSL) = 0 .
  eq size(TS1 ; TSL) = 1 + size(TSL) .
  eq elemAtTS(1, (TS1 ; TSL)) = TS1 .
  eq elemAtTS(s(N), (TS1 ; TSL)) = elemAtTS(N, TSL) .
  ceq toTokenSeqList(TS1) = TS1 if not (’‘, inTS TS1) .
  ceq toTokenSeqList(TS1 ’‘, TS2) = TS1 ; toTokenSeqList(TS2)
    if not (’‘, inTS TS1) and (’‘, inTS TS2) .
  ceq toTokenSeqList(TS1 ’‘, TS2) = TS1 ; TS2 if not (’‘, inTS TS1) and not (’‘, inTS TS2) .
endfm

fmod MACRO-DEF is
  pr STRINGS . pr DEFINE-SYNTAX . pr MACRO-CALL-SYNTAX . pr LOC .
  pr CONDITIONS . pr TOKEN-TO-ARG .
  sorts MacroDef MacroDefList MacroCallDescr MacroCallDescrList MacroCallStack .
  subsort MacroDef < MacroDefList .
  subsort MacroCallDescr < MacroCallDescrList .
  subsort MacroCallDescr < MacroCallStack .
  op nil : -> MacroDefList .
  op _,_ : MacroDefList MacroDefList -> MacroDefList [assoc comm id: nil] .
  op nil : -> MacroCallDescrList .
  op _;_ : MacroCallDescrList MacroCallDescrList -> MacroCallDescrList [assoc comm id: nil] .
  op nil : -> MacroCallStack .
  op __ : MacroCallStack MacroCallStack -> MacroCallStack [assoc id: nil] .

  op name_def_defLoc_condition_calls_undefLoc_ :
    Identifier MacroDefDir Location CppCondition MacroCallDescrList Location -> MacroDef .
  op name_def_ : Identifier MacroDefDir -> MacroDef . *** to create a command line macro
  op name_def_defLoc_condition_ : Identifier MacroDefDir Location CppCondition -> MacroDef .
  op macroDefs_args_loc_ : MacroDefList StringList Location -> MacroCallDescr .

  op name : MacroDef -> Identifier .
  op hasArgs : MacroDef -> Bool .
  op guardCond : MacroDef -> CppCondition .
  op expand : MacroDef -> TokenSequence .               --- idem Cpp
  op expandWithArgs : MacroDef ArgListP -> TokenSequence .  --- idem Cpp
  op expandWithTSArgs : MacroDef TokenSeqList -> TokenSequence . --- idem Cpp
  op undef : MacroDef Location -> MacroDef .
  op addCall : MacroDef MacroCallDescr -> MacroDef .
  op findWithGuardCond : MacroDefList CppCondition -> MacroDef .
  op addCallToAll : MacroDefList MacroCallDescr -> MacroDefList .
  op expandMacroCall : MacroCallDescr -> TokenSequence .
  op expMC-rec : MacroDefList -> TokenSequence .
```

```
  op andGuardCond : MacroDef CppCondition -> MacroDef .
  op andGuardCondToAll : MacroDefList CppCondition -> MacroDefList .
  op nameIn : MacroCallStack Identifier -> Bool .

  var N : Identifier . vars C C' : CppCondition . vars L UL L' : Location .
  var MCL : MacroCallDescrList . var D : MacroDefDir . var TS : TokenSequence .
  var MC : MacroCallDescr . var M : MacroDef . var MDL : MacroDefList .
  eq name N def D = name N def D defLoc nilLoc condition trueCondition calls nil undefLoc nilLoc .
  eq name N def D defLoc L condition C = name N def D defLoc L condition C calls nil undefLoc nilLoc .
  eq guardCond(name N def D defLoc L condition C calls MCL undefLoc UL) = C .
  eq undef(name N def D defLoc L condition C calls MCL undefLoc UL, L') =
    name N def D defLoc L condition C calls MCL undefLoc L' .
  eq addCall(name N def D defLoc L condition C calls MCL undefLoc UL, MC)
     = name N def D defLoc L condition C calls (MCL ; MC) undefLoc UL .
  eq addCallToAll(nil, MC) = nil .
  eq addCallToAll((M , MDL), MC) = addCall(M, MC) , addCallToAll(MDL, MC) .
  eq andGuardCond(name N def D defLoc L condition C calls MCL undefLoc UL, C')
     = name N def D defLoc L condition (C and C') calls MCL undefLoc UL .
  eq andGuardCondToAll(nil, C) = nil .
  eq andGuardCondToAll((M , MDL), C) = andGuardCond(M, C) , andGuardCondToAll(MDL, C) .
  eq expandMacroCall(macroDefs M args nil loc L) = expand(M) .
  eq expandMacroCall(macroDefs (M , MDL) args nil loc L)
    =  '#if tokenize(guardCond(M)) 'cr
       expand(M) 'cr
      expMC-rec(MDL)
      '#endif 'cr .
  eq expMC-rec(nil) = nil .
  eq expMC-rec(M , MDL) = '#elif tokenize(guardCond(M)) 'cr
                    expand(M) 'cr
                   expMC-rec(MDL) .
endfm


fmod MACRO-TABLE is
  pr MACRO-DEF .
  sort MacroTable .
  op empty : -> MacroTable .
  op [_:_] : Identifier MacroDefList -> MacroTable .
  op __ : MacroTable MacroTable -> MacroTable [assoc comm id: empty] .
  op _[_] : MacroTable Identifier -> MacroDefList .
  op _[_under_] : MacroTable Identifier CppCondition -> MacroDef .
  op _[_<-_] : MacroTable Identifier MacroDefList -> MacroTable .
  op isMacro : Identifier MacroTable -> Bool .
  op remove : MacroDef MacroTable -> MacroTable .
  op andConditionToAll : MacroTable CppCondition -> MacroTable .

  var N : Identifier . vars L L' : MacroDefList .
  vars M M' : MacroDef . var C : CppCondition .
  var MT : MacroTable .
  eq ([N : L] MT)[N] = L .
  eq ([N : L] MT)[N under C] = findWithGuardCond(L, C) .
  eq ([N : L'] MT)[N <- L] = [N : L] MT .
  eq MT[N <- L] = MT [N : L] [owise] .
  eq isMacro(N, empty) = false .
  eq isMacro(N, ([N : L] MT)) = true .
  eq andConditionToAll(([N : L] MT), C) = [N : andGuardCondToAll(L, C)] andConditionToAll(MT, C) .
endfm

fmod COND-EXP-SEMANTICS is
  pr COND-EXP-SYNTAX . pr CONDITIONS .
  op eval : CondExp -> CppCondition .
  var X : Int . var CE : CondExp .
  ceq eval(e(X)) = trueCondition    if X =/= 0 .
  eq eval(CE) = condition(CE) .
endfm

fmod CONFIG is
  pr STRINGS .
```

```
  sort CRConfiguration .
  op fileNames_includeDirs_commandLineMacros_falseConds_incompatConds_ :
    StringSet StringSet StringSet StringSet StringSet -> CRConfiguration .
endfm

fmod CR-TOKEN is pr TOKEN .
  pr MACRO-DEF . pr CONDITIONS .
  sort CRToken .
  sort CRTokenStream .
  subsort CRToken < CRTokenStream .
  op value_ : Token -> CRToken .
  op value_macroCalls_cond_ :
    Token MacroCallStack CppCondition -> CRToken .
  op empty : -> CRTokenStream .
  op __ : CRTokenStream CRTokenStream -> CRTokenStream [assoc id: empty] .
endfm

fmod INCLUDE-DEP-GRAPH is
  pr CR-TOKEN . pr MACRO-TABLE . pr CONDITIONS . pr STRINGS .
  sorts ProgramFile ProgramFileStack IncludeDepGraph IdgEdge IdgEdgeList .
  subsort ProgramFile < IncludeDepGraph .
  subsort ProgramFile < ProgramFileStack .
  subsort IdgEdge < IdgEdgeList .
  op empty : -> IncludeDepGraph .
  op __ : IncludeDepGraph IncludeDepGraph -> IncludeDepGraph [assoc comm id: empty] .
  op nil : -> ProgramFileStack .
  op _;_ : ProgramFileStack ProgramFileStack -> ProgramFileStack [assoc id: nil] .

  op includes : IncludeDepGraph String -> Bool .
  var PF : ProgramFile . var IDG : IncludeDepGraph . var N : String .
  eq includes(empty, N) = false .
  eq includes(PF IDG, N) = (name(PF) == N) or includes(IDG, N) .

  op dest_pos_under_ : ProgramFile Location CppCondition -> IdgEdge .
  op nil : -> IdgEdgeList .
  op _,_ : IdgEdgeList IdgEdgeList -> IdgEdgeList [assoc id: nil] .
  op name_tokenStream_activeMacrosAtStart_macrosDefined_Csymbols_includingFiles_includedFiles_ :
    String CRTokenStream MacroTable MacroTable StringSet IdgEdgeList IdgEdgeList
        -> ProgramFile [ctor] .
  op programFile : String -> ProgramFile [ctor] .
  op programFileWithName : String -> ProgramFile .   --- returns PF with that name
  op appendOutputToken : ProgramFile CRToken -> ProgramFile .
  op addEdgeFrom_to_at_under_ : ProgramFile ProgramFile Location CppCondition -> ProgramFile .
  op name : ProgramFile -> String .
  op addMacroDefinition : ProgramFile MacroDef -> ProgramFile .
  op macrosDefInPredsOf : ProgramFile -> MacroTable .
  op macrosDefIn : IdgEdgeList -> MacroTable .

  vars FN FN' : String . vars TS TS' : CRTokenStream . vars AM MD AM' MD' : MacroTable .
  vars SS SS' : StringSet . vars Suc Pred Suc' Pred' : IdgEdgeList .
  var T : CRToken . var L : Location . var C : CppCondition .
  eq programFile(FN) = name FN tokenStream empty activeMacrosAtStart empty
        macrosDefined empty Csymbols nil includingFiles nil includedFiles nil .
  eq appendOutputToken(name FN tokenStream TS activeMacrosAtStart AM
        macrosDefined MD Csymbols SS includingFiles Suc includedFiles Pred, T)
    = name FN tokenStream TS T activeMacrosAtStart AM
        macrosDefined MD Csymbols SS includingFiles Suc includedFiles Pred .
  eq addEdgeFrom (name FN tokenStream TS activeMacrosAtStart AM macrosDefined MD
        Csymbols SS includingFiles Suc includedFiles Pred)
    to (name FN' tokenStream TS' activeMacrosAtStart AM' macrosDefined MD'
        Csymbols SS' includingFiles Suc' includedFiles Pred')
    at L under C
    = name FN tokenStream TS activeMacrosAtStart AM macrosDefined MD Csymbols SS
        includingFiles (Suc, (dest
        (name FN' tokenStream TS' activeMacrosAtStart AM' macrosDefined MD'
        Csymbols SS' includingFiles Suc' includedFiles (Pred' ,
                        dest programFileWithName(FN) pos L under C))
        pos L under C)) includedFiles Pred .
```

```
  eq name(name FN tokenStream TS activeMacrosAtStart AM
        macrosDefined MD Csymbols SS includingFiles Suc includedFiles Pred) = FN .

  eq macrosDefInPredsOf(name FN tokenStream TS activeMacrosAtStart AM
        macrosDefined MD Csymbols SS includingFiles Suc includedFiles Pred)
    = MD macrosDefIn(Pred).
  eq macrosDefIn(nil) = empty .
  eq macrosDefIn(((dest PF pos L under C), Pred))
    = andConditionToAll(macrosDefInPredsOf(PF), C) macrosDefIn(Pred) .
endfm

fmod LINE-SEQ-STACK is pr LINE-SEQ-SYNTAX .
  sort LineSeqStack .
  subsort LineSeq < LineSeqStack .
  op nil : -> LineSeqStack .
  op _;_ : LineSeqStack LineSeqStack -> LineSeqStack [assoc id: nil] .
endfm

--- --- --- ---
--- PCPP-STATE ---

fmod PCPP-STATE is
  pr CONFIG . pr LINE-SEQ-STACK . pr INCLUDE-DEP-GRAPH .
  pr CONDITIONS . pr MACRO-TABLE .
  pr STRINGS .
  sorts PcppState PcppStateAttribute .
  subsort PcppStateAttribute < PcppState .
  op empty : -> PcppState .
  op _,_ : PcppState PcppState  -> PcppState [assoc comm id: empty] .

  op inputStack : LineSeqStack -> PcppStateAttribute .
  op fileNames : StringSet -> PcppStateAttribute .
  op includeDirs : StringSet -> PcppStateAttribute .
  op commandLineMs : StringSet -> PcppStateAttribute .
  op falseConds : CondSet -> PcppStateAttribute .
  op incompatConds : CondPairSet -> PcppStateAttribute .
  op idg : IncludeDepGraph -> PcppStateAttribute .
  op curPF : ProgramFileStack -> PcppStateAttribute .
  op macroTbl : MacroTable -> PcppStateAttribute .
  op curMacroStack : MacroCallStack -> PcppStateAttribute .
  op curCond : CondStackStack -> PcppStateAttribute .
  op skip : Bool -> PcppStateAttribute .
  op nestLevelOfSkipped : Nat -> PcppStateAttribute .
  op curLoc : Location -> PcppStateAttribute .
  op Csymbols : StringSet -> PcppStateAttribute .
endfm

fmod HELPING-OPS is
  pr STRINGS . pr CPP-SYNTAX . pr PCPP-STATE .
  sort MacroDefDirList .
  subsort MacroDefDir < MacroDefDirList .
  op nil : -> MacroDefDirList .
  op _;_ : MacroDefDirList MacroDefDirList -> MacroDefDirList [assoc id: nil] .

  op readFile : String -> LineSeq . --- idem Cpp
  op readFile : String StringSet -> LineSeq . --- idem Cpp

  op initialCppState : CRConfiguration -> PcppState .
  op createFCondsSet : StringSet -> CondSet .
  op createIConds : StringSet -> CondPairSet .
  op initMacroTable : StringList -> MacroTable .
  op subsetCompatible : MacroDefList CppCondition CondPairSet -> MacroDefList .
  op subsetIncompat : MacroDefList CppCondition CondPairSet -> MacroDefList .

  var M : MacroDef . var MDL : MacroDefList . var C : CppCondition .
  var IC : CondPairSet .
  eq subsetCompatible(nil, C, IC) = nil .
  ceq subsetCompatible((M , MDL), C, IC) = M , subsetCompatible(MDL, C, IC)
```

```
    if compatible(guardCond(M), C, IC) .
  eq subsetCompatible((M , MDL), C, IC) = subsetCompatible(MDL, C, IC) [owise] .

  vars FNs IDs Ms FCs ICs : StringSet .
  var IDG : IncludeDepGraph . var S : PcppState .
  eq initialCppState(fileNames (FNs) includeDirs IDs commandLineMacros Ms falseConds FCs
        incompatConds ICs)
    = inputStack(nil),
        fileNames(FNs),
        includeDirs(IDs),
        commandLineMs(Ms),
        falseConds(createFCondsSet(FCs)),
        incompatConds(createIConds(ICs)),
        idg(empty),
        macroTbl(initMacroTable(Ms)),
        curPF(nil),
        curMacroStack(nil),
        curCond(nil),
        skip(false),
        nestLevelOfSkipped(0),
        curLoc(nilLoc),
        Csymbols(nil) .
endfm

--- --- --- ---
--- SEMANTICS ---

fmod PCPP-DIR-SEMANTICS is pr CPP-DIR-SYNTAX .
  pr PCPP-STATE .
  op state : CppDirective PcppState -> PcppState .
endfm

fmod INCLUDE-SEMANTICS is pr INCLUDE-SYNTAX .
  ex PCPP-DIR-SEMANTICS . pr HELPING-OPS .
  var LS : LineSeq . var LSS : LineSeqStack .
  var FN : String . var S : PcppState . var LO : Location . var MT : MacroTable .
  var Dirs : StringSet . var IDG : IncludeDepGraph . var T : CRToken .
  var PFS : ProgramFileStack . vars PF PF' : ProgramFile .
  var CS : CondStack . var CSS : CondStackStack . var MCS : MacroCallStack .

  ceq state(#include FN cr LS, (inputStack(LSS), includeDirs(Dirs), idg(PF IDG),
        curPF(PF ; PFS), curMacroStack(MCS), curCond(CS ; CSS), curLoc(LO), S))
    = state(readFile(FN, Dirs), (inputStack(LS ; LSS), includeDirs(Dirs),
        idg((addEdgeFrom programFile(FN) to PF at LO under condFromStack(CS)) PF IDG),
        curPF(programFileWithName(FN) ;
            appendOutputToken(PF, value qid("#include" + FN) macroCalls MCS cond condFromStack(CS)) ;
            PFS),
        curMacroStack(MCS), curCond(nil ; CS ; CSS), curLoc(LO), S))
    if not includes(IDG, FN) .

  ceq state(#include FN cr, (idg(PF' PF IDG), curPF(PF ; PFS), macroTbl(MT),
        curMacroStack(MCS), curCond(CS ; CSS), curLoc(LO), S))
    = idg((addEdgeFrom PF' to PF at LO under condFromStack(CS)) PF IDG),
      curPF(appendOutputToken(PF, value qid("#include" + FN) macroCalls MCS cond condFromStack(CS))
          ; PFS),
      macroTbl(MT macrosDefInPredsOf(PF')), curMacroStack(MCS), curCond(CS ; CSS), curLoc(LO), S
    if name(PF') == FN .
endfm

fmod DEFINE-SEMANTICS is pr DEFINE-SYNTAX .
  ex PCPP-DIR-SEMANTICS .
  var I : Identifier . var TS : TokenSequence . var L : Location .
  var PF : ProgramFile . var PFS : ProgramFileStack . var MT : MacroTable .
  var S : PcppState . var CS : CondStack . var CSS : CondStackStack .

  eq state(#define I TS cr, (curPF(PF ; PFS), macroTbl(MT), curCond(CS ; CSS), curLoc(L), S))
    = curPF(addMacroDefinition(
            appendOutputToken(PF, value qid("#define") macroCalls nil cond condFromStack(CS)),
```

```
            (name I def (#define I TS cr) defLoc L condition condFromStack(CS)))
          ; PFS),
      macroTbl([I : (name I def (#define I TS cr) defLoc L condition condFromStackStack(CS ; CSS))] MT),
      curCond(CS ; CSS), curLoc(L), S .

  --- similarly for a macro with arguments
endfm

fmod COND-DIR-SEMANTICS is pr COND-DIR-SYNTAX .
  ex PCPP-DIR-SEMANTICS .  pr COND-EXP-SEMANTICS .
  var CE : CondExp . var FC : CondSet . var CS : CondStack . var CSS : CondStackStack .
  var S : PcppState . var N : Nat . var PF : ProgramFile . var PFS : ProgramFileStack .
  --- Case 1 of #if: Not skipping -> Not skipping
  ceq state(#if CE cr, (falseConds(FC), curCond(CS ; CSS), skip(false), curPF(PF ; PFS), S))
    = falseConds(FC), curCond((eval(CE) ; CS) ; CSS), skip(false),
       curPF(appendOutputToken(PF, value '#if macroCalls nil cond condFromStack(CS)) ; PFS), S
    if not(eval(CE) in FC) .
  --- Case 2 of #if: Not skipping -> Skipping
  eq state(#if CE cr, (skip(false), nestLevelOfSkipped(0), S))
    = skip(true), nestLevelOfSkipped(1), S  [owise] .
  --- Case 3 of #if: Skipping -> Skipping
  eq state(#if CE cr, (skip(true), nestLevelOfSkipped(N), S))
    = skip(true), nestLevelOfSkipped(N + 1), S .
endfm


fmod LINE-SEQ-SEMANTICS is pr LINE-SEQ-SYNTAX .
  ex PCPP-DIR-SEMANTICS . pr HELPING-OPS .
  op state : LineSeq PcppState -> PcppState .
  var L : Line .  var LS : LineSeq .  var S : PcppState . var T : Token .
  vars TS O : TokenSequence . var MT : MacroTable . var MCS : MacroCallStack .
  var LSS : LineSeqStack . var CS : CondStack . var CSS : CondStackStack .
  var PF : ProgramFile . var PFS : ProgramFileStack . var LOC : Location .
  vars MDL CompatDefs IncomDefs : MacroDefList . var MC : MacroCallDescr .
  var IC : CondPairSet .
  ceq state((T TS) cr, (curPF(PF ; PFS), macroTbl(MT), curMacroStack(MCS),
            curCond(CS ; CSS), skip(false), S))
    = state(TS cr, (curPF(appendOutputToken(PF, value T macroCalls MCS cond condFromStack(CS))
                    ; PFS),
        macroTbl(MT), curMacroStack(MCS), curCond(CS ; CSS), skip(false), S))
    if not (isMacro(T, MT)) or nameIn(MCS, T).

  ceq state((T TS) cr, (inputStack(LSS), incompatConds(IC), skip(false), curLoc(LOC),
            macroTbl([T : CompatDefs, IncomDefs] MT), curMacroStack(MCS),
            curCond(CSS), S))
    = state(expandMacroCall(macroDefs CompatDefs args nil loc LOC) cr,
        (inputStack((TS cr) ; LSS), incompatConds(IC), skip(false), curLoc(LOC),
        macroTbl([T :
            addCallToAll(CompatDefs, macroDefs CompatDefs args nil loc LOC),
            IncomDefs] MT),
        curMacroStack((macroDefs CompatDefs args nil loc LOC) MCS),
        curCond(CSS),  S))
    if (subsetCompatible(MDL, condFromStackStack(CSS), IC)) := CompatDefs
        and (subsetIncompat(MDL, condFromStackStack(CSS), IC) := IncomDefs) .

  eq state(nil cr, (inputStack(LS ; LSS), skip(false), curMacroStack(MC MCS), S))
    = state(LS, (inputStack(LSS), skip(false), curMacroStack(MCS), S)) .
  eq state((T TS) cr LS, (skip(true), curPF(PF ; PFS), S))
    = state(TS cr LS, (skip(true),
        curPF(appendOutputToken(PF, value qid(string(T) + "comment")) ; PFS), S)) .

  eq state(L LS, (skip(false), S)) = state(LS, state(L, (skip(false), S))) .
  eq state(L LS, (skip(true), S)) = state(LS, state(L, (skip(true), S))) .
endfm


fmod PCPP-SEMANTICS is
  pr CPP-SYNTAX . pr CONFIG . pr HELPING-OPS . pr LINE-SEQ-SEMANTICS .
```

```
  op preprocess : CRConfiguration -> IncludeDepGraph .
  op returnOutput : PcppState -> IncludeDepGraph .

  var CO : CRConfiguration . var N : String . vars FNs Ms SY : StringSet .
  var CS : CondStack . var CSS : CondStackStack .
  var IDG : IncludeDepGraph . var MT : MacroTable . var LO : Location . var S : PcppState .
  var LS : LineSeq . var LSS : LineSeqStack . var PF : ProgramFile . var PFS : ProgramFileStack .
  eq preprocess(CO) = returnOutput(state(nilLS, initialCppState(CO))) .
  eq state(nilLS, (fileNames(N ; FNs), idg(IDG), curPF(nil), curLoc(LO), S))
    = state(readFile(N), (fileNames(FNs), idg(IDG programFile(N)),
        curPF(programFile(N)), curLoc(file N offset 1), S)) .
  eq state(nilLS, (inputStack(LS ; LSS), curPF(PF ; PFS), curMacroStack(nil), curCond(CS ; CSS), S))
    = state(LS, (inputStack(LSS), curPF(PFS), curMacroStack(nil), curCond(CSS), S)) .
  eq state(nilLS, (inputStack(nil), fileNames(nil), S)) = S .
  eq returnOutput(idg(IDG), S) = IDG .
endfm
```

Appendix C

C Grammar with Cpp extensions

This Appendix lists the grammar used by CRefactory's parser. CRefactory uses SmallCC, a parser generator for Smalltalk, to generate the parser from this grammar specification.

The grammar specification uses standard notation, where each grammar production names the non-terminal represented, then a semi-colon, and then the different alternatives separated by a '|' character. Terminals of the grammar are surrounded by angle brackets. The scripts that create AST nodes appear in between curly braces. They have Smalltalk code that creates a node, listing the node's class name first and a message that sets the parts of the node. The expression '1' represents the first element in the right-hand side of the production, '2' represents the second, and so on.

The grammar looks mostly like the standard ANSI-C grammar, although there are additions to parse Cpp directives and some GCC extensions, like assembler instructions and statement expressions.

```
translationUnit
    : externalDeclaration
        {CRTranslationUnitNode with: '1' first}
    | translationUnit externalDeclaration
        {'1' add: '2' first; yourself}
    ;

externalDeclaration
    : functionDefinition
```

```
    | declaration
    | controlLine
    ;

functionDefinition
    : declarationSpecifiers declarator declarationList compoundStatement
        {|func| func := CRFunctionDefinitionNode declarationSpecifiers: '1' declarator: '2'
                declarationList: '3' compoundStatement: '4'.
         scanner hasComments ifTrue: [func comments: scanner getComments].
         func}
    | declarator declarationList compoundStatement
        {|func| func := CRFunctionDefinitionNode declarator: '1'
                declarationList: '2' compoundStatement: '3'.
         scanner hasComments ifTrue: [func comments: scanner getComments].
         func}
    | declarationSpecifiers declarator compoundStatement
        {|func| func := CRFunctionDefinitionNode declarationSpecifiers: '1' declarator: '2'
                compoundStatement: '3'.
         scanner hasComments ifTrue: [func comments: scanner getComments].
         func}
    | declarator compoundStatement
        {|func| func := CRFunctionDefinitionNode declarator: '1' compoundStatement: '2'.
         scanner hasComments ifTrue: [func comments: scanner getComments].
         func}
    ;

declarationSpecifiers
    : declarationSpecifiers declarationSpec
        {'1' add: '2'; yourself}
    | declarationSpec
        {|node| node := CRSpecifiersNode with: '1'.
         scanner hasComments ifTrue: [node comments: scanner getComments].
         node}
    ;

declarationSpec
    : storageClassSpecifier
        {CRStorageClassSpecifier value: '1' first}
    | typeQualifier
        {'1'}
    | typeSpecifier
        {'1'}
    ;

storageClassSpecifier
    : <AUTO>
    | <REGISTER>
    | <STATIC>
    | <EXTERN>
    | <INLINE>
    | <TYPEDEF>
        {scanner typedefSeen.
        OrderedCollection with: '1'}
    ;

typeQualifier
    : <CONST>
        {CRTokenSpecifier value: '1'}
    | <VOLATILE>
        {CRTokenSpecifier value: '1'}
    ;

declarator
    : pointer directDeclarator
        {CRDeclaratorNode pointer: '1' directDeclarator: '2'}
    | directDeclarator
        {CRDeclaratorNode directDeclarator: '1'}
    ;
```

```
pointer
    : <STAR_OP>
        {CRPointerNode operator: '1'}
    | <STAR_OP> typeQualifierList
        {CRPointerNode operator: '1' typeQualifierList: '2'}
    | <STAR_OP> pointer
        {CRPointerNode operator: '1' pointer: '2'}
    | <STAR_OP> typeQualifierList pointer
        {CRPointerNode operator: '1' typeQualifierList: '2' pointer: '3'}
    ;

typeQualifierList
    : typeQualifier
        {CRCollectionNode with: '1'}
    | typeQualifierList typeQualifier
        {'1' add: '2'; yourself}
    ;

directDeclarator
    : identifierToken
        {self checkForTypedef: '1' token.
         '1'}
    | <LEFT_PAREN> declarator <RIGHT_PAREN>
        {CRParenthesizedDeclaratorNode leftParen: '1' declarator: '2' rightParen: '3'}
    | directDeclarator <LEFT_BLOCK> constantExpressionOrNil <RIGHT_BLOCK>
        {CRArrayDeclaratorNode directDeclarator: '1' leftBrace: '2' constantExpression: '3'
            rightBrace: '4'}
    | directDeclarator <LEFT_PAREN> parameterTypeList <RIGHT_PAREN>
        {CRFunctionTypedDeclaratorNode directDeclarator: '1' leftParen: '2' parameterTypeList: '3'
            rightParen: '4'}
    | directDeclarator <LEFT_PAREN> identifierList <RIGHT_PAREN>
        {CRFunctionDeclaratorNode directDeclarator: '1' leftParen: '2' identifierList: '3'
            rightParen: '4'}
    | directDeclarator <LEFT_PAREN> <RIGHT_PAREN>
        {CRFunctionDeclaratorNode directDeclarator: '1' leftParen: '2' rightParen: '3'}
    ;

identifierToken : <IDENTIFIER>
        {CRIdentifierNode value: '1'}
    ;

typeSpecifier
    : basicTypeSpecifier
        {CRBasicTypeSpecifier value: '1' first}
    | structOrUnionSpecifier
        {'1'}
    | enumSpecifier
        {'1'}
    | <TYPE_NAME>
        {CRTypedefNameNode value: '1'}
    ;

basicTypeSpecifier : <VOID>
    | <CHAR>
    | <SHORT>
    | <INT>
    | <LONG>
    | <FLOAT>
    | <DOUBLE>
    | <SIGNED>
    | <UNSIGNED>;

structOrUnionSpecifier
    : structOrUnionToken identifierToken
        {CRStructSpecifier tokenSpecifier: '1' identifier: '2'}
    | structOrUnionToken identifierToken <LEFT_BRACE> structDeclarationList <RIGHT_BRACE>
        {CRStructSpecifier tokenSpecifier: '1' identifier: '2' leftBrace: '3' structDeclarationList: '4'
            rightBrace: '5'}
```

```
    | structOrUnionToken <LEFT_BRACE> structDeclarationList <RIGHT_BRACE>
        {CRStructSpecifier tokenSpecifier: '1' leftBrace: '2' structDeclarationList: '3' rightBrace: '4'}
# this added to make it work in stdio.h
    | structOrUnionToken <TYPE_NAME>
        {CRStructSpecifier tokenSpecifier: '1' identifier: (CRTypedefNameNode value: '2')}
# this added to make it work in sysctl.h
    | structOrUnionToken <TYPE_NAME> <LEFT_BRACE> structDeclarationList <RIGHT_BRACE>
        {CRStructSpecifier tokenSpecifier: '1' identifier: (CRTypedefNameNode value: '2') leftBrace: '3'
            structDeclarationList: '4' rightBrace: '5'}
    ;

structOrUnionToken
    : <STRUCT>
        {CRStructOrUnionTokenSpecifier value: '1'}
    | <UNION>
        {CRStructOrUnionTokenSpecifier value: '1'}
    ;

structDeclarationList
    : structDeclaration
        {CRCollectionNode with: '1'}
    | structDeclarationList structDeclaration
        {'1' add: '2'}
    ;

structDeclaration
    : specifierQualifierList structDeclaratorList <SEMI_COLON>
        {CRStructDeclarationNode specifierQualifierList: '1' structDeclaratorList: '2' semiColon: '3'}
    | conditionalDirective
        {'1'}
    | macroDirective
        {'1'}
    ;

specifierQualifierList
    : typeSpecifier specifierQualifierList
        {'2' addFirst: '1'}
    | typeSpecifier
        {CRSpecifiersNode with: '1'}
    | typeQualifier specifierQualifierList
        {'2' addFirst: '1'}
    | typeQualifier
        {CRSpecifiersNode with: '1'}
    ;

structDeclaratorList
    : structDeclarator
        {CRCommaCollectionNode with: '1'}
    | structDeclaratorList <COMMA> structDeclarator
        {'1' add: '3' afterComma: '2'}
    ;

structDeclarator
    : declarator
        {CRStructDeclaratorNode declarator: '1'}
    | <COLON> constantExpression
        {CRStructDeclaratorNode colon: '1' constantExpression: '2' first}
    | declarator <COLON> constantExpression
        {CRStructDeclaratorNode declarator: '1' colon: '2' constantExpression: '3' first}
    ;

enumSpecifier
    : <ENUM> identifierToken <LEFT_BRACE> enumeratorList <RIGHT_BRACE>
        {|eNode| eNode := CREnumSpecifier enumToken: '1' identifier: '2' leftBrace: '3'
                            enumeratorList: '4' rightBrace: '5'.
         scanner possibleTypedef: '2' token.
         scanner endTypedef.
         eNode
```

```
        }
    | <ENUM> <LEFT_BRACE> enumeratorList <RIGHT_BRACE>
        {CREnumSpecifier enumToken: '1' leftBrace: '2' enumeratorList: '3' rightBrace: '4'}
    | <ENUM> identifierToken
        {|eNode| eNode := CREnumSpecifier enumToken: '1' identifier: '2'.
         scanner possibleTypedef: '2' token.
         scanner endTypedef.
         eNode
        }
    ;

enumeratorList
    : {CRCommaCollectionNode new}
    | enumerator
        {CRCommaCollectionNode with: '1'}
    | enumerator <COMMA> enumeratorList
        {'3' addFirst: '1' beforeComma: '2'}
    | controlLine enumeratorList
        {'2' addFirst: '1' beforeComma: nil}
    | enumerator controlLine enumeratorList
        {'3' addFirst: '2' beforeComma: nil.
        '3' addFirst: '1' beforeComma: nil}
    ;

enumerator
    : identifierToken
        {CREnumeratorNode identifier: '1'}
    | identifierToken <EQUALS_OP> constantExpression
        {CREnumeratorNode identifier: '1' equalsOperator: '2' constantExpression: '3' first}
    ;

typeName
    : specifierQualifierList
        {CRTypeNameNode specifierQualifierList: '1'}
    | specifierQualifierList abstractDeclarator
        {CRTypeNameNode specifierQualifierList: '1' abstractDeclarator: '2'}
    ;

abstractDeclarator
    : pointer
        {CRAbstractDeclaratorNode pointer: '1'}
    | pointer directAbstractDeclarator
        {CRAbstractDeclaratorNode pointer: '1' directAbstractDeclarator: '2'}
    | directAbstractDeclarator
        {CRAbstractDeclaratorNode directAbstractDeclarator: '1'}
    ;

directAbstractDeclarator
    : <LEFT_PAREN> abstractDeclarator <RIGHT_PAREN>
        {CRParenthesizedDeclaratorNode leftParen: '1' declarator: '2' rightParen: '3'}
    | directAbstractDeclarator <LEFT_BLOCK> constantExpressionOrNil <RIGHT_BLOCK>
        {CRAbstractArrayDeclaratorNode directAbstractDeclarator: '1' leftBrace: '2'
            constantExpressionOrNil: '3' rightBrace: '4'}
    | <LEFT_BLOCK> constantExpressionOrNil <RIGHT_BLOCK>
        {CRAbstractArrayDeclaratorNode leftBrace: '1' constantExpressionOrNil: '2' rightBrace: '3'}
    | directAbstractDeclarator <LEFT_PAREN> parameterTypeListOrNil <RIGHT_PAREN>
        {CRAbstractFunctionTypedDeclaratorNode directAbstractDeclarator: '1' leftParen: '2'
            parameterTypeListOrNil: '3' rightParen: '4'}
    | <LEFT_PAREN> parameterTypeListOrNil <RIGHT_PAREN>
        {CRAbstractFunctionTypedDeclaratorNode leftParen: '1' parameterTypeListOrNil: '2' rightParen: '3'}
    ;

constantExpressionOrNil
    : constantExpression {'1' first}
    |  {nil}
    ;

parameterTypeListOrNil
```

```
    : parameterTypeList {'1'}
    | {nil}
    ;

parameterTypeList
    : parameterList
        {CRParameterTypeListNode parameterList: '1'}
    | parameterList <COMMA> <ELLIPSIS>
        {CRParameterTypeListNode  parameterList: '1' comma: '2' ellipsis: '3'}
    ;

parameterList
    : parameterDeclaration
        {CRCommaCollectionNode with: '1'}
    | parameterList <COMMA> parameterDeclaration
        {'1' add: '3' afterComma: '2'}
    ;

parameterDeclaration
    : declarationSpecifiers declarator
        {CRParameterDeclarationNode declarationSpecifiers: '1' declarator: '2'}
    | declarationSpecifiers abstractDeclarator
        {CRParameterDeclarationNode declarationSpecifiers: '1' declarator: '2'}
    | declarationSpecifiers
        {CRParameterDeclarationNode declarationSpecifiers: '1'}
    ;

identifierList
    : identifierToken
        {CRCommaCollectionNode with: '1'}
    | identifierList <COMMA> identifierToken
        {'1' add: '3' afterComma: '2'}
    ;

declarationList
    : declarationList declaration
        {'1' add: '2'}
    | declarationList controlLine
        {'1' add: '2'}
    | declaration
        {CRCollectionNode with: '1'}
    | controlLine
        {CRCollectionNode with: '1'}
    ;

declaration
    : declarationSpecifiers initDeclaratorList <SEMI_COLON>
        {|dNode| dNode := CRDeclarationNode declarationSpecifiers: '1' initDeclaratorList: '2' semiColon: '3'.
         scanner hasComments ifTrue: [dNode comments: scanner getComments].
         scanner endTypedef.
         dNode
        }
    | declarationSpecifiers <SEMI_COLON>
        {|dNode| dNode := CRDeclarationNode declarationSpecifiers: '1' semiColon: '2'.
         scanner hasComments ifTrue: [dNode comments: scanner getComments].
         scanner endTypedef.
         dNode
        }
    ;

initDeclaratorList
    : initDeclarator
        {CRCommaCollectionNode with: '1'}
    | initDeclaratorList <COMMA> initDeclarator
        {'1' add: '3' afterComma: '2'}
    ;

initDeclarator
```

```
    : declarator
        {CRInitDeclaratorNode declarator: '1'}
    | declarator <EQUALS_OP> initializer
        {CRInitDeclaratorNode declarator: '1' equalsOp: '2' initializer: '3'}
    ;

initializer
    : assignmentExpression
        {'1'}
    | <LEFT_BRACE> initializerList <RIGHT_BRACE>
        {CRInitializerListNode leftBrace: '1' list: '2' rightBrace: '3'}
#   | <LEFT_BRACE> initializerList <COMMA> <RIGHT_BRACE> #
{CRInitializerListNode leftBrace: '1' list: '2' comma: '3'
rightBrace: '4'}
    ;

initializerList
    : {CRCommaCollectionNode new}
    | initializer
        {CRCommaCollectionNode with: '1'}
    | initializer <COMMA> initializerList
        {'3' addFirst: '1' beforeComma: '2'}
    | conditionalDirective initializerList
        {'2' addFirst: '1' beforeComma: nil}
    | initializer conditionalDirective initializerList
        {'3' addFirst: '2' beforeComma: nil.
        '3' addFirst: '1' beforeComma: nil}
    ;

#***********Expressions

constantExpression
    : conditionalExpression
    ;

conditionalExpression
    : logicalOrExpression
        {'1'}
    | logicalOrExpression <QUESTION_MARK> expression <COLON> conditionalExpression
        {CRConditionalExpressionNode logicalOrExpression: '1' questionMark: '2' expression: '3'
            colon: '4' conditionalExpression: '5'}
    ;

logicalOrExpression
    : logicalAndExpression
        {'1'}
    | logicalOrExpression logicalOrOperatorToken logicalAndExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

logicalOrOperatorToken : <LOGIC_OR_OP>
        {CROperatorNode token: '1'}
    ;

logicalAndExpression
    : inclusiveOrExpression
        {'1'}
    | logicalAndExpression logicalAndOperatorToken inclusiveOrExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

logicalAndOperatorToken : <LOGIC_AND_OP>
        {CROperatorNode token: '1'}
    ;

inclusiveOrExpression
    : exclusiveOrExpression
        {'1'}
```

```
    | inclusiveOrExpression inclusiveOrOperatorToken exclusiveOrExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

inclusiveOrOperatorToken : <INCL_OR_OP>
        {CROperatorNode token: '1'}
    ;

exclusiveOrExpression
    : andExpression
        {'1'}
    | exclusiveOrExpression exclusiveOrOperatorToken andExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

exclusiveOrOperatorToken : <EXCL_OR_OP>
        {CROperatorNode token: '1'}
    ;

andExpression
    : equalityExpression
        {'1'}
    | andExpression andOperatorToken equalityExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

andOperatorToken : <AND_OP>
        {CROperatorNode token: '1'}
    ;

equalityExpression
    : relationalExpression
        {'1'}
    | equalityExpression equalityOperatorToken relationalExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

equalityOperatorToken : <EQUALITY_OP>
        {CROperatorNode token: '1'}
    ;

relationalExpression
    : shiftExpression
        {'1'}
    | relationalExpression relationalOperatorToken shiftExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

relationalOperatorToken : <REL_OP>
        {CROperatorNode token: '1'}
    ;

shiftExpression
    : additiveExpression
        {'1'}
    | shiftExpression shiftOperatorToken additiveExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

shiftOperatorToken : <SHIFT_OP>
        {CROperatorNode token: '1'}
    ;

additiveExpression
    : multiplicativeExpression
        {'1'}
    | additiveExpression additiveOperatorToken multiplicativeExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
```

```
    ;

additiveOperatorToken : <ADD_OP>
        {CROperatorNode token: '1'}
    ;

multiplicativeExpression
    : castExpression
        {'1'}
    | multiplicativeExpression multiplicativeOperatorToken castExpression
        {CRBinaryExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

multiplicativeOperatorToken
    : <STAR_OP>
        {CROperatorNode token: '1'}
    | <MULT_OP>
        {CROperatorNode token: '1'}
    ;

castExpression
    : unaryExpression
        {'1'}
    | <LEFT_PAREN> typeName <RIGHT_PAREN> castExpression
        {CRCastExpressionNode leftParen: '1' typeName: '2' rightParen: '3' castExpression: '4'}
    ;

unaryExpression
    : postfixExpression
        {'1'}
    | doubleOperator unaryExpression
        {CRPrefixAdditionNode operator: '1' unaryExpression: '2'}
    | unaryOperator castExpression
        {CRPrefixExpressionNode operator: '1' expression: '2'}
    | <SIZEOF> unaryExpression
        {CRSizeOfExpressionNode sizeOfToken: '1' expression: '2'}
    | <SIZEOF> <LEFT_PAREN> typeName <RIGHT_PAREN>
        {CRSizeOfExpressionNode sizeOfToken: '1' leftParen: '2' expression: '3' rightParen: '4'}
    ;

doubleOperator
    : <INC_OP>
        {CROperatorNode token: '1'}
    | <DEC_OP>
        {CROperatorNode token: '1'}
    ;

unaryOperator
    : <AND_OP>
        {CROperatorNode token: '1'}
    | <STAR_OP>
        {CROperatorNode token: '1'}
    | <ADD_OP>
        {CROperatorNode token: '1'}
    | <UNARY_OP>
        {CROperatorNode token: '1'}
    ;

postfixExpression
    : primaryExpression
        {'1'}
    | postfixExpression <LEFT_BLOCK> expression <RIGHT_BLOCK>
        {CRArrayExpressionNode postfixExpression: '1' leftBrace: '2' indexExpression: '3' rightBrace: '4'}
    | postfixExpression <LEFT_PAREN> <RIGHT_PAREN>
        {CRFunctionCallNode postfixExpression: '1' leftParen: '2' argumentExpressionList: nil rightParen: '3'}
    | postfixExpression <LEFT_PAREN> argumentExpressionList <RIGHT_PAREN>
        {CRFunctionCallNode postfixExpression: '1' leftParen: '2' argumentExpressionList: '3' rightParen: '4'}
    | postfixExpression <DOT> identifierToken
        {CRStructFieldExpressionNode postfixExpression: '1' pointerOp: '2' field: '3'}
```

```
    | postfixExpression <PTR_OP> identifierToken
        {CRStructFieldExpressionNode postfixExpression: '1' pointerOp: '2' field: '3'}
    | postfixExpression doubleOperator
        {CRPostfixAdditionNode postfixExpression: '1' operator: '2'}
    ;

primaryExpression
    : identifierToken
        {'1'}
    | constant
        {CRConstantExpressionNode value: '1' first}
    | stringLiteralToken
        {'1'}
    | <LEFT_PAREN> expression <RIGHT_PAREN>
        {CRParenthesizedExpressionNode leftParen: '1' expression: '2' rightParen: '3'}
    | <LEFT_PAREN> compoundStatement <RIGHT_PAREN>
        {CRStatementExpressionNode leftParen: '1' compoundStatement: '2' rightParen: '3'}
    ;

constant
    : <CHAR_CONSTANT>
    | <INT_CONSTANT>
    | <FLOAT_CONSTANT>
    ;

stringLiteralToken : <STRING_LITERAL>
        {CRStringValueNode value: '1'}
    | stringLiteralToken <STRING_LITERAL>
        {'1' addValue: '2'}
    ;

expression
    : assignmentExpression
        {CRComposedExpressionNode with: '1'}
    | expression <COMMA> assignmentExpression
        {'1' add: '3' afterComma: '2'}
    ;

assignmentExpression
    : conditionalExpression
        {'1'}
    | unaryExpression assignmentOperatorToken assignmentExpression
        {CRAssignmentExpressionNode leftOperand: '1' operator: '2' rightOperand: '3'}
    ;

assignmentOperatorToken
    : <EQUALS_OP>
        {CROperatorNode token: '1'}
    | <ASSIGN_OP>
        {CROperatorNode token: '1'}
    ;

argumentExpressionList
    : assignmentExpression
        {CRCommaCollectionNode with: '1'}
    | argumentExpressionList <COMMA> assignmentExpression
        {'1' add: '3' afterComma: '2'}
    ;

###**************************Statements

compoundStatement
    : <LEFT_BRACE> <RIGHT_BRACE>
        {|stat| stat := CRCompoundStatementNode leftBrace: '1' rightBrace: '2'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    | <LEFT_BRACE> declarationOrStatementList <RIGHT_BRACE>
```

```
        {|stat| stat := CRCompoundStatementNode leftBrace: '1' statementList: '2' rightBrace: '3'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
;

declarationOrStatementList
    : declaration
        {CRCollectionNode with: '1'}
    | statement
        {CRCollectionNode with: '1' first}
    | declarationOrStatementList declaration
        {'1' add: '2'}
    | declarationOrStatementList statement
        {'1' add: '2' first}
    ;

statement
    : labeledStatement
    | compoundStatement
    | expressionStatement
    | selectionStatement
    | iterationStatement
    | jumpStatement
    | controlLine
    | asmInstruction
    ;

labeledStatement
    : identifierToken <COLON> statement
        {|stat| stat := CRLabeledStatementNode identifier: '1' colon: '2' statement: '3' first.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    | <CASE> constantExpression <COLON> statement
        {|stat| stat := CRCaseStatementNode caseToken: '1' constantExpression: '2' first colon: '3'
                            statement: '4' first.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    | <DEFAULT> <COLON> statement
        {|stat| stat := CRDefaultStatementNode defaultToken: '1' colon: '2' statement: '3' first.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    ;

expressionStatement
    : <SEMI_COLON>
        {|stat| stat := CRExpressionStatementNode semiColon: '1'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat}
    | expression <SEMI_COLON>
        {|stat| stat := CRExpressionStatementNode expression: '1' semiColon: '2'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat}
    ;

selectionStatement
    : <IF> <LEFT_PAREN> expression <RIGHT_PAREN> statement
        {|stat| stat := CRIfStatementNode ifToken: '1' leftParen: '2' expression: '3' rightParen: '4'
                            thenStatement: '5' first.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat}
    | <IF><LEFT_PAREN> expression <RIGHT_PAREN> statement <ELSE> statement
        {|stat| stat := CRIfStatementNode ifToken: '1' leftParen: '2' expression: '3' rightParen: '4'
                            thenStatement: '5' first elseToken: '6' elseStatement: '7' first.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
```

```
        stat}
    | <SWITCH> <LEFT_PAREN> expression <RIGHT_PAREN> statement
        {|stat| stat := CRSwitchStatementNode switchToken: '1' leftParen: '2' expression: '3'
                        rightParen: '4' statement: '5' first.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    ;

iterationStatement
    : <WHILE> <LEFT_PAREN> expression <RIGHT_PAREN> statement
        {|stat| stat := CRWhileStatementNode whileToken: '1' leftParen: '2' expression: '3'
                        rightParen: '4' statement: '5' first.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <DO> statement <WHILE> <LEFT_PAREN> expression <RIGHT_PAREN> <SEMI_COLON>
        {|stat| stat := CRDoWhileStatementNode doToken: '1' statement: '2' first whileToken: '3'
                        leftParen: '4' expression: '5' rightParen: '6' semiColon: '7'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <FOR> <LEFT_PAREN> expressionStatement expressionStatement <RIGHT_PAREN> statement
        {|stat| stat := CRForStatementNode forToken: '1' leftParen: '2' expressionStat1: '3'
                        expressionStat2: '4' rightParen: '5' statement: '6' first.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <FOR> <LEFT_PAREN> expressionStatement expressionStatement expression <RIGHT_PAREN> statement
        {|stat| stat := CRForStatementNode forToken: '1' leftParen: '2' expressionStat1: '3'
                        expressionStat2: '4' expression: '5' rightParen: '6' statement: '7' first.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    ;

jumpStatement
    : <GOTO> identifierToken <SEMI_COLON>
        {|stat| stat := CRGotoStatementNode gotoToken: '1' identifier: '2' semiColon: '3'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <CONTINUE> <SEMI_COLON>
        {|stat| stat := CRContinueStatementNode continueToken: '1' semiColon: '2'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <BREAK> <SEMI_COLON>
        {|stat| stat := CRBreakStatementNode breakToken: '1' semiColon: '2'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <RETURN> <SEMI_COLON>
        {|stat| stat := CRReturnStatementNode returnToken: '1' semiColon: '2'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    | <RETURN> expression <SEMI_COLON>
        {|stat| stat := CRReturnStatementNode returnToken: '1' expression: '2' semiColon: '3'.
        scanner hasComments ifTrue: [stat comments: scanner getComments].
        stat}
    ;

##### Preprocessor

conditionalDirective
    : <CONDITIONAL_START_IF>
        {CRControlConditionalStartIfNode token: '1'}
    | <CONDITIONAL_START_IFDEF>
        {CRControlConditionalStartIfdefNode token: '1'}
    | <CONDITIONAL_ELIF>
        {CRControlConditionalElifNode token: '1'}
    | <CONDITIONAL_ELSE>
        {CRControlConditionalElseNode token: '1'}
    | <CONDITIONAL_END>
        {CRControlConditionalEndNode token: '1'}
    ;
```

```
macroDirective
    : <DEFINE>
        {CRControlDefineNode token: '1'}
    | <UNDEF>
        {CRControlUndefineNode token: '1'}
    ;


controlLine: <INCLUDE>
        {|node| node := CRControlIncludeNode token: '1'.
         scanner hasComments ifTrue: [node comments: scanner getComments].
         node}
    | macroDirective
        {|node| node := '1'.
         scanner hasComments ifTrue: [node comments: scanner getComments].
         node}
    | conditionalDirective
        {|node| node := '1'.
         scanner hasComments ifTrue: [node comments: scanner getComments].
         node}
    | <OTHER_DIRECTIVE>
        {|node| node := CRControlOtherNode token: '1'.
         scanner hasComments ifTrue: [node comments: scanner getComments].
         node}
    | <CONDITIONAL_START_IF> <DEFINED>
        {CRControlConditionalStartIfNode token: '1'}
            #It will never parse this, but we need it to generate the token "defined"
    ;


###### Assembler instructions

asmInstruction
    : <ASM> <LEFT_PAREN> asmExpression <RIGHT_PAREN> <SEMI_COLON>
        {|stat| stat := CRAsmInstructionNode asmToken: '1' leftParen: '2' asmExpression: '3'
            rightParen: '4' semiColon: '5'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    | <ASM> <VOLATILE> <LEFT_PAREN> asmExpression <RIGHT_PAREN> <SEMI_COLON>
        {|stat| stat := CRAsmInstructionNode asmToken: '1' volatileToken: '2' leftParen: '3'
            asmExpression: '4' rightParen: '5' semiColon: '6'.
         scanner hasComments ifTrue: [stat comments: scanner getComments].
         stat
        }
    ;

asmExpression
    : stringLiteralToken
        {CRAsmExpressionNode assemblerTemplate: '1'}
    | stringLiteralToken <COLON> asmOperandList
        {CRAsmExpressionNode assemblerTemplate: '1' colon1: '2' outputOperands: '3'}
    | stringLiteralToken <COLON> asmOperandList <COLON> asmOperandList
        {CRAsmExpressionNode assemblerTemplate: '1' colon1: '2' outputOperands: '3' colon2: '4'
            inputOperands: '5'}
    | stringLiteralToken <COLON> asmOperandList <COLON> asmOperandList <COLON> stringList
        {CRAsmExpressionNode assemblerTemplate: '1' colon1: '2' outputOperands: '3' colon2: '4'
            inputOperands: '5' colon3: '6' hardRegisters: '7'}
    ;

asmOperandList
    : {CRCommaCollectionNode new}
    | asmOperand
        {CRCommaCollectionNode with: '1'}
    | asmOperandList <COMMA> asmOperand
        {'1' add: '3' afterComma: '2'}
    ;
```

```
asmOperand
    : stringLiteralToken <LEFT_PAREN> expression <RIGHT_PAREN>
        {CRAsmOperandNode constraint: '1' leftParen: '2' operand: '3' rightParen: '4'}
    | <LEFT_BLOCK> identifierToken <RIGHT_BLOCK> stringLiteralToken <LEFT_PAREN> expression <RIGHT_PAREN>
        {CRAsmOperandNode leftBlock: '1' symbolicName: '2' rightBlock: '3' constraint: '4' leftParen: '5'
            operand: '6' rightParen: '7'}
    ;

stringList
    : stringLiteralToken
        {CRCommaCollectionNode with: '1'}
    | stringList <COMMA> stringLiteralToken
        {'1' add: '3' afterComma: '2'}
    ;
```

Appendix D

Source Code of Examples

This appendix first shows the original source code for rm.c and remove.h in the package "coreutil-5.2.1" [73]. The comments at the beginning of rm.c were shortened since they were not important for our purposes. Then, the last two listings are the final versions of both files after the transformations listed in Chapter 1 has been applied.

Following is the original source code of rm.c.

```
/* 'rm' file deletion utility for GNU.
   Copyright (C) 88, 90, 91, 1994-2004 Free Software Foundation, Inc.
   ...
  */

#include <config.h>
#include <stdio.h>
#include <getopt.h>
#include <sys/types.h>
#include <assert.h>

#include "system.h"
#include "dirname.h"
#include "error.h"
#include "quote.h"
#include "remove.h"
#include "root-dev-ino.h"
#include "save-cwd.h"

/* The official name of this program (e.g., no 'g' prefix).  */
#define PROGRAM_NAME "rm"

#define AUTHORS \
  "Paul Rubin", "David MacKenzie, Richard Stallman", "Jim Meyering"

/* Name this program was run with.  */
char *program_name;

/* For long options that have no equivalent short option, use a
   non-character as a pseudo short option, starting with CHAR_MAX + 1.  */
```

```
enum
{
  NO_PRESERVE_ROOT = CHAR_MAX + 1,
  PRESERVE_ROOT,
  PRESUME_INPUT_TTY_OPTION
};

static struct option const long_opts[] =
{
  {"directory", no_argument, NULL, 'd'},
  {"force", no_argument, NULL, 'f'},
  {"interactive", no_argument, NULL, 'i'},

  {"no-preserve-root", no_argument, 0, NO_PRESERVE_ROOT},
  {"preserve-root", no_argument, 0, PRESERVE_ROOT},

  /* This is solely for testing.  Do not document.  */
  /* It is relatively difficult to ensure that there is a tty on stdin.
     Since rm acts differently depending on that, without this option,
     it'd be harder to test the parts of rm that depend on that setting.  */
  {"presume-input-tty", no_argument, NULL, PRESUME_INPUT_TTY_OPTION},

  {"recursive", no_argument, NULL, 'r'},
  {"verbose", no_argument, NULL, 'v'},
  {GETOPT_HELP_OPTION_DECL},
  {GETOPT_VERSION_OPTION_DECL},
  {NULL, 0, NULL, 0}
};

void
usage (int status)
{
  if (status != EXIT_SUCCESS)
    fprintf (stderr, _("Try '%s --help' for more information.\n"),
         program_name);
  else
    {
      char *base = base_name (program_name);
      printf (_("Usage: %s [OPTION]... FILE...\n"), program_name);
      fputs (_("\
Remove (unlink) the FILE(s).\n\
\n\
  -d, --directory       unlink FILE, even if it is a non-empty directory\n\
                          (super-user only; this works only if your system\n\
                           supports 'unlink' for nonempty directories)\n\
  -f, --force           ignore nonexistent files, never prompt\n\
  -i, --interactive     prompt before any removal\n\
"), stdout);
      fputs (_("\
      --no-preserve-root do not treat '/' specially (the default)\n\
      --preserve-root   fail to operate recursively on '/'\n\
  -r, -R, --recursive   remove the contents of directories recursively\n\
  -v, --verbose         explain what is being done\n\
"), stdout);
      fputs (HELP_OPTION_DESCRIPTION, stdout);
      fputs (VERSION_OPTION_DESCRIPTION, stdout);
      printf (_("\
\n\
To remove a file whose name starts with a '-', for example '-foo',\n\
use one of these commands:\n\
  %s -- -foo\n\
\n\
  %s ./-foo\n\
"),
          base, base);
      fputs (_("\
\n\
Note that if you use rm to remove a file, it is usually possible to recover\n\
the contents of that file.  If you want more assurance that the contents are\n\
```

```
truly unrecoverable, consider using shred.\n\
"), stdout);
      printf (_("\nReport bugs to <%s>.\n"), PACKAGE_BUGREPORT);
    }
  exit (status);
}

static void
rm_option_init (struct rm_options *x)
{
  x->unlink_dirs = 0;
  x->ignore_missing_files = 0;
  x->interactive = 0;
  x->recursive = 0;
  x->root_dev_ino = NULL;
  x->stdin_tty = isatty (STDIN_FILENO);
  x->verbose = 0;
}

int
main (int argc, char **argv)
{
  bool preserve_root = false;
  struct rm_options x;
  int fail = 0;
  int c;

  initialize_main (&argc, &argv);
  program_name = argv[0];
  setlocale (LC_ALL, "");
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);

  atexit (close_stdout);

  rm_option_init (&x);

  while ((c = getopt_long (argc, argv, "dfirvR", long_opts, NULL)) != -1)
    {
      switch (c)
    {
    case 0:     /* Long option.  */
      break;

    case 'd':
      x.unlink_dirs = 1;
      break;

    case 'f':
      x.interactive = 0;
      x.ignore_missing_files = 1;
      break;

    case 'i':
      x.interactive = 1;
      x.ignore_missing_files = 0;
      break;

    case 'r':
    case 'R':
      x.recursive = 1;
      break;

    case NO_PRESERVE_ROOT:
      preserve_root = false;
      break;

    case PRESERVE_ROOT:
```

```
      preserve_root = true;
      break;

    case PRESUME_INPUT_TTY_OPTION:
      x.stdin_tty = 1;
      break;

    case 'v':
      x.verbose = 1;
      break;

    case_GETOPT_HELP_CHAR;
    case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
    default:
      usage (EXIT_FAILURE);
    }
    }

  if (argc <= optind)
    {
      if (x.ignore_missing_files)
    exit (EXIT_SUCCESS);
      else
    {
      error (0, 0, _("too few arguments"));
      usage (EXIT_FAILURE);
    }
    }

  if (x.recursive && preserve_root)
    {
      static struct dev_ino dev_ino_buf;
      x.root_dev_ino = get_root_dev_ino (&dev_ino_buf);
      if (x.root_dev_ino == NULL)
    error (EXIT_FAILURE, errno, _("failed to get attributes of %s"),
           quote ("/"));
    }

  {
    size_t n_files = argc - optind;
    char const *const *file = (char const *const *) argv + optind;

    enum RM_status status = rm (n_files, file, &x);
    assert (VALID_STATUS (status));
    if (status == RM_ERROR)
      fail = 1;
  }

  exit (fail);
}
```

Following is the original source code of remove.h.

```
#ifndef REMOVE_H
# define REMOVE_H

# include "dev-ino.h"

struct rm_options
{
  /* If nonzero, ignore nonexistent files.  */
  int ignore_missing_files;

  /* If nonzero, query the user about whether to remove each file.  */
  int interactive;
```

```
  /* If nonzero, recursively remove directories.  */
  int recursive;

  /* Pointer to the device and inode numbers of '/', when --recursive.
     Otherwise NULL.  */
  struct dev_ino *root_dev_ino;

  /* If nonzero, stdin is a tty.  */
  int stdin_tty;

  /* If nonzero, remove directories with unlink instead of rmdir, and don't
     require a directory to be empty before trying to unlink it.
     Only works for the super-user.  */
  int unlink_dirs;

  /* If nonzero, display the name of each file removed.  */
  int verbose;
};

enum RM_status
{
  /* These must be listed in order of increasing seriousness. */
  RM_OK = 2,
  RM_USER_DECLINED,
  RM_ERROR,
  RM_NONEMPTY_DIR
};

# define VALID_STATUS(S) \
  ((S) == RM_OK || (S) == RM_USER_DECLINED || (S) == RM_ERROR)

# define UPDATE_STATUS(S, New_value)                \
  do                                    \
    {                                   \
      if ((New_value) == RM_ERROR               \
      || ((New_value) == RM_USER_DECLINED && (S) == RM_OK)) \
    (S) = (New_value);                      \
    }                                   \
  while (0)

enum RM_status rm (size_t n_files, char const *const *file,
           struct rm_options const *x);

#endif
```

Following is the refactored source code of rm.c.

```
/* 'rm' file deletion utility for GNU.
   Copyright (C) 88, 90, 91, 1994-2004 Free Software Foundation, Inc.
   ...
  */

#include <config.h>
#include <stdio.h>
#include <getopt.h>
#include <sys/types.h>
#include <assert.h>

#include "system.h"
#include "dirname.h"
#include "error.h"
#include "quote.h"
#include "remove.h"
#include "root-dev-ino.h"
```

```
#include "save-cwd.h"

/* The official name of this program (e.g., no ‘g’ prefix).  */
#define PROGRAM_NAME "rm"

#define AUTHORS \
  "Paul Rubin", "David MacKenzie, Richard Stallman", "Jim Meyering"

/* Name this program was run with.  */
char *program_name;

/* For long options that have no equivalent short option, use a
   non-character as a pseudo short option, starting with CHAR_MAX + 1.  */
enum
{
  NO_PRESERVE_ROOT = CHAR_MAX + 1,
  PRESERVE_ROOT,
  PRESUME_INPUT_TTY_OPTION
};

static struct option const long_opts[] =
{
  {"directory", no_argument, NULL, 'd'},
  {"force", no_argument, NULL, 'f'},
  {"interactive", no_argument, NULL, 'i'},

  {"no-preserve-root", no_argument, 0, NO_PRESERVE_ROOT},
  {"preserve-root", no_argument, 0, PRESERVE_ROOT},

  /* This is solely for testing.  Do not document.  */
  /* It is relatively difficult to ensure that there is a tty on stdin.
     Since rm acts differently depending on that, without this option,
     it'd be harder to test the parts of rm that depend on that setting.  */
  {"presume-input-tty", no_argument, NULL, PRESUME_INPUT_TTY_OPTION},

  {"recursive", no_argument, NULL, 'r'},
  {"verbose", no_argument, NULL, 'v'},
  {GETOPT_HELP_OPTION_DECL},
  {GETOPT_VERSION_OPTION_DECL},
  {NULL, 0, NULL, 0}
};

void
usage (int status)
{
  if (status != EXIT_SUCCESS)
    fprintf (stderr, _("Try ‘%s --help’ for more information.\n"),
         program_name);
  else
    {
      char *base = base_name (program_name);
      printf (_("Usage: %s [OPTION]... FILE...\n"), program_name);
      fputs (_("\
Remove (unlink) the FILE(s).\n\
\n\
  -d, --directory       unlink FILE, even if it is a non-empty directory\n\
                          (super-user only; this works only if your system\n\
                           supports ‘unlink’ for nonempty directories)\n\
  -f, --force           ignore nonexistent files, never prompt\n\
  -i, --interactive     prompt before any removal\n\
"), stdout);
      fputs (_("\
      --no-preserve-root do not treat ‘/’ specially (the default)\n\
      --preserve-root   fail to operate recursively on ‘/’\n\
  -r, -R, --recursive   remove the contents of directories recursively\n\
  -v, --verbose         explain what is being done\n\
"), stdout);
      fputs (HELP_OPTION_DESCRIPTION, stdout);
```

```
      fputs (VERSION_OPTION_DESCRIPTION, stdout);
      printf (_("\
\n\
To remove a file whose name starts with a ‘-’, for example ‘-foo’,\n\
use one of these commands:\n\
  %s -- -foo\n\
\n\
  %s ./-foo\n\
"),
          base, base);
      fputs (_("\
\n\
Note that if you use rm to remove a file, it is usually possible to recover\n\
the contents of that file.  If you want more assurance that the contents are\n\
truly unrecoverable, consider using shred.\n\
"), stdout);
      printf (_("\nReport bugs to <%s>.\n"), PACKAGE_BUGREPORT);
    }
  exit (status);
}

static void
rm_option_init (struct rm_options *opts)
{
  opts->unlink_directories = 0;
  opts->ignore_missing_files = 0;
  opts->interactive = 0;
  opts->recursive = 0;
  opts->root_dev_ino = NULL;
  opts->stdin_tty = isatty (STDIN_FILENO);
  opts->verbose = 0;
  opts->preserve_root = false;
}

int
main (int argc, char **argv)
{
  struct rm_options opts;
  int fail = 0;
  int c;

  initialize_main (&argc, &argv);
  program_name = argv[0];
  setlocale (LC_ALL, "");
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);

  atexit (close_stdout);

  rm_option_init (&opts);

  while ((c = getopt_long (argc, argv, "dfirvR", long_opts, NULL)) != -1)
    {
      switch (c)
    {
    case 0:     /* Long option.  */
      break;

    case 'd':
      opts.unlink_directories = 1;
      break;

    case 'f':
      opts.interactive = 0;
      opts.ignore_missing_files = 1;
      break;

    case 'i':
```

```
      opts.interactive = 1;
      opts.ignore_missing_files = 0;
      break;

    case 'r':
    case 'R':
      opts.recursive = 1;
      break;

    case NO_PRESERVE_ROOT:
      opts.preserve_root = false;
      break;

    case PRESERVE_ROOT:
      opts.preserve_root = true;
      break;

    case PRESUME_INPUT_TTY_OPTION:
      opts.stdin_tty = 1;
      break;

    case 'v':
      opts.verbose = 1;
      break;

    case_GETOPT_HELP_CHAR;
    case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);
    default:
      usage (EXIT_FAILURE);
    }
    }

  if (argc <= optind)
    {
      if (opts.ignore_missing_files)
    exit (EXIT_SUCCESS);
      else
    {
      error (0, 0, _("too few arguments"));
      usage (EXIT_FAILURE);
    }
    }

  if (opts.recursive && opts.preserve_root)
    {
      static struct dev_ino dev_ino_buf;
      opts.root_dev_ino = get_root_dev_ino (&dev_ino_buf);
      if (opts.root_dev_ino == NULL)
    error (EXIT_FAILURE, errno, _("failed to get attributes of %s"),
           quote ("/"));
    }

  {
    size_t n_files = argc - optind;
    char const *const *file = (char const *const *) argv + optind;

    enum RM_status status = rm (n_files, file, &opts);
    assert (VALID_STATUS (status));
    if (status == RM_ERROR)
      fail = 1;
  }

  exit (fail);
}
```

Following is the refactored source code of “remove.h”.

```
#ifndef REMOVE_H
# define REMOVE_H

# include "dev-ino.h"

struct rm_options
{
  /* If nonzero, ignore nonexistent files.  */
  int ignore_missing_files;

  /* If nonzero, query the user about whether to remove each file.  */
  int interactive;

  /* If nonzero, recursively remove directories.  */
  int recursive;

  /* Pointer to the device and inode numbers of '/', when --recursive.
     Otherwise NULL.  */
  struct dev_ino *root_dev_ino;

  /* If nonzero, stdin is a tty.  */
  int stdin_tty;

  /* If nonzero, remove directories with unlink instead of rmdir, and don't
     require a directory to be empty before trying to unlink it.
     Only works for the super-user.  */
  int unlink_directories;

  /* If nonzero, display the name of each file removed.  */
  int verbose;
  bool preserve_root;
};

enum RM_status
{
  /* These must be listed in order of increasing seriousness. */
  RM_OK = 2,
  RM_USER_DECLINED,
  RM_ERROR,
  RM_NONEMPTY_DIR
};

# define VALID_STATUS(S) \
  ((S) == RM_OK || (S) == RM_USER_DECLINED || (S) == RM_ERROR)

# define UPDATE_STATUS(S, New_value)                \
  do                                  \
    {                                 \
      if ((New_value) == RM_ERROR               \
      || ((New_value) == RM_USER_DECLINED && (S) == RM_OK)) \
    (S) = (New_value);                    \
    }                                 \
  while (0)

enum RM_status rm (size_t n_files, char const *const *file,
           struct rm_options const *x);

#endif
```

References

[1] M. Fowler, *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

[2] W. Opdyke, *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[3] "Eclipse.org main page." http://www.eclipse.org.

[4] "Refactoring Browser." http://www.refactory.com/RefactoringBrowser/index.html.

[5] "jFactor - Home Page." http://www.instantiations.com/jfactor/, 2003.

[6] D. Roberts, *Eliminating Analysis in Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[7] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, 1997.

[8] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proc. of the Fifth International Workshop on Principles of Software Evolution (IWPSE)* (M. Aoyama, K. Inoue, and V. Rajlich, eds.), (Orlando), pp. 6–14, ACM, 2002.

[9] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *Proceedings of the IEEE Automated Software Engineering Conference (ASE)*, (Montreal, Canada), pp. 323–326, 2003.

[10] M. Lehman, "Laws of program evolution - rules and tools for programming management," in *Infotech State of the Art Conference, Why Software Projects Fail*, pp. 11/1 – 11/25, 1978.

[11] M. Lehman, "On understanding laws, evolution and conservation in the large program life cycle," *Journal of Sys. And Software*, vol. 1, no. 3, pp. 213–221, 1980.

[12] M. Lehman, "Laws of software evolution revisited," in *Proceedings of EWSPT'96*, (Nancy), 1996.

[13] R. Arnold, "Software restructuring," in *Proceedings IEEE*, vol. 77, pp. 607–617, 1989.

[14] M. Ó Cinnéide, *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, 2001.

[15] B. Foote and W. Opdyke, "Lifecycle and refactoring patterns that support evolution and reuse," in *Pattern Languages of Program Design I* (Coplien and Schmidt, eds.), pp. 239–257, Addison-Wesley, 1995.

[16] A. Garrido, "Software refactoring applied to C programming language," Master's thesis, University of Illinois at Urbana-Champaign, May 2000.

[17] W. Opdyke and R. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[18] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for Fortran and High-Performance Computing," in *2nd. Int. Workshop on Software Engineering for High Performance Computing System Applications*, (St. Louis, MO), 2005.

[19] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *Proc. of the International Conference on Software Maintenance (ICSM)*, (Budapest, Hungary), 2005.

[20] L. Tokuda and D. Batory, "Evolving object oriented designs with refactoring," in *Proc. IEEE Conference on Automated Software Engineering (ASE)*, 1999.

[21] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *Proc. of the International Conference on Software Maintenance (ICSM)*, (Budapest, Hungary), 2005.

[22] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, second ed., 1988.

[23] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," Revision of Technical Report UW-CSE-97-04-06, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, 1999.

[24] B. Stroustrup, *The Design and Evolution of C++*. Reading, Massachusetts: Addison-Wesley, 1994.

[25] R. Stallman and Z. Weinberg, "The C preprocessor." GNU Online documentation. http://gcc.gnu.org/onlinedocs/, 2001.

[26] S. Harbison and G. Steele, *C. A reference manual*. Prentice Hall, third ed., 1991.

[27] J. Steffen, "The CScope Program." Berkeley UNIX Release 3.2, 1981.

[28] D. A.Kinloch and M.Munro, "Understanding c programs using the combined c graph representation," in *Proceedings of the Int. Conference on Software Maintenance*, (Victoria, Canada), pp. 172–180, IEEE Computer Soc. Press, 1994.

[29] P. Devanbu, "Genoa - a customizable, front-end retargetable source code analysis framework," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 2, 1999.

[30] M.-A. Storey, K. Wong, and H. Mueller, "Rigi: A visualization environment for reverse engineering," in *Proceedings of the Int. Conference on Software Engineering*, 1997.

[31] R. Bowdidge, *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization*. PhD thesis, University of California, San Diego, Dep. of Computer Science and Engineering, 1995. Technical Report CS95-457.

[32] P. Devanbu, "The GEN++ page." http://seclab.cs.ucdavis.edu/ devanbu/-genp.

[33] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, 1982.

[34] D. Atkinson and W. Griswold, "Effective whole-program analysis in the presence of pointers," in *Proc. of the 6th ACM Int. Symposium on the Foundations of Software Engineering*, (Lake Buena Vista, FL), pp. 46–55, 1998.

[35] M. Mock, D. Atkinson, C. Chambers, and S. Eggers, "Improving program slicing with dynamic points-to data," in *Proceedings of the 10th Int. Symposium on the Foundations of Software Engineering*, ACM Press, 2002.

[36] R. Bowdidge and W. Griswold, "Supporting the restructuring of data abstractions through manipulation of a program visualization," *ACM Transactions of Software Enginnering and Methodology*, vol. 7, pp. 109–157, April 1998.

[37] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[38] Y. Chen, M. Nishimote, and C. Ramanoorthy, "The C information abstraction system," *IEEE Transactions on Software Engineering*, vol. 16, pp. 325– 334, March 1990.

[39] "C and C++ editor reverse engineering, code navigation and automatic documentation." http://www.scitools.com/ucpp.html, 2003.

[40] G. J. Badros and D. Notkin, "A framework for preprocessor-aware C souce code analyses," *Software Practice and Experience*, vol. 30, no. 8, 2000.

[41] D. Evans, "LCLint user's guide." MIT Laboratory for Computer Science, August 1996.

[42] S. Somé and T. Lethbridge, "Parsing minimization when extracting information from code in the presence of conditional compilation," in *Sixth International Workshop on Program Comprehension*, (Ischia, Italy), IEEE, 1998.

[43] P. Livadas and D. Small, "Understanding code containing preprocessor constructs," tech. rep., Computer and Information Science Department, Univ. of Florida, 1994.

[44] A. Cox and C. Clarke, "Relocating XML elements from preprocessed to unprocessed code," in *Int. Workshop on Program Comprehension*, 2002.

[45] B. Kullback and V. Riediger, "Folding: An approach to enable program understanding of preprocessed languages," in *Eight Working Conference on Reverse Engineering*, (Stuttagart, Germany), 2001.

[46] M. Fowler, "Crossing refactoring's rubicon." http://www.martinfowler.com/articles/refactoringRubicon.html.

[47] "IntelliJ IDEA: the most intelligent Java IDE around." http://www.intellij.com/idea/.

[48] "Xrefactory - a C/C++ development tool with refactoring browser." http://xreftech.com/xrefactory.

[49] M. Vittek, "Refactoring browser with preprocessor," in *7th European Conference on Software Maintenance and Reengineering*, (Benevento, Italy), March 2003.

[50] "DMS Software Reengineering Toolkit." http://www.semdesigns.com/Products/DMS/DMSToolkit.html.

[51] D. Roberts and J. Brant, "Refactoring tools," in *Refactoring. Improving the Design of Existing Code*, ch. 14, Addison-Wesley, 1999.

[52] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program transformations for practical scalable software evolution," in *Proceedings of the International Conference of Software Enigineering*, 2004.

[53] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Workshop on Analysis, Slicing, and Transformation at the Eighth Working Conference on Reverse Engineering (WCRE'01)*, 2001.

[54] M. D. P. Aversano, L. and I. Baxter, "Handling preprocessor-conditioned declarations," in *2nd IEEE Int. Workshop on Source Code Analysis and Manipulation, (SCAM'02)*, (Montreal), 2002.

[55] "Ref++ - C++ Refactoring Tool for Visual C++.NET." http://www.refpp.com/index.htm.

[56] "SlickEdit." http://www.slickedit.com/.

[57] R. Fanta and V. Rajlich, "Restructuring legacy c code into c++," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 1999.

[58] R. Fanta and V. Rajlich, "Reengineering object-oriented code," in *Proc. of the International Conference on Software Maintenance (ICSM)*, pp. 238–246, 1998.

[59] "The Maude System." http://maude.cs.uiuc.edu/.

[60] G. Roşu, "CS422 - Programming Language Design - Class Notes." http://fsl.cs.uiuc.edu/ grosu/classes/2004/fall/cs422/, Fall 2004.

[61] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "A Maude Tutorial." http://maude.cs.uiuc.edu/primer/maude-tutorial.pdf.

[62] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude Manual (Version 2.1.1)*. SRI International, Menlo Park, CA, April 2005.

[63] T. McCombs, "Maude 2.0 primer." http://maude.cs.uiuc.edu/primer/maude-primer.pdf.

[64] D. Friedman, M. Wand, and C. Haynes, *Essentials of Programming Languages*. McGraw-Hill, 1997.

[65] "Flex - GNU project - Free Software Foundation (FSF)." http://www.gnu.org/software/flex/flex.html.

[66] "Extensions to the C language family." http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html.

[67] A. Aho, R. Sethi, and J. Ullman, *Compilers. Principles, Techniques, and Tools*. Addison Wesley, 1988.

[68] "SmaCC by Refactory, Inc.." http://www.refactory.com/Software/SmaCC/.

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[70] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[71] C. Shanbhag, "The design of a user interface for a refactoring tool for C," Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[72] N. Martí-Oliet, J. Meseguer, and M. Palomino, "Theoroidal maps as algebraic simulations," in *Proc of WADT'04*, pp. 126–143, Springer-Verlag LNCS, 2004.

[73] "Coreutils - GNU Project - Free Software Foundation (FSF)." http://www.gnu.org/software/coreutils/coreutils.html.

Vita

Alejandra Garrido was born in La Plata, Argentina, on June 5, 1972. She attended the University of La Plata, Argentina, where she earned a Computer Analyst degree in 1993. While at the University of La Plata, Mrs. Garrido worked at LIFIA as a teaching and research assistant, while pursuing the degree of Licentiate in Computer Science, which she completed in 1997. That year she won the IMF-IDB scholarship for graduate studies and was admitted to the Department of Computer Science of the University of Illinois at Urbana-Champaign. She completed a Master of Science from the University of Illinois in 2000. That year she served as the Conference Chair of the Seventh Conference on Pattern Languages of Programs, and she became a member of the Hillside Group. During her graduate studies, Mrs. Garrido worked mostly as a teaching assistant, and she received the "Excellent Teaching Assistant" award in April 2003. She worked as an Intern in IBM T.J. Watson Research Center during Summer 2003. During her graduate studies at the University of Illinois, Mrs. Garrido has been a member of the Software Architecture Group, leaded by Prof. Ralph Johnson, where she reviewed several books and publications on object-oriented software engineering, design patterns and refactoring. Following the completion of her Ph.D., Mrs. Garrido will begin working as a Visiting Scholar with Prof. José Meseguer at the Dept. of Computer Science of the University of Illinois.