12-2011

The Design & Implementation of an Abstract Semantic Graph for Statement-Level Dynamic Analysis of C++ Applications

Edward Duffy
Clemson University, eduffy@clemson.edu

The Design & Implementation of an Abstract Semantic Graph for Statement-Level Dynamic Analysis of C++ Applications

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Edward B. Duffy
December 2011

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. James B. von Oehsen
Dr. Jason P. Hallstrom
Dr. Pradip K. Srimani

In this thesis, we describe our system, Hylian, for statement-level analysis, both static and dynamic, of a C++ application. We begin by extending the GNU *gcc* parser to generate parse trees in XML format for each of the compilation units in a C++ application. We then provide verification that the generated parse trees are structurally equivalent to the code in the original C++ application. We use the generated parse trees, together with an augmented version of the *gcc* test suite, to recover a grammar for the C++ dialect that we parse. We use the recovered grammar to generate a schema for further verification of the parse trees and evaluate the coverage provided by our C++ test suite. We then extend the parse tree, for each compilation unit, with semantic information to form an abstract semantic graph, ASG, and then link the ASGs for all of the compilation units into a unified ASG for the entire application under study. In addition, to relieve the cognitive burden of information that may inundate a developer, we describe our development of extensions to Hylian to build abbreviated abstract semantic graphs, which incorporate information about user code, but not about compiler provided library code. Finally, we describe the various approaches that we adopted to provide assurance for the developer that the ASGs that Hylian builds, correctly represent the program under study.

For Shelby and her limitless patience.

I would like to thank my friend and advisor, Dr. Brian Malloy, for his advice, encouragement, and invaluable insight in guiding my research and the writing of this dissertation and my friend and colleague, Dr. Barr von Oehsen for his support. I am truly grateful for the love, pride, and enthussiastic support from my family; particularly my parents, Karen and Edward, and my sister, Frances. And especially my wife for forgoing our weekends together so that I may complete this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of software maintenance, including comprehension, modification, and refactoring of complex object oriented systems, requires extensive and detailed information about the system under study. However, software artifacts that provide this information are frequently unavailable and for large, open-source applications, they are virtually nonexistent. Thus, much of the research in software maintenance has focused on the development of inquiry and analysis tools to automate the process of generating information to improve comprehension, and to facilitate analysis, modification and testing of the application under study.

However, the C++ language has proven to be particularly problematic for maintenance engineers interested in developing tools to facilitate analysis and modification of C++ applications. The difficulty in developing tools for C++ is mostly due to the scope and complexity of the language; for example, the grammatical representation of C++ has been shown to be larger and more complex than other, commonly used languages [73]. A particularly perplexing problem for C++ maintenance engineers entails the correct recognition of the language constructs as specified in the ISO standard, for example class template partial specializations and argument-dependent lookup [39, §A.8]. Moreover, statement-

level analysis, which is required for pointer analysis and program slicing [7, 27, 35, 87], relies on the correct recognition of expressions such as *expression*, *postfix-expression*, and *unary-expression* [39, §A.4].

Nevertheless, the C++ language is frequently used and recently has been shown to outperform other, commonly used, languages by a large margin [16]. Therefore, to support software maintenance and other software engineering efforts for the C++ language, it's important to develop analysis tools for the language.

In this thesis, we extend the GNU gcc parser to generate parse trees in XML format for each of the compilation units in a C++ application. We then provide verification that the generated parse trees are structurally equivalent to the code in the original C++ application. We use the generated parse trees together with a suite of test cases to recover a grammar for the C++ dialect that we parse. We use the recovered grammar to generate a schema for further verification of the parse trees and evaluate the coverage provided by our C++ test suite. We then extend the parse trees with semantic information and then link, or merge, the ASGs for each of the compilation units into a single ASG. Also, to relieve the cognitive burden of information that may inundate the developer, we also describe our development of extensions to Hylian to build abbreviated parse and abstract syntax trees, which incorporate information about user code, but does not include information about system code. Finally, we describe the various approaches that we adopted to provide assurance for the developer that the ASGs that Hylian builds, correctly represent the program under study.

In the next section, we provide background about languages, parsing, abstract semantic graphs (ASGs) and schemas. In Chapter 3 we describe the research that relates to parsing and construction of ASGs. In Chapter 4 we provide an overview of Hylian, our system for comprehension and analysis of C++ applications, and in Chapter 5 we describe our approach to parse tree construction and reverse engineering a grammar from a parse tree. In Chapter 6 we describe our approach to construction of ASGs and abbreviated ASGs to cap-

ture syntactic and semantic information about the program under investigation. In Chapter 7 we describe transformations on the ASG to enable dynamic analysis of the application.

Chapter 2

Background

In this section we provide background about our work by defining terminology associated with grammars, languages, scanning and parsing. A general description of languages, context-free grammars and parsing can be found in reference [2]. Since the main focus of our work is Abstract Semantic Graphs (ASGs), We review background about abstract syntax trees (ASTs) and their promotion to abstract semantic graphs (ASGs). We also review the terminology for grammar recovery and the use of schemas for validation of files in the extended markup language format (XML) and some of the schema languages that are used for validation of schema-based XML.

2.1 Terminology

Given a set of words (known as a lexicon), a *language* is a set of valid sequences formed from these words. A *grammar* is, basically, a set of rules, or production rules, that are used to define a language and any language can be defined by a set of different grammars. When describing formal languages such as programming languages, we typically use a grammar to describe the *syntax* of that language; other aspects, such as the semantics

of the language, usually cannot be described by context-free grammars.

A grammar defines a language by specifying valid sequences of derivation steps that produce sequences of terminals, known as the *sentences* of the language. One procedure for using a grammar to derive a sentence in its language is as follows. We begin with the start symbol S and apply the production rules, interpreted as left-right rewriting rules, in some sequence until only non-terminals remain. This process defines a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree correspond precisely to those symbols on the right-hand-side of a production rule. This tree is known as a *parse tree*; the process by which it is produced is known as *parsing*.

2.2 Abstract Syntax Trees and Abstract Semantic Graphs

An *abstract syntax tree* (AST) is a pruned, refined parse tree, with some non-terminals, keywords, and punctuation removed. Using the semantic rules for the input language, a semantic analyzer transforms an AST to an *abstract syntax graph* (ASG). An ASG is often the output of a compiler front end, and includes semantic information such as edges from uses of variables to their corresonding declarations, edges from type information to their corresponding definitions, and for C++, template instantiations, specializations and partial specializations.

```
1   class Node
2   {
3       int value;
4       Node *next;
5   };
```

Figure 2.1: Source code for class Node. *Definition of the C++ class Node. Node consists of an int, value, and a pointer to another Node, next.*

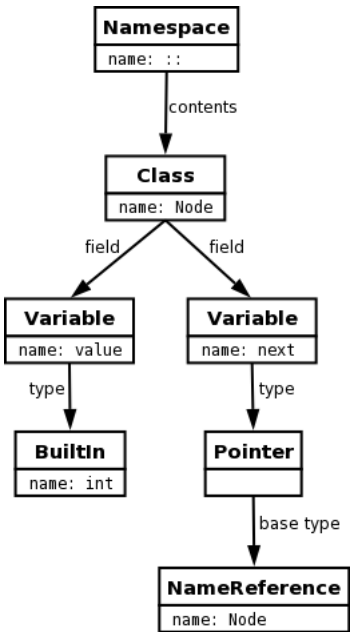


Figure 2.2: Sample AST for class Node. *An AST for class Node, which is defined in Source Listing 2.1. Uses of the types int and Node have not yet been resolved to their definitions.*

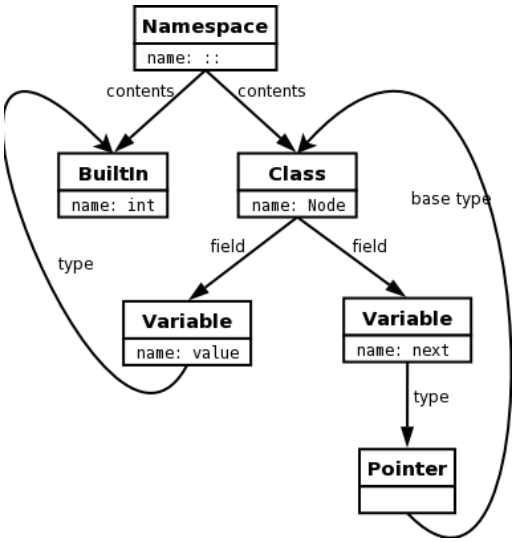


Figure 2.3: Sample ASG for class Node. *An ASG for class Node, which is defined in Source Listing 2.1. Uses of the types int and Node have been resolved to their definitions.*

In Source Listing 2.1, we list C++ code for the definition of class Node. Node consists of an int, `value`, on line 3, and a pointer to another Node, `next`, on line 4. In Figure 2.2, we illustrate a possible AST for class Node, and in Figure 2.3, we illustrate a possible ASG for class Node. Note that in Figure 2.2 the uses of the types `int` and Node have not yet been resolved to their definitions, but that in Figure 2.3 they have been resolved.

2.3 Grammar Recovery

Grammar recovery is a reverse engineering endeavor that attempts to recover a grammar representation of a language from an available source, such as, in the best case, a compiler, or in the worst case from a language reference or documentation [50]. A recovered grammar G for a language L is likely to be an approximation of G, since G is likely to be incomplete or incorrect with respect to L, depending on the source used to recover the grammar and on the available information about the language. Grammar recovery is more involved than other forms of grammar reengineering [41, 77].

2.4 Schemas

The term schema can assume several different meanings but in computer science a *schema* is a model. An XML schema is a specific, well-documented definition of the structure, content and, to some extent, the semantics of an XML document. An XML schema provides a view of the document type at a relatively high level of abstraction. Relax NG is an expressive XML schema language written specifically for grammar specification [40].

Chapter 3

Related Work

In this chapter, we review the work that relates to the construction of an abstract semantic graph (ASG) to enable dynamic analysis of C++ programs. We begin by describing some work about parsers and parser front-ends that enable or support ASG construction; Since verification of the generated parse tree and ASG is an important part of our research, in Chapter 3.2 we review the work that relates to testing grammars and parse trees. Finally, since part of our verification process includes the recovery of a grammar for the dialect of C++ that we study, in Chapter 3.3 we describe work that relates to grammar recovery.

3.1 Parse Tree and ASG Construction

A reverse engineering tool that accepts C++ source code must utilize a parser, and likely, a corresponding front end. The difficulties that arise during the construction of a parser for C++ are well documented, and are largely due to the complexity of the template sublanguage [9, 42, 56, 71, 75, 76, 86]. Consequently, the selection of robust reverse engineering tools that accept C++ programs is inadequate.

The available reverse engineering tools for C++ can be divided into two categories:

(1) those that provide their own parser (and possibly front end), and (2) those that utilize the C++ parser and front end from either the GNU Compiler Collection (*gcc*) [24], or the Edison Design Group (EDG) [19]. We provide an overview of the first category in Subsection 3.1.1, and an overview of the second category in Subsection 3.1.2. Since we are only interested in open-source software, our focus in Subsection 3.1.2 is on the related work that uses *gcc*.

3.1.1 Tools that Build a C++ Parser

Elkhound is a parser generator, similar to Bison, whose generated parsers use the Generalized LR (GLR) parsing algorithm [66, 67, 82, 83, 84]. Elkhound is used to write a parser, Elsa, which attempts to accommodate several dialects of C++ including *gcc* and Visual C++ [66]. Elsa can accommodate most of the C++ grammar. However, Elsa does not fully accommodate either dialect. For example, 89% of the test cases in the *gcc* test suite, described in Chapter 5, were parsed by Elsa. However, for the Fluxbox test case only 32 of the 74 translation units were parsed by Elsa and for FiSim only 2 of the 23 translation units were parsed by Elsa. Also, the Elsa project has not been updated for over two years [66]. Thus, we have chosen the GNU *gcc* front-end to generate parse trees for a dialect of the C++ language, a more commonly used C++ dialect than Elsa.

SourceNavigator™from Red Hat is an analysis and graphical browsing framework for C, C++, Java, Tcl, FORTRAN, and COBOL [78]. The provided fuzzy parser extracts enough high level information to provide class hierarchies, imprecise call graphs, and include graphs. SourceNavigator does not provide statement level information, and the plain text output does not conform to a schema.

Ferenc et al. [22] present Columbus, a fully integrated reverse engineering framework supporting fact extraction, linking, and analysis for C and C++ programs. Columbus

provides output in a variety of formats, including CPPML (OCaml-like pattern-matching extensions for C++), GXL (a schema for storing graphs in XML), RSF (a reverse-engineering tool that supports source code transformatios for C++ applications), and XMI (an Object Manager Group metadata standard for XML). Nevertheless, Columbus is unable to fully accept templates, as noted by [34]

The *Keystone* parser and front-end attempts to address the problems associated with the early phases of compiler development for object-oriented languages: lexical analysis, parsing and construction of a parser front-end for the C++ language [71, 72, 57, 58]. *Keystone* uses *token decorated parsing*, a technique for parsing ambiguous language constructs that exploits semantic information previously gathered in the parse. The technique permits parsing of ambiguous C++ constructs in the grammar provided in the ISO C++ standard [39], without modifying, refactoring or extending the grammar. The *Keystone* parser includes a front-end consisting of an abstract semantic graph that permits users of the system to gather detailed information at the statement level. The front-end also includes an application programmers interface, API, that permits users to extract information about the names in a program without any knowledge of the underlying system. The *Keystone* parser is shown to be conformant to the ISO C++ standard and is, in fact, more conformant than many of the popular C++ compilers in common use today [39, 59]; however, the *Keystone* parser is unable to parse open-source programs without the use of declaration-only (stub) header files for modules in the C++ standard library.

Lapierre et al. [54] present Datrix, an analyzer that extracts information from C, C++, or Java programs. Datrix extracts information for each translation unit in accordance with the Datrix ASG Model [6], and output is expressed in either TA (Tuple-Attribute Language) or VCG format. The Datrix project at Bell Canada ended in the year 2000, and the Datrix analyzer is no longer available.

3.1.2 Tools that Utilize the *gcc* C++ Parser or ASG

Industrial strength C++ parser front ends are provided by the GNU Compiler Collection [24] and the Edison Design Group [19]. They both accept virtually all of the constructs defined by the ISO C++ standard, including templates [39, 59]. However, *gcc* is in the public domain, which allows the reverse engineering tools that use it to be freely distributable; we summarize only tools that use *gcc* in this subsection.

There are two common approaches to using *gcc*. The first approach is to use the `tu` files generated in versions *gcc* 3.4.x. The second approach is to modify the source code of the parser, which creates a custom version of *gcc*.

The g^{4}re system is designed as a tool chain, consisting of applications and libraries that can be used either individually or as a single unit [45, 70, 47, 48, 44]. The implementation of g^{4}re uses a GXL-based pipe-filter architecture where each constituent application or library in the chain takes, as input, the output of the preceding application or library in the chain. An important benefit of this architecture is that g^{4}re consists of a set of loosely coupled, reusable modules: the *ASG module*, the *schema and serialization modules*, the *transformation module*, the *linking module*.

The g^{4}re system uses the GENERIC output of *gcc* to build an *ASG module*. The *ASG module* is used to build an Application Programmers Interface (API) to facilitate easy access to information about declarations, including classes, functions, and variables, as well as information about scopes, types, and control statements. The advantage of the g^{4}re tool chain, as with all of the parsers that are based on the *gcc* compiler collection, is that it can analyze any program that can be compiled by the *gcc* C++ compiler.

However, there are drawbacks to the g^{4}re system. First, the GENERIC output of *gcc*, and therefore the generated *ASG module*, is not code-complete so that the original source code cannot be regenerated from the *ASG module*. Thus, the g^{4}re system is lim-

ited to static analysis of type information for an application under study. Second, since a version of the original program cannot be generated, structural verification of the *ASG module* cannot be guaranteed. Nevertheless, the API schema for the g^{4}re system provides a language neutral specification for statement level representation of constructs in an ASG. However, the g^{4}re system cannot perform statement level analysis of an application since the GENERIC output does not include this information. Therefore, use of the g^{4}re system precludes operations such as data flow analysis, call graph construction and slicing.

LLVM (Low Level Virtual Machine) is a compiler framework designed to support program analysis and transformation by providing high-level information for compiler transformations at compile-time, link-time and run-time [55]. The LLVM framework allows code written in an *gcc* supported language to be represented in a uniform representation to permit optimizations throughout the execution lifetime of the code. The LLVM system is enable by patching *gcc* version 3.4.0 and version 4.0. The LLVM approach is complementary to our work. In our proposed system graphical representations, a parse tree and an ASG, are constructed for the source code of an application. These graphical representations can then be analyzed or transformed and then code regenerated. LLVM does not permit source code analysis or transformation, and does not permit code regeneration, but rather permits transformations and analysis of a low-level, typed, SSA-based instruction set. The LLVM approach could be applied to applications that have been analyzed or transformed by our proposed system.

Dean et al. [14] present *CPPX*, a tool that uses *gcc* for parsing and semantic analysis. *CPPX* predates the incorporation of `tu` files into *gcc*, and is built directly into the source code. *CPPX* constructs an ASG that is compliant to the Datrix ASG Schema [6], and can be serialized to GXL, TA, or VCG format. The Datrix ASG Schema is meant to accommodate several languages; this generality makes it difficult to accurately represent many C++ language constructs, such as template specializations. The last release of *CPPX*,

based on version 3.0 of *gcc*, does not properly handle the C++ Standard Library.

Hennessy et al. [36] present *gccXfront*, a tool that harnesses the *gcc* parser to tag C and C++ source code. The tool annotates source code with syntactic tags in XML by modifying the *bison* parser generator tool, as described by Malloy et al. [63]. This approach is no longer viable, because the C++ parser in *gcc* has migrated to recursive descent technology.

GCC.XML uses `tu` files to generate an XML representation for class, function, and namespace declarations, but does not propagate information such as function and method bodies [1]. As a result, many common program representations, such as the call graph or the ORD, cannot be constructed using the output of *GCC.XML*.

Antoniol et al. [4] present *XOGASTAN*, a collection of tools that convert a `tu` file to a GXL instance graph, and construct an in-memory representation of the GXL instance graph. *XOGASTAN* fails to create GXL output for certain GENERIC node types, including `try_catch_expr` and `using_directive`. Additionally, *XOGASTAN* has limited analysis capabilities for C++.

Gschwind et al. [34] present *TUAnalyzer*, a system that uses `tu` files to perform analysis of template instantiations of classes and functions. *TUAnalyzer* performs virtual method resolution by using the '`base`' and '`binf`' attributes, along with the output provided by the compiler switch `-fdump-class-hierarchy`, which reconstructs the virtual method table. The scope of *TUAnalyzer* is limited to analysis of templates; Also, *TUAnalyzer* does not produce an output representation of the `tu` file for exchange with other reverse engineering tools.

3.1.3 A Tool that Utilizes Compiler Debugging Information

The tool, *libthorin*, permits a developer to reverse engineer a program to capture design and implementation artifacts, including design metrics or a UML class or sequence

diagram [17]. However, unlike the other tools described in the chapter, *libthorin* can be applied to a variety of procedural programming languages including C, C++, Java, Fortran 90 and C#. The language independent feature of *libthorin* derives from the fact that it reads an executable program embeded with DWARF debugging information and constructs an XMI Document Object Model (DOM) representation of the code. The XMI[1] DOM includes type information about the variables and declarations in the source code. However, the DOM tree representation of the program is not source complete so that a version of the original program cannot be constructed; thus, *libthorin* cannot provide dynamic analysis of the source code.

3.2 Testing and Verification of Compiler Artifacts

There has been little research in verification, validation or testing of compiler artifacts. The only important research on this topic is a seminal paper addressing the issue of automatic generation of test cases to test grammars and grammar-based tools is the work of Purdom for generating sentences from a context-free grammar [74]. The goal of Purdom's algorithm is to use each production in the grammar at least once and to rapidly generate a minimal set of sentences that are short. However, the expression of the Purdom algorithm makes comprehension difficult and explication of the algorithm cannot be based on consideration of the algorithm alone. For example, there are places in the algorithm where it is difficult to determine what is to be done after completion of a given step. Thus, an interpretation of the algorithm is described in reference [62].

Several researchers have either based their test case generation on Purdom's work, or an extension of the work including references [5, 12, 38] and [69]. However, the problem

[1]XMI is a specification drafted by the Object Modeling Group (OMG) for the transportation of metadata among modeling tools [81]

with Purdom's algorithm is that the automatically generated test cases, while they test each production of the grammar, are not valid programs since grammars are virtually always a superset of the actual language that they specify. Thus, the test cases generated by Purdom's algorithm cannot be compiled and are therefore not very useful [62].

3.3 Research on Grammar Recovery

Lämmel and Verhoef present a semi-automatic approach to grammar recovery [51]. Their approach requires a language manual and a test suite. They use the manual to construct syntax diagrams for the language, they correct the diagrams, write transformations to correct connectivity errors, and then use the test cases to further correct the generated grammar. One advantage of their approach is that their grammar recovery is not connected to a specific parser implementation. The disadvantage of their approach is that many phases of the grammar recovery are manual. The approach that we propose uses a parse tree and a test suite and our grammar is recovered automatically.

Bouwers et al. present a methodology for recovering precedence rules from grammars [11]. They describe an algorithm for YACC and implement the method using tools for YACC and SDF. The methodology for precedence recovery could be incorporated into our grammar recovery system and used to transform our recovered grammar for GNU *gcc* into a grammar that more closely resembles the grammar specified in the ISO C++ standard [39].

Chapter 4

Overview of the System for Dynamic Program Analysis

In this Chapter, we describe Hylian, the system that we developed to empower a researcher or developer to perform statement-level analysis of a program written in the gcc dialect of the C++ language. Figure 4.1 is an overview of Hylian, which consists of three phases: (1) Parse tree extraction and grammar recovery, (2) development and generation of an abstract semantic graph (ASG), and (3) transformation of the ASG.

In Chapter 4.1, we describe the first four modules in the development of the first phase of the Hylian system: the parse tree generation and post processing modules that utilize the GNU compiler suite to parse the input program, generate parse trees and recover the gcc C++ grammar. This first phase is summarized in Figure 4.1 by the rectangles labeled I, II, III and IV. In Chapter 4.2, we describe the second phase: construction of ASGs from the parse trees, their storage and representation using the Graph Exchange Language (GXL), linkage of the ASGs for each compilation unit into a single ASG, code generation from the ASG to facilitate verification of the ASG, and an interactive GXL viewer to assist in comprehension of the ASG and the corresponding program. This second phase is

summarized in Figure 4.1 by the rectangles labeled V and VI. Finally, in Chapter 4.3, we describe the last phase: the ASG transformation system that permits modification of the source program, including insertion of probes into the original program and generation of the source application for dynamic statement-level analysis of the system under study. This final phase is summarized by rectangle VII.

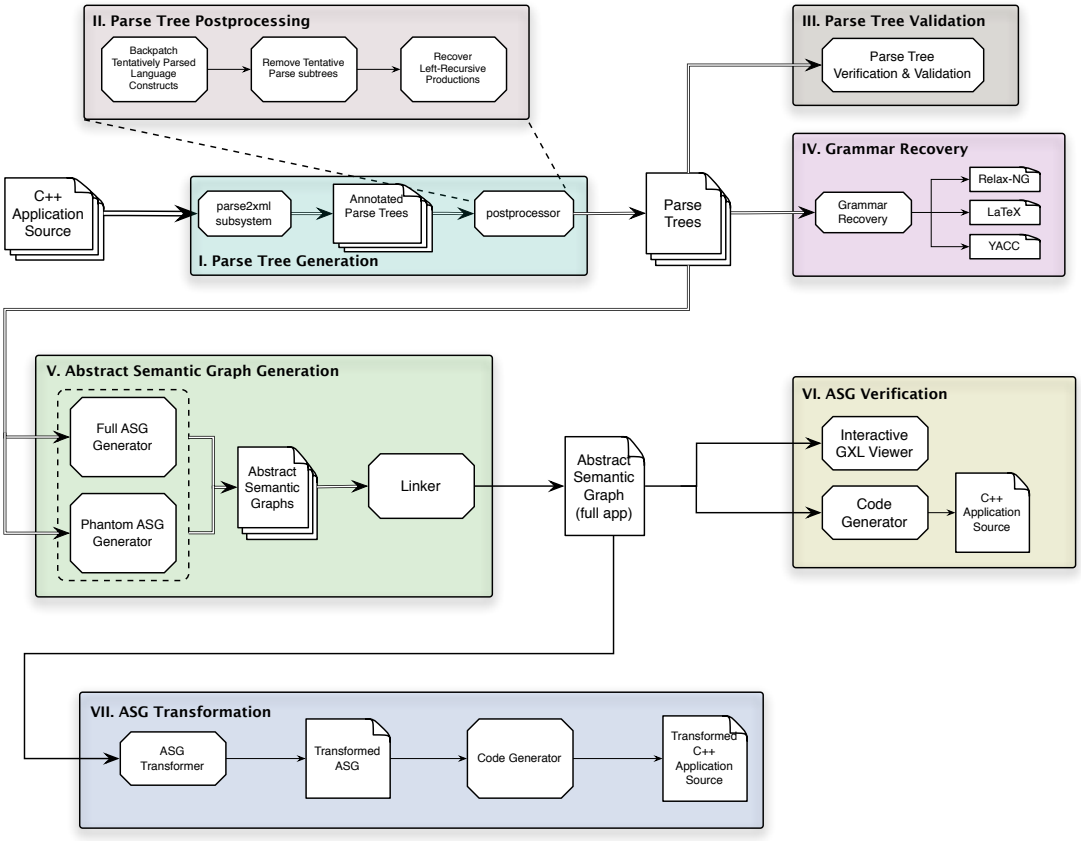


Figure 4.1: Overview

4.1 Generation of Parse Trees and Grammar Recovery

The first phase in construction of an Abstract Semantic Graph for a C++ application is to generate parse trees for each compilation unit in the application. To do this, we

exploit the gcc C++ compiler by augmenting the gcc parser with probes to generate an XML representation for each of the parse trees. To validate the generated parse trees, we recover a grammar for the gcc C++ grammar, and then use the grammar to generate a schema, in Relax NG format. The final step in this first phase is to validate the parse trees against the generated schema.

4.2 Generation of an Abstract Semantic Graph

To generate an abstract semantic graph, ASG, we first prune the generated parse trees by eliminated unnecessary non-terminals and empty productions. We then annotate the pruned parse tree with semantic information, such as the type or scope of a variable. In the case of templates, we must build an ASG representation of the instantiated template. After we have extended each of the parse trees with semantic information to produce an ASG for each compilation unit, we then *link*, or merge, the ASGs into a single ASG for the entire program.

4.3 Transformation of an Abstract Semantic Graph

To illustrate transformations, and subsequent code generation, for a Hylian ASG, we examine a previous work where a tool was developed that generated regular expressions that captured the interactions between users of the class. We show that the use of parse trees did not provide sufficient information to fully automate the process of generating interface protocols for the classes in a library and that using the Hylian ASG, the process can be fully automated and, in fact, there are even more benefits of using a Hylian ASG.

Chapter 5

Parse Tree Generation, Verification and Validation

5.1 Parse Tree Generation

For the parse tree generation phase, we use an augmented version of the GNU *gcc* parser version 4.0.0, labeled parse2xml, summarized in rectangle I of Figure 4.1 of Chapter 4. The parse2xml component generates an annotated parse tree, Annotated Parse Trees, for the input C++ test case or application by inserting probes into the file parser.c to generate a trace of grammar terminals and non-terminals. The parser.c file contains the core of the backtracking recursive-descent parser included in the source of the GNU *gcc* C++ compiler; we use version 4.0.0 in our study but our technique can be applied to any *gcc* C++ parser or to any language whose corresponding parser generates a parse tree.

The postprocessor component, shown in rectangle II of Figure 4.1, processes the Annotated Parse Trees by completing three tasks: (1) Backpatch Tentatively Parsed Language Constructs, (2) Remove Tentative Parse Subtrees, and (3) Recover Left Recursive Productions, generated iteratively. The *gcc* parser emits productions for member function bodies and default parameter lists after the associated class is parsed. Thus, the

first task entails backpatching member function bodies and default parameter lists. Also, the *gcc* parser performs tentative parsing and then backtracks to recover from incorrectly chosen alternatives. Thus, the second task entails deleting parse subtrees that were emitted as part of an incorrect alternative and writing committed subtrees to a file in XML format. Finally, the *gcc* parser is a recursive-descent parser and productions that are expressed left recursively in the ISO C++ standard must be parsed iteratively by *gcc*. Thus, the third task that the postprocessor component performs is to recover left recursive productions from productions that were generated iteratively by the recursive-descent *gcc* parser.

The output of the postprocessor component is a set of parse trees, one tree for each compilation unit of the C++ application; this is illustrated in Figure 4.1 by the document stack icon labeled Parse Trees. These parse trees are then used in the parse tree validation, grammar recovery, and ASG generation subsystems, illustrated as rectangles labeled III, IV and V.

5.2 Parse Tree Verification

Our grammar recovery technique requires a parse tree that correctly represents the particular language or language dialect under recovery. Thus, an important aspect of our work entails verification that the structure of the generated parse tree correctly reflects the structure of the particular language or language dialect under consideration. The first phase of our verification is a Partial Structural Verification of the Parse Tree, a process that entails a traversal of the generated parse tree to regenerate the code for the input test case or application. Comparison of the regenerated code with the original code for the test case or application will verify that the tokens appear in the parse tree in the correct order and that there are no extra tokens, nor have any tokens been omitted.

The partial structural verification phase of our system is illustrated in Figure 5.1,

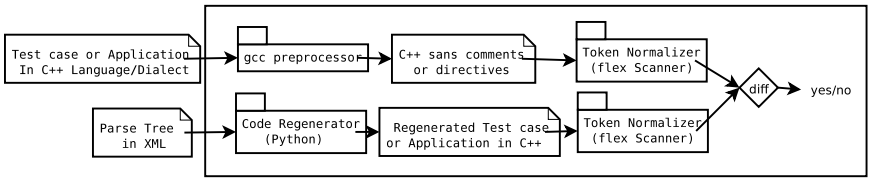


Figure 5.1: Partial Structural Verification of Parse Tree. *This figure summarizes the initial steps for verifying that our generated parse tree represents the language dialect under recovery. In this initial step we traverse the generated parse tree to regenerate the source code for the test case or application to verify that the tokens in the tree are complete and that their order of appearance in the tree matches the order of appearance in the test case or application.*

---

where the input C++ test case or application is processed by the *gcc* preprocessor, shown as gcc preprocessor in the upper left corner of the figure. The preprocessor performs macro substitution, inserts include files, and replaces trigraphs. However, the source code for the C++ test case or application cannot be compared directly to the source code that is regenerated from the parse tree since white space, comments and compiler directives are not included in their parse tree representation. Thus, the preprocessed application, must first be processed by the Token Normalizer, shown in the upper right corner of Figure 5.1. One function of the Token Normalizer is to "normalize" white space, so that the statement **int** f(); is converted into a normalized form, with one token emitted on each line of the output.

The last row in Figure 5.1 illustrates the components involved in code regeneration using the Parse Tree in XML, which is processed by the Code Regenerator, shown in the bottom left side of the figure. The Code Regenerator is a SAX parser, which traverses the Parse Tree in XML and regenerates the C++ source code by emitting the tokens in the tree; the regenerated source is then normalized using the Token Normalizer described above. Finally, we use the *diff* utility to compare the preprocessed normalized C++ application, with the source code produced by normalizing the emitted tokens during a traversal of the

```
1   Improper nesting:       Proper nesting:
2   <unary_operator>        <unary_operator>
3   </unary_operator>           <token />
4   <token />               </unary_operator>
```

Figure 5.2: A Structurally Incorrect Parse Tree. *The XML sequence on the left side of the figure illustrates an improper nesting and the sequence on the right illustrates a proper nesting.*

Parse Tree in XML.

5.2.1 Further Parse Tree Verification is Required

The partial structural verification of the parse tree provided by source code regeneration is necessary because the structure of the parse tree may be incorrect, causing tokens to be omitted, regenerated in the wrong order, or regenerated incorrectly. However, source code regeneration does not guarantee that the parse tree is correct. In particular, after completion of our source code regeneration validation, our subsequent Relax NG schema-based verification uncovered errors in the parse tree. For example, Figure 5.2 illustrates improper nesting of tokens within their corresponding production symbols: the XML sequence on the left side of the figure illustrates an improper nesting and the sequence on the right illustrates a proper nesting. An improper nesting of terminal and non-terminal grammar symbols is not exposed by code regeneration since, in regenerating the code, only the terminal symbols are considered.

An important result of our work is that generation of a parse tree in XML format, the correct regeneration of the source code, and the determination that the XML tags are properly nested will not expose a faulty parse tree such as the one represented by the XML sequence illustrated on the left side of Figure 5.2. For example, the $xmllint$ tool, without a corresponding schema, will validate the XML sequence on the left side of Figure

5.2. Our grammar recovery and Relax NG schema generation techniques, provide further structural verification of the parse tree, but does not provide semantic verification that the structures represented in the test case or application are correctly represented in the parse tree; semantic verification of the parse trees is described in Chapter 6.

5.3 Grammar Recovery from Parse Trees

In this section we present a technique for automated recovery of a grammar from parse tree instances, we describe a metrics-guided approach to semi-automated identification of candidate nonterminals to be refactored by replacing iteration with left recursion, and we present an algorithm to perform this refactoring.

By definition, a parse tree instance captures the derivation of a sentence from the language; thus, a parse tree instance encodes an instance of the grammar for the language. Indeed, we can recover a *grammar instance* (a partial grammar for a language) from a parse tree instance. By taking the union of two grammar instances, that is, the union of the productions in the two grammar instances, we can recover a grammar that captures the productions encoded in each of the parse tree instances. It follows that by taking the union of all grammar instances recovered for a test suite we can forge a grammar that generates the sentences in the test suite. However, depending on the parsing technology used by the parser that generates the parse tree instances, we might recover an *iterative* grammar rather than a *recursive* grammar. More specifically, we might recover a grammar that generates a *subset* of the language rather than one that generates a *superset* of the language.

Grammatical difficulties, such as the "Names Too Specific" problem exhibited by grammars for Java, C++, and C$^{\#}$ when used in LALR(1) parsers [32], can require that a grammar be altered from its standard form. As another example, a grammar that uses left recursion must be rewritten to use right recursion and/or iteration to be used with a recur-

base_specifier_list
→ *base_specifier*
→ *base_specifier_list* , *base_specifier*

Figure 5.3: Rewriting Left Recursion as Right Recursion. *A left recursive nonterminal that can be easily rewritten to use right recursion.*

postfix_expression
→ *primary_expression* (1)
→ *postfix_expression* [*expression*] (2)
→ *postfix_expression* ($expression\_list_{opt}$) (3)
→ *simple_type_specifier* ($expression\_list_{opt}$) (4)
→ typename $::_{opt}$ *nested_name_specifier identifier* ($expression\_list_{opt}$) (5)
→ typename $::_{opt}$ *nested_name_specifier* $template_{opt}$ *template_id* ($expression\_list_{opt}$) (6)
→ *postfix_expression* . $template_{opt}$ *id_expression* (7)
→ *postfix_expression* -> $template_{opt}$ *id_expression* (8)
→ *postfix_expression* . *pseudo_destructor_name* (9)
→ *postfix_expression* -> *pseudo_destructor_name* (10)
→ *postfix_expression* ++ (11)
→ *postfix_expression* -- (12)
→ dynamic_cast < *type_id* > (*expression*) (13)
→ static_cast < *type_id* > (*expression*) (14)
→ reinterpret_cast < *type_id* > (*expression*) (15)
→ const_cast < *type_id* > (*expression*) (16)
→ typeid (*expression*) (17)
→ typeid (*type_id*) (18)

Figure 5.4: Difficult to Rewrite with Right Recursion. *A left recursive nonterminal that cannot be easily rewritten to use right recursion.*

sive descent parser, such as gcc 4.0. For example, the productions shown in Figure 5.3 use left recursion to generate a simple list and may be easily rewritten with right recursion. Other productions are not as easy to rewrite; for example, consider the productions shown in Figure 5.4. These productions are taken from the grammar in the ISO C++98 standard, where they are expressed using left recursion but are not easily rewritten with right recursion. In this case iteration can be used to obviate the need for rewriting the grammar with right recursion.

An iterative grammar specifies a finite grammar; that is, an iterative grammar ex-

plicitly expresses all possible production right hand sides. However, such a grammar is unnecessarily restrictive; iteration must be bounded and the bound is often an implementation-defined value. Because bounds may differ, an iterative grammar may lead to the introduction of several language dialects. In addition, unless an exhaustive test suite is used to recover a grammar as described above, the recovered grammar will generate only a subset of the intended language. This is in stark contrast to the model typically used when developing language-based software, where the grammar generates a superset of intended language and semantic rules are used to eliminate invalid sentences. Furthermore, an iterative grammar specification is likely an artifact of a parser, and a parser can force artificial restrictions on the form of a grammar. The introduction of left recursion in place of iteration is an example of a transformation that would make an iterative grammar more useful to another application [52], as well as more useful to a human. To actually replace iteration with left recursion is a string rewriting task; however, a key remaining issue is the identification of candidate nonterminals (sets of productions) to which the rewriting is to be applied.

Power and Malloy [73] describe six size metrics for grammars; three of these metrics are *Number of Productions* (PROD), *Average RHS Size* (AVS) and *Halstead Effort* (HAL), all three of which can be computed at the grammar or nonterminal granularity. A large value of PROD for a nonterminal might indicate the use of iteration; however, it might indicate a more general need for refactoring. When computed over the set of productions for a nonterminal, AVS is the average number of symbols on the right-hand side of a production for that nonterminal. A large value for AVS can result from the use of iteration in a grammar. For example, if a grammar includes a nonterminal with a set of productions that generates a parenthesized list of parameters, and if that set of productions is expressed using iteration rather than left recursion, a production that expresses a list of each possible length must be included, which results in a large value for AVS. HAL is a relativization of

McCabe cyclomatic complexity that weights the counts of unique symbols by the numbers of occurrences of each symbol. The divisor in the equation expressing HAL is the number of unique operands times a constant; thus, minimizing this value will maximize the value of HAL. Similarly to AVS, a large value for HAL can result from the use of iteration in a grammar. Continuing the previous example, the repetition of symbols in the lists of each possible length keeps the number of unique operands low, which increases the value of HAL.

In Chapter 5.3.1 we present a technique for automated recovery of a grammar from parse tree instances. In Chapter 5.3.2 we describe a metrics-guided approach to semi-automated identification of candidate nonterminals to be refactored by replacing iteration with left recursion, and we present an algorithm to perform this refactoring in Chapter 5.3.3.

5.3.1 Automated Recovery of a Grammar

Our methodology for automated grammar recovery minimally requires as input a single parse tree instance but can accept multiple parse tree instances with no modification. For simplicity, in this section we describe the recovery of a grammar instance from a single parse tree instance. Figures 5.5 and 5.6 illustrate our approach to recovering a grammar instance from a parse tree instance. In Figure 5.5 we illustrate three example artifacts for the expression $a + 1$, and in Figure 5.6 we illustrate our grammar recovery algorithm.

Figure 5.5a shows a graphical representation of an example parse tree instance for a C++-like language generated for $a + 1$. By definition, the parse tree is hierarchical; its interior nodes are labeled by nonterminals from the grammar and its leaf nodes are labeled by terminals from the grammar. Given hierarchical data and an accompanying schema one can express that data as an XML document. Thus, with the grammar serving as the schema, we can encode the example parse tree instance from Figure 5.5a as the XML

illustrated in Figure 5.5b. The XML-encoded parse tree instance in Figure 5.5b contains two categories of tags. The first category contains only one tag, `token`, which is used to represent terminals. The `token` tag uses the `type` attribute to differentiate terminal types, and where appropriate, the `value` or `expr_value` attribute to hold an instance value. The second category of tags contains all tags other than `token` and is used to represent nonterminals. The names of the tags in this category differentiate the nonterminals and none of these tags have attributes.

Figure 5.6 illustrates our algorithm for recovering a grammar instance from a parse tree instance; because we encode parse tree instances as XML documents, we express the recovery algorithm as a content handler for a SAX parser (using Python syntax). Line 1 of the figure lists the declaration for the global set that holds the recovered productions. Lines 2–20 list the definition of the Handler class, the SAX content handler. The Handler class contains two instance variables. The first instance variable is production_stack, which holds partially and/or totally complete productions, and the second is token, which holds the type of the current token (e.g., NUMBER or ID). The declaration for the startElement method is on line 7 of the figure. This method is a callback and is called by the SAX parser whenever a new XML tag is opened. When startElement is called, we push the current tag onto production_stack and, only if the current tag is 'token', we set the instance variable token. On line 12 of the figure the method endElement is declared. This method is again a callback and is called by the SAX parser whenever the current XML tag is closed. When endElement is called, we pop the top of production_stack into the local variable production. Next, if production_stack is not empty, we check for a value in token. If a value is present in token then we append it to the top item in production_stack and reset the value of token; if a value is not present then we append the current tag to the top item in production_stack and add production to the global set productions. Upon termination of the algorithm, productions will contain all productions encoded in the XML parse tree

instance. The productions are in the form of lists of strings such that the first string in the list is the left-hand side of the production and the remaining strings in the list are the right-hand side of the production. Lambda productions are represented by a list containing a single string.

Figure 5.5c illustrates the grammar that is recovered from the XML parse tree in Figure 5.5b by the algorithm shown in Figure 5.6. For example, the second production listed in Figure 5.5c

$$binary\_expression$$
$$\rightarrow postfix\_expression\ PLUS\ binary\_expression$$

is recovered from the outermost `binary_expression` tag from Figure 5.5b and its three immediate children. Additionally, a lambda production is recovered from `nested_name_specifier`$_{opt}$, which is an empty tag.

5.3.2 Semi-Automated Identification of Candidate Nonterminals

Our technique for automated grammar recovery might recover an iterative grammar, which (most likely) represents a subset of the intended language. Therefore, a key issue is the identification of nonterminals that are expressed using iteration; the identified nonterminals are candidates to be rewritten with left recursion. Size metrics are useful to guide the process of identifying candidates; given the earlier descriptions of PROD, AVS, and HAL we expect nonterminals expressed using iteration to exhibit unusually large values for these three size metrics. However, the definition of unusually large is dependent on context and in some cases is subjective. Thus, while we can automatically compute these metrics for a grammar, human intervention is required to use these metrics for identification of candidate nonterminals.

5.3.3 Recovery of a Left Recursive Grammar

Figure 5.7 illustrates five productions and corresponding example sentences. The productions are expressed using iteration and generate a subset of the language that is generated by productions 1, 3, and 7 from Figure 5.4. For example, the productions in Figure 5.7 do not generate the sentence `o.foo(p).bar(q).foobar(r)` but the specified productions from Figure 5.4 do. We could add a sixth production to Figure 5.7 to generate that sentence; however, it would be easy to then identify another sentence generated by the specified productions from Figure 5.4 but not by the resulting six productions. Thus, the approach of repeatedly adding iterative productions to a grammar is neither scalable nor elegant. Furthermore, the grammar that results is unwieldy, making it difficult for a human to comprehend, and specific to a particular parsing algorithm, rendering it not useful to other applications.

Figure 5.8 illustrates our algorithm, which takes as input a nonterminal and its set of right-hand sides expressed using iteration and gives as output the set of right-hand sides expressed using left recursion. The input nonterminal is identified as a candidate for refactoring using the metrics-guided approach described in the previous section. The algorithm considers a right-hand side to be a string of symbols and assumes the following definitions:

prefix string a is a prefix of string b if the symbols in a match the first len(a) symbols of b

suffix string a is a suffix of string b if the symbols in a match the last len(a) symbols of b

Furthermore, the algorithm contains two stages: (1) *Replace Prefixes* and (2) *Remove Suffixes*. In the first stage, we introduce left recursion wherever possible, and in the second stage we eliminate portions of productions if they can be generated by the new left recursion productions.

Figure 5.9 illustrates the left recursive productions given as output by our algorithm when applied to the iterative productions in Figure 5.7. Clearly, the language generated by

the productions in Figure 5.9 is a superset of the language generated by the productions in Figure 5.7. This underscores the importance of the identification of candidate nonterminals, and, in particular, of the role of the human in the identification process. Refactoring a grammar to generate a superset of the intended language can be desirable, but performing this refactoring arbitrarily can significantly change the meaning of the grammar in addition to degrading its understandability. For example, consider Figure 5.10, which illustrates the havoc that our algorithm can wreak upon a grammar if misapplied.

5.4 Parse Tree Validation

Lämmel and Verhoef assert that a recovered grammar must be transformed so that it is useful for another application [51, page 5]. In this section we describe transformations of our recovered grammar for GNU *gcc* version 4.0.0. into three different schemas: Relax NG, LaTeX and YACC.

The Relax NG Schema: We use the Relax NG schema to verify that the recovered grammar conforms to the parse tree representation of the grammar. We do this by using *xmllint*, together with the Relax NG schema language, to validate the generated schema for the recovered grammar.

The LaTeX Schema: We automatically generate a LaTeXversion of our grammar and we found our LaTeXversion of the grammar for GNU *gcc* dialect invaluable for comparison to the ISO C++ grammar. Our removal of iteration from and incorporation of left recursion into the recovered grammar further enhanced readability. We also found the *gcc* compiler source version of the grammar difficult or impossible to comprehend and we agree with Lämmel and Verhoef that compiler grammars are not optimal reading for humans [51, page 10].

The YACC Schema: Grammar deployment is the process of turning a given grammar specification into a working parser [43]. We used our recovered grammar to automatically generate a YACC schema and we deployed the schema to the *Bison* parser generator to generate an LALR(1) parser. The LALR(1) parser contained 9,603 shift/reduce conflicts and 30,637 reduce/reduce conflicts. We also deployed the grammar to a *Bison GLR* parser generator and the GLR parser contained no conflicts. Although the Bison GLR parser for the GNU gcc dialect requires disambiguation rules, the deployment comparison supports the use of GLR parsing for ambiguous grammars [84].

5.5 Study on Grammar Recovery

In this section we present a case study in which we recover and refactor a grammar from a hard-coded parser to demonstrate the feasibility and utility of the methodology we describe in Chapter 5. In particular, we recover a grammar for GNU C++ using the technique we describe in Chapter 5.3 and then refactor the recovered iterative grammar using the metrics-guided approach we describe in Chapter 5.3.1. Finally, we evaluate the recovered and refactored GNU C++ grammar by comparing it to the ISO C++98 grammar.

In Chapter 5.5.0.1 we describe the four test suites that we use in our studies. We constructed multiple test suites because, using the methodology that we described in Chapter 5.3, we only can recover a terminal, nonterminal, or production if it is exercised by a test case; that is, if it is contained in the parse tree for a test case. Therefore, the coverage of the subject language that we obtain using a test suite is more strongly dependent on the variety than the number of terminals, nonterminals, and productions exercised. To address this need for variety we selected 12 C++ programs, or test cases, from which we constructed four test suites: (1) Benchmarks, (2) Libraries, (3) Applications, and (4) Mozilla.

In Chapter 5.5.0.2 we report results we obtained when applying our grammar recov-

ery technique to the four test suites. We report results that address the following research questions:

Q1. *How many terminals, nonterminals, and productions are recovered from the parse trees for each test case, each test suite, and all test suites?* Specifically, we wanted to determine whether the sets of terminals, nonterminals, and productions recovered from the parse trees for the GCC test case, part of the Benchmarks test suite, contain the same terminals, nonterminals, and productions recovered from the parse trees for all test cases. That is, we wanted to determine whether any test case exercises parts of the GNU C++ grammar not exercised by the GCC test case. In addition, we wanted to determine whether the Mozilla test cases exercise precisely the same parts of the GNU C++ grammar.

Q2. *How many unique productions are recovered from the parse trees for each test case and each test suite?* We wanted to determine which test cases, if any, make significant unique contributions to the overall recovered grammar. That is, we wanted to determine if particular test cases contribute the majority of the unique productions, or if all test cases contribute a proportional share of the unique productions. Moreover, we wanted to determine whether any of the four test cases contribute a disproportionally high or low number of unique productions.

Q3. *How significantly can each test case and each test suite be reduced without sacrificing coverage?* We wanted to determine how many translation units (source files) in each test case and each test suite can be removed without removing terminals, nonterminals, or productions from the recovered GNU C++ grammar.

In Chapter 5.5.0.5 we report results we obtained when applying our metrics-guided grammar refactoring approach to the recovered GNU C++ grammar. In addition, we report information gathered from our comparison of the recovered grammar to the ISO C++98 grammar. We address the following additional research questions:

Q4. *How many candidate nonterminals are identified using the three size metrics?*

We wanted to determine whether the size metrics can clearly identify a nonempty set of candidates.

Q5. *How many of the identified candidate nonterminals can be (reasonably) rewritten with left recursion?* We wanted to determine whether candidate nonterminals identified in the investigation of Q4: (a) are actually expressed using iteration and (b) can be rewritten with left recursion without significantly changing the language recognized by the grammar.

Q6. *How similar are the refactored left recursive nonterminals to their counterparts in the ISO C++98 grammar?* We wanted to compare the nonterminals produced by our refactoring algorithm (shown in Figure 5.8) to their counterparts in the ISO C++98 grammar to determine if our algorithm behaves as expected.

We obtained all results using a Dell™ OptiPlex™ 755 workstation with an Intel® Core™2 Q6600 processor, 4096 MB of RAM, and a 160GB 7200 RPM SATA hard drive on which we installed the Slackware 12.0 operation system. We used xmllint version 20630 to validate the XML-encoded parse trees, we wrote a handler for the SAX parser included with Python version 2.5.1 to recover a grammar from XML-encoded parse trees, and we implemented our left recursion recovery algorithm in Python. Finally, we wrote both Bash and Python scripts to automate execution of the grammar recovery system.

5.5.0.1 Test Suites

Table 5.1 lists information about the 12 test cases that form our four test suites. For each test case we list the name, version, number of C++translation units (TUs), and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). In C++, a *translation unit* consists of a source file and all files that it includes either directly or transitively, and we used the source code line counter [79] to compute NCLOC. For each test suite we list the three test cases that form the test suite, as well as the total numbers of C++TUs and NCLOC for the test suite. In the last row of the table we

Test Suite	Test Case	Version	C++ TUs	NCLOC (≈ K)
Benchmarks	Dr. Dobbs	1.0	407	4
	GCC	4.0.0	1,318	20
	Keystone	0.6	111	2
	Subtotal		1,836	26
Libraries	Blitz++	0.9	153	95
	Boost	1.35.0	1,646	320
	POOMA	2.4.1	229	115
	Subtotal		2,028	530
Applications	*Doxygen*	1.5.5	79	188
	FiSim	1.1b	23	14
	Fluxbox	1.0.0	119	38
	Subtotal		221	240
Mozilla	Gecko	1.9b5pre	254	215
	Necko	1.9b5pre	114	75
	XPCOM	1.9b5pre	188	120
	Subtotal		556	410
Total			4,641	1,206

Table 5.1: Test suites. *The four test suites that we use in our studies. For each test suite we list the test cases that constitute the test suite. For each test case we list the version, the number of C++ translation units (TUs), and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). The test suites contain* 12 *test cases, and the test cases contain over* 4,600 *C++ TUs and approximately* 1.2 *million NCLOC.*

list the total numbers of C++TUs and NCLOC for the four test suites. The 12 test cases in the test suites contain over 4,600 C++TUs and approximately 1.2 million NCLOC.

The first test suite, Benchmarks, consists of three benchmarks designed to exercise C++compilers, particularly their parsers and front ends. The first test case in Benchmarks is Dr. Dobbs, a test suite designed by Malloy, et al. [60] to measure the conformance of a C++parser to the ISO C++98 standard. We included only the translation units that could be successfully compiled by version 4.0.0 of g++. The GCC test case consists of the translation units from the g++.dgdirectory of the test suite for the C++compiler included with GCC [30]. While GCC contains many TUs, many of those TUs contain code written to test code optimization or code generation routines, not parser front end routines. The third test case, *Keystone*, is the combination of a test suite written by the authors to evaluate *Keystone* [64], an ISO C++98 conformant parser, and a test suite written by the authors to

evaluate g^{4}re [49], a tool chain for reverse engineering C++ programs. The *Keystone* test case is intended to complement the Dr. Dobbs test case.

The second test suite, Libraries, consists of the test suites for three C++ class template libraries: Blitz++, Boost, and Pooma. Blitz++ is a scientific computation library that uses templates to achieve performance on par with Fortran [8]. Boost is a highly-regarded free library that is peer-reviewed and portable [10]. POOMA is a collection of class templates that can be used to write parallel PDE solvers [25]. We chose these test cases because they are listed on the *GCC Testing Efforts* page [31], they are listed in the *GCC Release Criteria*, and they are each well-known for making heavy use of templates. However, as these test cases are class template libraries, we had to instantiate them to compile them (and thus to obtain parse trees for them). Therefore, to construct the Libraries test suite we used translations units from the test suites provided with each of the test cases.

The third test suite, Applications, consists of three C++ applications: *Doxygen*, FiSim, and Fluxbox. *Doxygen* is a documentation system for C, C++, and Java [85]; FiSim is a scientific modeling tool for advanced engineering of fiber and film [13]; and Fluxbox is a light-weight X11 window manager built for speed and flexibility [23]. We chose these applications for their variety of size and application, including a language processing tool, a scientific application, and a window managing system. We specifically chose FiSim for its use of the Loki library [3], which make heavy use of templates.

The fourth test suite, Mozilla, consists of three modules from the open source, cross-platform web and email application suite [68]: Gecko, Necko, and XPCOM. Gecko (a.k.a. layout) is a cross-platform rendering engine that forms the core of the Mozilla browser. Necko (a.k.a. netwerk) is the Mozilla network library and provides a platform-independent API for several layers of networking ranging from the transport to the presentation layer. XPCOM is a cross-platform variant of the well-known Component Object Model (COM). In addition to Mozilla being a popular application, it is also frequently used

as a test suite for C++ fact extractors and other C++ reverse engineering tools. For example, iPlasma is an integrated environment for analysis of object-oriented software systems written in the C++ or Java languages. The scalability of the iPlasma environment is demonstrated by its ability to handle large-scale projects in the size of millions of lines of code, including Mozilla [33, 53]. Similarly, Mozilla is used to evaluate the TUAnalyzer system [34] and the Columbus reverse engineering tool [21].

In the next two sections we report the results of applying our grammar recovery system to the four test suites.

5.5.0.2 Grammar Recovery Study

In this section we report results we obtained when applying our grammar recovery technique to the four test suites. These results address research questions Q1–Q3 from the beginning of this section. In particular, Table 5.2 addresses Q1, Table 5.3 addresses Q2, and Table 5.4 addresses Q3.

5.5.0.3 Measurements of the Recovered Grammars

Table 5.2 lists size metrics for the grammars that we recovered for each test case, for each test suite, and for all test suites. For each recovered grammar we list three size metrics: number of terminals (TERM), number of nonterminals (VAR), and number of productions (PROD). The first size metric, TERM, is listed in the third column of the table. For each test case, TERM is the number of unique terminals found in the recovered grammars for the translation units in the test case. For each test suite, TERM is listed in a row labeled Combined and is the number of unique terminals found in the recovered grammars for the test cases in the test suite. For all test suites, TERM is listed in a row labeled Recovered and is the number of unique terminals found in the recovered grammars for all test suites. The second metric, VAR, is located in the fourth column of the table,

Test Suite	Test Case	TERM	VAR	PROD
Benchmarks	Dr. Dobbs	90	125	368
	GCC	136	138	612
	Keystone	117	132	462
	Combined	137	138	620
Libraries	Blitz++	123	131	486
	Boost	127	133	723
	POOMA	122	125	517
	Combined	127	133	765
Applications	*Doxygen*	107	113	401
	FiSim	120	133	514
	Fluxbox	119	131	515
	Combined	122	133	581
Mozilla	Gecko	115	121	525
	Necko	113	123	445
	XPCOM	117	123	462
	Combined	119	124	553
Recovered		137	138	930

Table 5.2: Grammar Size. *This table lists the sizes of the grammars recovered for the test cases in our four test suites.*

while the third metric, PROD, is located in the fifth column of the table. Both of these metrics are calculated over the same sets of recovered grammars as the first metric.

The data listed in Table 5.2 shows that the grammar recovered for the GCC test case contains all but one of the 137 recovered terminals and that it contains all of the 138 recovered nonterminals. However, note that not only does this grammar contain only 65% of the total productions recovered from all test suites, but also that this grammar does not even contain the most productions of the 12 test cases. In particular, the grammar recovered for the Boost test case contains 723 productions, which is 12% more than the grammar recovered for GCC. Nonetheless, the grammar recovered for Boost does not contain 10 of the terminals and 5 of the nonterminals recovered by other test cases in the test suites. We also note that the Mozilla test cases do exercise different parts of the GNU C++ grammar, although there is significant overlap.

Table 5.3 lists statistics about the number of unique productions contributed by each test case and by each test suite to the grammars that we recovered for each test suite and

Test Suite	Test Case	# This Suite	% This Suite	All Suites	% All Suites
Benchmarks	Dr. Dobbs	3	0.5	0	0.0
	GCC	130	21.0	63	6.8
	Keystone	5	0.8	3	0.3
	Combined	N/A	N/A	74	8.0
Libraries	Blitz++	9	1.2	9	1.0
	Boost	221	28.9	113	12.2
	POOMA	30	3.9	24	2.6
	Combined	N/A	N/A	154	16.6
Applications	*Doxygen*	22	3.8	8	0.9
	FiSim	40	6.9	8	0.9
	Fluxbox	34	5.9	9	1.0
	Combined	N/A	N/A	27	2.9
Mozilla	Gecko	71	12.8	35	3.8
	Necko	11	2.0	3	0.3
	XPCOM	14	2.5	3	0.3
	Combined	N/A	N/A	47	5.0

Table 5.3: Unique Productions. *This table lists the numbers of productions uniquely recovered from each test case and test suite.*

for all test suites. For the recovered grammar for each test case we list four statistics; for the recovered grammar for each test suite we list only two statistics. We first describe the two statistics that apply only to the recovered grammars for the test cases; we then describe the two statistics that apply to the recovered grammars for the test cases as well as the recovered grammars for the test suites.

The first statistic is located in the third column of the table and is the number of unique productions contributed to the grammar for this test suite. For example, for the Gecko test case the value in the third column is 71; this means that the recovered grammar for Gecko contains 71 productions that are not found in either the recovered grammar for the Necko test case or the recovered grammar for the XPCOM test case. The second statistic is located in the fourth column of the table and is the number in the third column of the table divided by the total number of unique productions for the grammar for this test suite. Continuing the previous example, for the Gecko test case the value in the fourth column is 12.8; this means that by uniquely contributing 71 of the 553 productions in the

recovered grammar for the Mozilla test suite, Gecko uniquely contributes 12.8% of the productions in that grammar.

The third statistic in Table 5.3 is listed in the fifth column and is the number of unique productions contributed to the grammar for all test suites. For example, for the Gecko test case the value in the fifth column is 35; this means that the recovered grammar for Gecko contains 35 productions that are not found in the recovered grammars for the other 11 test cases. Similarly, for the Mozilla test suite the value in the fifth column of the table is 47; this means that the recovered grammar for Mozilla contains 47 productions that are not found in the recovered grammars for the other three test suites. The fourth statistic is listed in the sixth column of the table and is the number in the fifth column divided by the total number of unique productions for the grammar for all test suites. Continuing the previous example, for the Gecko test case the value in the sixth column is 3.8; this means that by uniquely contributing 35 of the 930 productions in the recovered grammar for all test suites, Gecko uniquely contributes 3.8% of the productions in that grammar. Similarly, for the Mozilla test suite the value in the sixth column is 5.0; this means that by uniquely contributing 47 of the 930 productions in the recovered grammar for all test suites, Mozilla uniquely contributes 5.0% of the productions in that grammar.

For three of the four test suites, one of the three test cases in the test suite contributes the majority of unique productions; only in the Applications test suite is there no test case that does so. Yet, there is no test case among the 12 that does not contribute a unique production within its test suite, though 7 of the 12 test cases uniquely contribute less than 5% of the productions in their respective test suites. When considering productions uniquely contributed among all 12 test cases, we note that Dr. Dobbs has a total of zero. We also experimented with the FTensor library [26] and found that it too failed to contribute a production among those recovered from our 12 test cases. Finally, we note that the GCC and Boost test cases each contribute disproportionally high numbers of unique productions.

5.5.0.4 Reduction of the Test Suites

Hennessy and Power present a test suite reduction technique for grammar-based software that is based on rule coverage [37]. When performing grammar recovery, we store data that allows us to apply an analogous technique. When recovering the grammar for a test case, we store in a set the name of the translation unit that contributes each new production. Upon completion of grammar recovery the resulting set of translation units represents a (possibly) reduced version of the test case, known as the *essential set*. If we apply the grammar recovery system to the reduced test case, we will obtain the same grammar as we would from the unreduced test case. However, because we did not attempt attempt to order the translation units (we simply ordered them alphabetically), we cannot guarantee that we compute the minimal test case (set of translation units). Further, as noted by Hennessy and Power, given the essential set, choosing the minimal test case that covers the productions is equivalent to the minimum cardinality hitting set, which is an intractable problem [29].

Table 5.4 lists percentage reductions achieved using the above technique. For each test case we list two numbers: the number of translation units in the essential set and the percentage reduction in the number of translation units. We also list the aggregate values for each test suite and for all test suites. The first number is located in the third column of the table and is the number of translation units in the essential set. For example, for the Gecko test case the value in the third column is 52; this means that all of the productions from the recovered grammar for Gecko can be recovered from a particular set of 52 translation units. The second number is located in the fourth column of the table and is the percentage reduction in the number of translation units. Continuing the previous example, for the Gecko test case the value in the fourth column is 79.5; this means that by eliminating all but 52 of the 254 translation units for the Gecko test case, we have reduced the Gecko test

Test Suite	Test Case	# C++ TUs	% Reduction
Benchmarks	Dr. Dobbs	91	77.6
	GCC	129	90.2
	Keystone	32	71.2
	Subtotal	252	86.3
Libraries	Blitz++	14	90.8
	Boost	85	94.8
	POOMA	17	92.6
	Subtotal	116	94.3
Applications	*Doxygen*	26	67.1
	FiSim	8	65.2
	Fluxbox	29	75.6
	Subtotal	63	71.5
Mozilla	Gecko	52	79.5
	Necko	28	75.4
	XPCOM	35	81.4
	Subtotal	115	79.1
Total		546	88.2

Table 5.4: Percentage reduction of test suites. *The reduced numbers of C++ translation units (TUs) and the percentage reductions for the individual test cases and test suites, as well as the aggregate reduction for the test suites.*

case by 79.5%.

The results in Table 5.4 show that we can achieve an 88.2% reduction in the number of C++ translation units in our test suites without sacrificing the recovery of any terminals, nonterminals, or productions. We did not leverage this reduction in our studies or even measure the time savings that result. However, we note that the largest reduction, 94.3%, is for the Libraries test suite and that this test suite accounted for the vast majority of the running time consumed by our grammar recovery process [18]. Furthermore, this result has obvious utility to anyone who wishes to recover a grammar for a C++ dialect from test cases.

5.5.0.5 Grammar Refactoring Study

In this section we report results we obtained when applying our metrics-guided grammar refactoring approach to the GNU C++ grammar recovered in Chapter 5. These

Grammar	TERM	VAR	PROD	LR	RR
Recovered	137	138	930	0	39
Refactored	137	138	601	3	39
ISO	117	184	479	27	6

Table 5.5: Grammar Size Metrics. *This table lists size metrics for the recovered grammar before and after refactoring, and the ISO C++98 grammar.*

results address research questions Q4 and Q5 from the beginning of this section. In addition, we report information gathered by inspection of the refactored grammar and from our comparison of the recovered grammar to the ISO C++98 grammar. This information addresses question Q6 from the beginning of this section.

As described in Chapter 5.3.1, the three grammar size metrics PROD, AVS, and HAL can be computed at the nonterminal granularity. That is, we can compute the value of the metric for a nonterminal and its set of right-hand sides and we did so for each of the 138 nonterminals in the recovered GNU C++ grammar. When examining the results we found that one nonterminal had values for PROD and AVS that were one order of magnitude larger than the values for any other nonterminal. Further, we found that two nonterminals had values for HAL that were one order of magnitude larger than the values for any other nonterminal. More specifically, the values of PROD and AVS were particularly large for `postfix_expression` and the value of HAL was particularly large for `direct_declarator` and `direct_abstract_declarator`. Indeed, upon inspection of these three nonterminals, we found that each had expressions that were expressed using iteration.

Table 5.5 lists data about the recovered grammar before refactoring, the recovered grammar after refactoring by replacing iteration with left recursion, and (for comparison) the ISO C++98 grammar. In particular, we list the numbers of terminals, nonterminals, and productions, as well as the numbers of nonterminals that use left and right recursion. Only two values differ between the recovered and refactored grammars; specifically, the number

of productions and the number of nonterminals using left recursion. Note that the number of productions for our recovered grammar has decreased by 329; this can largely be attributed to `postfix_expression` having 337 productions before the refactoring. Also note that the recovered grammar now contains three nonterminals that use left recursion; in particular, the three nonterminals that we identified as candidates using size metrics.

The recovered grammars (before and after refactoring) contain 20 terminals not contained in the ISO grammar; these terminals are GNU extensions to C++, such as the keyword `__restrict__`. Additionally, the recovered grammars contain 46 less nonterminals than the ISO grammar; however, this is largely due to the 41 distinct productions that only implement optionality in the ISO grammar. The recovered grammars typically do not use distinct productions to implement optionality, but rather simply introduce a lambda production for the optional nonterminal. The two most striking differences between the recovered grammars and the ISO grammar are in the number of productions and the use of recursion. Before refactoring, the recovered grammar contains almost twice as many productions as the ISO grammar; this is because iteration is used in the original recovered grammar. Also, while the recovered grammars actually use more recursion than the ISO grammar, because that recursion originates from a recursive descent parser, the recovered grammars use right recursion rather left recursion.

Figure 5.5.0.5 illustrates the 22 productions for `postfix_expression` that remain in the recovered grammar after refactoring. A comparison of these productions to those in Figure 5.5.0.5 yields several items of interest. Firstly, the sets of productions are very similar and, in fact, significant subsets of the productions are identical. Secondly, we note that production 22 in Figure 5.5.0.5 is a GNU extension. Thirdly, some optionality in the ISO grammar is expressed as multiple productions in the recovered grammar. Finally, note that in the recovered grammar, parentheses are included in the productions for the nonterminal $expression\_list_{opt}$; that is, $expression\_list_{opt}$ in the recovered grammar

Grammar	TERM	VAR	PROD	LR	RR
Recovered	137	138	930	0	39
ISO	117	184	479	27	6

Table 5.6: Grammar Coverage I (Before Refactoring). *This table lists size metrics for the grammars recovered for the test cases in our four test suites.*

Grammar	TERM	VAR	PROD	LR	RR
Recovered	137	138	601	3	39
ISO	117	184	479	27	6

Table 5.7: Grammar Coverage II (After Refactoring). *This table lists size metrics for the grammars recovered for the test cases in our four test suites.*

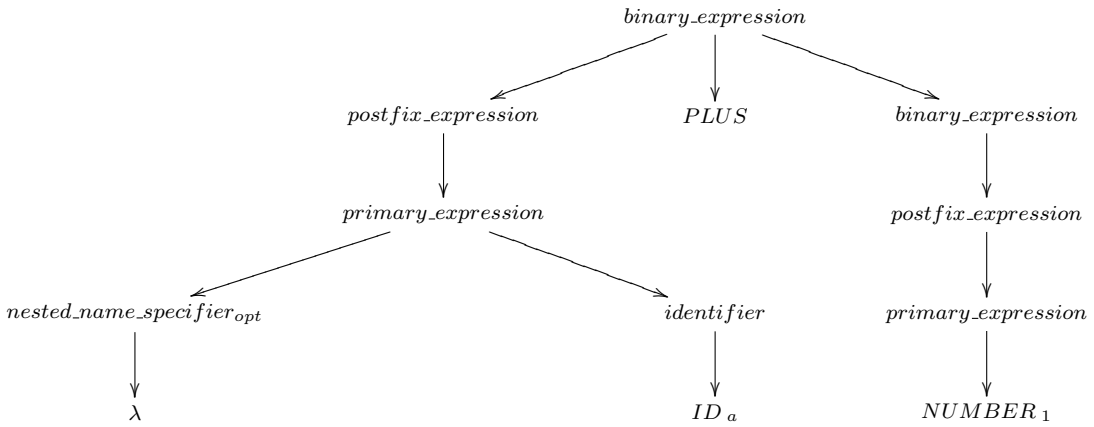
is equivalent to `(` $expression\_list_{opt}$ `)` in the ISO grammar.

Table 5.6 lists data about our recovered grammar (before refactoring) and the ISO C++98 grammar. In particular, we list the numbers of terminals, nonterminals, and productions, as well as the numbers of nonterminals that use left and right recursion. The recovered grammar contains 20 additional terminals not contained in the ISO grammar; these terminals are GNU extensions to C++, such as the keyword `__restrict__`. Additionally, the recovered grammar contains 46 less nonterminals than the ISO grammar; however, this is largely due to the 41 distinct productions that only implement optionality in the ISO grammar. The recovered grammar typically does not use distinct productions to implement optionality, but rather simply introduces a lambda production for the optional nonterminal. The two most striking differences between the two grammars are in the number of productions and the use of recursion. The recovered grammar contains almost twice as many productions as the ISO grammar; this is because iteration is used in the recovered grammar. Also, while the recovered grammar actually uses more recursion than the ISO grammar, because it originates from a recursive descent parser, the recovered grammar uses right recursion rather left recursion.

Table 5.7 lists data about our recovered grammar (after refactoring) and the ISO C++98

grammar. Only two values differ from those listed in Table 5.6; specifically, the number of productions and the number of nonterminals using left recursion. Note that the number of productions for our recovered grammar has decreased by 329; this can largely be attributed to `postfix_expression` having 337 productions before the refactoring. Also note that the recovered grammar now contains three nonterminals that use left recursion; in particular, the three nonterminals that we identified as candidates using size metrics.

Figure 5.13 illustrates the 22 productions for `postfix_expression` that remain in the recovered grammar after refactoring. A comparison of these productions to those in Figure 5.4 yields several items of interest. Firstly, the sets of productions are very similar and, in fact, significant subsets of the productions are identical. Secondly, we note that production 22 in Figure 5.13 is a GNU extension. Finally, some optionality in the ISO grammar is expressed as multiple productions in the recovered grammar.



(a) Parse Tree

```
<binary_expression>
   <postfix_expression>
      <primary_expression>
         <nested_name_specifier_opt />
         <identifier>
            <token type='ID'
value='a' />
         </identifier>
      </primary_expression>
   </postfix_expression>
   <token type='PLUS' />
   <binary_expression>
      <postfix_expression>
         <primary_expression>
            <token type='NUMBER'
value='1' />
         </primary_expression>
      </postfix_expression>
   </binary_expression>
</binary_expression>
```

(b) Parse Tree in XML

$$
\begin{aligned}
&\textit{binary\_expression} \\
&\quad \rightarrow \textit{postfix\_expression} \\
&\quad \rightarrow \textit{postfix\_expression} + \textit{binary\_expression} \\
&\textit{postfix\_expression} \\
&\quad \rightarrow \textit{primary\_expression} \\
&\textit{primary\_expression} \\
&\quad \rightarrow \textit{nested\_name\_specifier}_{opt}\ \textit{identifier} \\
&\quad \rightarrow NUMBER \\
&\textit{nested\_name\_specifier}_{opt} \\
&\quad \rightarrow \lambda \\
&\textit{identifier} \\
&\quad \rightarrow ID
\end{aligned}
$$

(c) Corresponding Grammar

Figure 5.5: Parse Trees and Corresponding Grammar. *Two representations of an example parse tree for the expression* $a + 1$ *and the corresponding grammar.*

```
1   productions = set()
2   class Handler(xml.sax.ContentHandler):
3       def __init__(self):
4           self.production_stack = [ ]
5           self.token = None
6
7       def startElement(self, tag, attrs):
8           self.production_stack.append([tag])
9           if 'token' == tag:
10              self.token = 'token(%s)' % attrs['type']
11
12      def endElement(self, tag):
13          production = self.production_stack.pop()
14          if self.production_stack:
15              if self.token:
16                  self.production_stack[-1].append(self.token)
17                  self.token = None
18              else:
19                  self.production_stack[-1].append(tag)
20                  productions.add(production)
```

Figure 5.6: Grammar Recovery Algorithm. *The important steps in our grammar recovery algorithm expressed as a SAX parser and in Python syntax.*

Production	**Sentence**
$postfix\_expression$	
$\rightarrow primary\_expression$	`o`
$\rightarrow primary\_expression$ `.` $id\_expression$	`o.x`
$\rightarrow primary\_expression$ `.` $id\_expression$ `(` $expression\_list_{opt}$ `)`	`o.foo(p)`
$\rightarrow primary\_expression$ `.` $id\_expression$ `(` $expression\_list_{opt}$ `)` `.` $id\_expression$	`o.foo(p).y`
$\rightarrow primary\_expression$ `.` $id\_expression$ `(` $expression\_list_{opt}$ `)` `.` $id\_expression$ `(` $expression\_list_{opt}$ `)`	`o.foo(p).bar(q)`

Figure 5.7: Iterative Productions. *An iterative expression of selected productions from Figure 5.4 and corresponding sentences.*

```
1   replace_iteration(lhs,rhsSet)
2     # PHASE 1: Replace Prefixes
3     max_len = len(longest rhs in rhsSet)
4     do
5       foreach j in 1..max_len
6         partition rhsSet into three sets: len_j, len_lt_j, len_gt_j
7         foreach x in len_j
8           foreach y in len_gt_j
9             if x is a prefix of y
10              remove first j symbols of y
11              add lhs to front of y
12              if y is not unique in rhsSet
13                remove y from rhsSet
14    while(rhsSet changes)
15
16    # PHASE 2: Remove Suffixes
17    partition rhsSet into two sets: left_recursive,
      not_left_recursive
18    foreach lr in left_recursive
19      lr_part = lr.pop_front() # lr without the recursive "call"
20      for n in not_left_recursive where len(n) >= len(lr_part)
21        if lr_part is a suffix of n
22          remove last len(lr_part) symbols of n
23          if n is not unique in rhsSet
24            remove n from rhsSet
```

Figure 5.8: Left Recursion Recovery Algorithm. *The important steps in our left recursion recovery algorithm.*

$$
\begin{aligned}
&\mathit{postfix\_expression}\\
&\quad \rightarrow \mathit{primary\_expression}\\
&\quad \rightarrow \mathit{postfix\_expression}\ \texttt{.}\ \mathit{id\_expression}\\
&\quad \rightarrow \mathit{postfix\_expression}\ \texttt{(}\ \mathit{expression\_list}_{opt}\ \texttt{)}
\end{aligned}
$$

Figure 5.9: Left Recursive Productions. *A left recursive expression of the productions from Figure 5.7.*

$$
\begin{aligned}
&\mathit{selection\_statement}\\
&\quad \rightarrow \texttt{if (}\ \mathit{condition}\ \texttt{)}\ \mathit{statement}\\
&\quad \rightarrow \texttt{if (}\ \mathit{condition}\ \texttt{)}\ \mathit{statement}\ \texttt{else}\ \mathit{statement}\\
&\quad \rightarrow \texttt{switch (}\ \mathit{condition}\ \texttt{)}\ \mathit{statement}
\end{aligned}
$$

(a) Before Refactoring

$$
\begin{aligned}
&\mathit{selection\_statement}\\
&\quad \rightarrow \texttt{if (}\ \mathit{condition}\ \texttt{)}\ \mathit{statement}\\
&\quad \rightarrow \mathit{selection\_statement}\ \texttt{else}\ \mathit{statement}\\
&\quad \rightarrow \texttt{switch (}\ \mathit{condition}\ \texttt{)}\ \mathit{statement}
\end{aligned}
$$

(b) After Refactoring

Figure 5.10: Result of Misapplying the Left Recursion Recovery Algorithm. *The consequences of misapplying the algorithm illustrated in Figure 5.8.*

postfix_expression

→ *primary_expression* (1)

→ *postfix_expression* [*expression*] (2)

→ *postfix_expression* (*expression_list*$_{opt}$) (3)

→ *simple_type_specifier* (*expression_list*$_{opt}$) (4)

→ typename ::$_{opt}$ *nested_name_specifier identifier*
(*expression_list*$_{opt}$) (5)

→ typename ::$_{opt}$ *nested_name_specifier*
template$_{opt}$ *template_id* (*expression_list*$_{opt}$) (6)

→ *postfix_expression* . template$_{opt}$ *id_expression* (7)

→ *postfix_expression* -> template$_{opt}$ *id_expression* (8)

→ *postfix_expression* . *pseudo_destructor_name* (9)

→ *postfix_expression* -> *pseudo_destructor_name* (10)

→ *postfix_expression* ++ (11)

→ *postfix_expression* -- (12)

→ dynamic_cast < *type_id* > (*expression*) (13)

→ static_cast < *type_id* > (*expression*) (14)

→ reinterpret_cast < *type_id* > (*expression*) (15)

→ const_cast < *type_id* > (*expression*) (16)

→ typeid (*expression*) (17)

→ typeid (*type_id*) (18)

Figure 5.11: Nonterminals Expressed Using Left Recursion. *ISO Version*

postfix_expression

→ *primary_expression* (1)

→ *postfix_expression* `[` *expression* `]` (2)

→ *postfix_expression* $expression\_list_{opt}$ (3)

→ *simple_type_specifier* (4)

→ `typename` *nested_name_specifier identifier* (5)

→ `typename ::` *nested_name_specifier template_id* (6)

→ `typename ::` *nested_name_specifier* `template` *template_id* $expression\_list_{opt}$ (7)

→ *postfix_expression* `.` *id_expression* (8)

→ *postfix_expression* `.` `template` *id_expression* $expression\_list_{opt}$ (9)

→ *postfix_expression* `->` `template` *id_expression* $expression\_list_{opt}$ (10)

→ *postfix_expression* `->` *id_expression* (11)

→ *postfix_expression* `.` *pseudo_destructor_name* $expression\_list_{opt}$ (12)

→ *postfix_expression* `->` *pseudo_destructor_name* $expression\_list_{opt}$ (13)

→ *postfix_expression* `++` (14)

→ *postfix_expression* `--` (15)

→ `dynamic_cast` `<` *type_id* `>` `(` *expression* `)` (16)

→ `static_cast` `<` *type_id* `>` `(` *expression* `)` (17)

→ `reinterpret_cast` `<` *type_id* `>` `(` *expression* `)` (18)

→ `const_cast` `<` *type_id* `>` `(` *expression* `)` (19)

→ `typeid` `(` *expression* `)` (20)

→ `typeid` `(` *type_id* `)` (21)

→ `(` *type_id* `)` `{` *initializer_list* `}` (22)

Figure 5.12: Nonterminals Expressed Using Left Recursion. *Recovered and Refactored Version*

postfix_expression

→ *primary_expression* (1)

→ *postfix_expression* `[` *expression* `]` (2)

→ *postfix_expression expression_list*$_{opt}$ (3)

→ *simple_type_specifier* (4)

→ `typename` *nested_name_specifier identifier* (5)

→ `typename ::` *nested_name_specifier template_id* (6)

→ `typename ::` *nested_name_specifier* `template` *template_id expression_list*$_{opt}$ (7)

→ *postfix_expression* `.` *id_expression* (8)

→ *postfix_expression* `.` `template` *id_expression expression_list*$_{opt}$ (9)

→ *postfix_expression* `->` `template` *id_expression expression_list*$_{opt}$ (10)

→ *postfix_expression* `->` *id_expression* (11)

→ *postfix_expression* `.` *pseudo_destructor_name expression_list*$_{opt}$ (12)

→ *postfix_expression* `->` *pseudo_destructor_name expression_list*$_{opt}$ (13)

→ *postfix_expression* `++` (14)

→ *postfix_expression* `--` (15)

→ `dynamic_cast <` *type_id* `> (` *expression* `)` (16)

→ `static_cast <` *type_id* `> (` *expression* `)` (17)

→ `reinterpret_cast <` *type_id* `> (` *expression* `)` (18)

→ `const_cast <` *type_id* `> (` *expression* `)` (19)

→ `typeid (` *expression* `)` (20)

→ `typeid (` *type_id* `)` (21)

→ `(` *type_id* `) {` *initializer_list* `}` (22)

Figure 5.13: Production after Left Recursion Recovery. *A left recursive expression of the recovered productions for* `postfix_expression`.

Chapter 6

Construction of an Abstract Semantic Graph

In Chapter 5, we described the four modules that we developed to complete the first phase of the construction of Hylian, our system that builds ASGs for comprehension and analysis of C++ applications. This first phase entailed using the GNU compiler suite to parse the input program and the development of post-processing modules to generate and validate the parse trees, and to recover the gcc C++ grammar. In this chapter, we describe the construction of abstract semantic graphs (ASGs) from the parse trees, their storage and representation using the Graph Exchange Language (GXL), code generation from the ASG to facilitate verification of the ASG, and an interactive GXL viewer to assist in comprehension of the ASG and the corresponding program. To facilitate analysis of user code, without the noise and clutter of library code, we have developed an extension to Hylian that builds abbreviated ASGs consisting of information about user code, but not library code. We refer to the Hylian module that builds abbreviated ASGs as a phantom parser, because some of the type information in the abbreviated ASGs is unknown. We also summarize CppInfo, a previously developed schema designed specifically to be capable

of representing constructs in the C++ language, including templates [47, 48]. However, since in its current form, CppInfo is not expressive enough to handle expressions [46], we also describe our extensions to CppInfo to handle expressions, and extensions to enable validation of the abbreviated ASGs generated in a phantom parse. We use our extended version of CppInfo to validate the ASGs built with Hylian.

Construction of an ASG entails attaching semantic information to the names used in the input program, including evaluation and lookup of constants, full type resolution for the names, determination of type equivalency and type promotion, full and partial template instantiation, operator overload resolution, function and method resolution including argument dependent (Koenig) name lookup, implicit method invocation commonly used in class constructor and destructor creation and deletion.

6.1 Problems in the Construction of an ASG

The attachment of type information to the names used in an application is fairly straightforward if the program is written in a procedural language such as C or Fortran. However, attaching semantic information to names used in applications written in a multi-paradigm, composite language [80, 15], such as C++, present unique challenges. These features include data hiding, generics, multiple inheritance and other constructs that make the attachment of semantic information much more difficult than in the processing of procedural languages. These challenges comprise the bulk of the effort required for the construction of the Hylian analysis system. The most imposing of these challenges, name lookup, actually sounds uniquely trivial, yet a solution to this problem entails resolving type information for virtually every production in the C++ grammar. Thus, name lookup is difficult firstly because of the breadth of the problem, since the C++ grammar is one of the largest grammars in use, yet more importantly, is the most complex grammar [73]. For

example, the impurity metric applied to the C++ grammar shows that, at 85%, the C++ grammar contains a considerable density in the edges in the closure of the call graph, especially compared to the grammars for C, Java and C#. The impurity metric, together with the application of McCabe's metric to the C++ grammar, provide further evidence of the complexity, and the tight coupling of the C++ grammar productions [65, 73].

As an illustrative example of the issues involved in name lookup in C++, consider resolution of the simple three-operand expression for printing "Hello World," illustrated here:

```
1    int main() { std::cout ≪ "Hello, World!" ≪ std::endl; }
```

The first operand, std::cout, is of type std::basic_ostream<char> and the second is a string literal, of type **const char**∗. Resolving the correct implementation of **operator**≪ involves examining seventeen methods of the instantation of std::basic_ostream, and five function templates in the std namespace, that accept a std::basic_ostream object as its first argument. Nine of the twenty-two candidate resolutions are shown in an excerpt of the definition of the std::basic_ostream class template in Figure 6.1. The parameters for the partially-speciallized function template listed on lines 11 through 13 most closely match the types of the first two operands. So, that function is instantiated, inserted into the ASG, and the type of the subexpression is std::basic_ostream.

The second subexpression, with the result of the first subexpression on the left, and std::endl on the right, is more complicated. In this case, endl is an *unqualified-id* naming the function template listed on lines 36 through 38. Resolving name lookup for endl is problematic because, normally, operands in executable code reference memory locations in the compiled code. However, endl is the name of a function template, which is a compiler artifact with an ambiguous name and, therefore, has not been instantiated. In the C++ grammar endl is an *unqualified-id*, as opposed to a *template-id*, which is an *unqualified-id* whose template parameters have been resolved. When the arguments to

endl have been resolved, the function template is instantiated and the *template-id* is no longer an ambiguous name but a function pointer referencing an instantiation of std::endl. However, since we do not explicitly supply the template arguments to std::endl, we do not actually have a *template-id* but are left with an ambiguous *unqualified-id*. Therefore, the closest match for the second subexpression is the method listed on lines 19 and 20 in Figure 6.1. This overload for **operator**≪ accepts a function pointer to a function that accepts an std::basic_ostream, instantiated with the same template arguments as the parent class (that is, the left-hand side of the ≪ operator). The match means that std::endl is resolved, without explicitly defining its template arguments, so it is instantiated with its _CharT argument set to **char** and its _Traits argument set to std::char_traits<**char**>.

```
1   namespace std {
2     template<typename _CharT, typename _Traits = char_traits<_CharT> >
3     class basic_ostream {
4       typedef basic_ostream<_CharT, _Traits> __ostream_type;
5
6       // There are 22 occurances of operator<< in this template class
7       template<typename _CharT2, typename _Traits2>
8         friend basic_ostream<_CharT2, _Traits2>&
9         operator<<(basic_ostream<_CharT2, _Traits2>&, const _CharT2*);
10
11      template<typename _Traits2>
12        friend basic_ostream<char, _Traits2>&
13        operator<<(basic_ostream<char, _Traits2>&, const char*);
14
15      template<typename _CharT2, typename _Traits2>
16        friend basic_ostream<_CharT2, _Traits2>&
17        operator<<(basic_ostream<_CharT2, _Traits2>&, const char*);
18
19      inline __ostream_type&
20        operator<<(__ostream_type& (*__pf)(__ostream_type&));
21
22      inline __ostream_type&
23        operator<<(__ios_type& (*__pf)(__ios_type&));
24
25      inline __ostream_type&
26        operator<<(ios_base& (*__pf) (ios_base&));
27
28      __ostream_type& operator<<(int __n);
29      __ostream_type& operator<<(const void* __p);
31      __ostream_type& operator<<(__streambuf_type* __sb);
32    };
33    template<typename _CharT, typename _Traits>
34    basic_ostream<_CharT, _Traits> &operator<<(basic_ostream<_CharT, _Traits>
    &__out,
35                                               const _CharT *__s);
36    template<typename _CharT, typename _Traits>
37      basic_ostream<_CharT, _Traits>& endl(basic_ostream<_CharT, _Traits>& __os)
38      { return flush(__os.put(__os.widen('\n'))); }
39  }
```

Figure 6.1: An excerpt from the ostream header file.

6.2 ASG Schema

As a starting point for validation of our ASGs, we use a previously developed schema, CppInfo, consisting of 137 nodes developed for the g^4re system [46]. CppInfo contains information about: (1) declarations, such as classes, class templates, class template instantiations, and class template specializations; (2) namespaces; (3) functions, including function templates and function template instantiations; (4) variables; and (5) most statements, including control statements, exception statements, and function calls. However, g^4re was not developed for statement level analysis so that even though CppInfo includes nodes for statements, they were never used for validation of the g^4re system and, more

importantly, CppInfo does not contain information to support validation of expressions. Therefore, to validate expressions, we augment CppInfo with fifty-eight additional nodes, including six nodes for unary expressions, twenty-two nodes for binary expressions, and twenty-four nodes for additional expressions such as a ternary operator and C++ style casts. In addition, to enable validation of ASGs generated with the phantom parser, we added five additional nodes, which are explained in Chapter 6.4.

6.3 Construction of a Full ASG

The algorithm in Figure 6.2 parses the XML representation of the parse-tree of the original user application code. The parse-tree parser is a top-down SAX XML parser, but we prefer to handle the grammar productions in bottom-up fashion. Thus, to simulate a bottom-up parse we must buffer the XML tags that represent terminals and non-terminals from the original user application code, and delay the semantic actions until a complete sentence encountered. The semantic action for each production is handled by a function that accepts two lists: (1) a list of the syntax symbols, and (2) a list of their corresponding semantic values. The return value of the semantic action function is the semantic value of that production.

To correctly handle template classes and functions, we delay total evaluation of their subtree until we encounter a use later in the parse tree. When a template definition is encountered, we must parse enough of its tree to determine its declaration all the while buffering the entire subtree for later use. Once we determine the declaration, we stop handling semantic actions until the matching close tag for the template declaration is reached. We then store the template declaration and its corresponding parse tree in a dictionary until a usage of the template requires instantiation.

The algorithm for ASG construction uses seven data structures and a while loop that examines each XML tag in turn. The first data structure, tag, is the current XML tag. The next two data structures facilitate a bottom up parse using the tags from the top-down XML parser: syntaxStack, a stack of lists that contain terminals and non-terminals encountered in the bottom-up parse; and semanticStack, a stack of lists that contain semantic values encountered in the bottom up parse of the parse tree. The final four data structures facilitate template declaration buffering and instantiation: parsetreeBuffer, a dictionary that maps a template declaration to its respective parse tree; currentParsetreeBuffer, a buffer for a

```
1   declare tag: XML tag
2   declare syntaxStack: stack
3   declare semanticStack: stack
4   declare parsetreeBuffer: dictionary
5   declare currentParsetreeBuffer: string
6   declare isBuffering: boolean
7   declare currentTemplateDecl: template declaration
8
9   push an empty list onto semanticStack and syntaxStack
10  tag ← getTag()
11  while ( more tags in XML Parse Tree ) :
12      if (tag is startTag) :
13          if ( tag is template declaration and not isBuffering ) :
14              currentParsetreeBuffer ← empty parse tree
15              currentTemplateDecl ← NULL
16              isBuffering ← TRUE
17          if ( isBuffering ) :
18              store tag in currentParsetreeBuffer
19              if (currentTemplateDecl is not NULL) :
20                  continue at top of while
21          append tag to list at top of syntaxStack
22          if ( tag is tokenTag ) :
25              // keyword, constant, symbol, or identifier
26              if ( tag is literal ) :
27                  determine representation and type
28                  append semantic value of the literal to list at top of semanticStack
29              else if ( tag is name ) :
30                  if ( tag is keyword ) :
31                      replace 'token' with actual keyword in list at top of
32                          syntaxStack (e.g., class)
33                      append NULL to list at top of semanticStack
34                  else :
35                      push identifier onto list at top of semanticStack
36              else if ( tag is symbolTag ) :
37                  replace 'token' with symbol value (e.g., replace token with '+')
38                  append NULL to list at top of semanticStack
39          else :
40              append NULL to list at top of semanticStack
41      else if ( tag is endTag ) :
42          if ( isBuffering ) :
43              store tag in currentParsetreeBuffer
44          if (not isBuffering xor currentTemplateDecl is not NULL) :
45              call function to perform semantic-action on_tag_name
46                  i.e., for the tag "</class_head>" call on_class_head()
47              pop syntaxStack
48              pop semanticStack
49              replace NULL in list at top of semanticStack with result of
50                  semantic-action, i.e. the semantic value of the production
51
52          if (isBuffering and currentTemplateDecl is not NULL) :
53              add to parsetreeBuffer, where the key is the declaration, and the
54                  value is the partially created buffer (in progress).
55          if (matching close of template_declaration that started buffering) :
56              isBuffering ← FALSE
57      tag ← getTag()
```

Figure 6.2: The algorithm used to convert a parse tree into a bottom-up style parse

parse tree of a template so that when a template declaration is encountered the parse action is delayed until the template is instantiated and fully defined by its usage; isBuffering, a boolean indicating that the current tag is the child of a template declaration; and, finally, currentTemplateDecl, a handle for a declaration that contains information to facilitate name lookup for the currently buffered template declaration, which contains scope information, name, arguments for the template, and possibly information about parameters to a function template.

The algorithm consists of a while loop, lines 11 through 57, that examines each start and end tag in the XML tree. The algorithm examines start tags first, lines 12 through 40, taking special actions for the start of a template declaration, handling the buffering of template declarations, and lexical tokens from the original C++ source code. If the start tag is the beginning of a template declaration, and we are not currently buffering a template declaration, then we reset the values of currentParsetreeBuffer and currentTemplateDecl and turn on buffering (lines 13 through 16). If parse tree buffering is turned on, then store the current tag in the current parse tree buffer (lines 17 and 18). If the parse tree is being buffered, and we have identified the declaration for the template, then continue looping (lines 19 and 20). If a token tag is encountered, lines 22 through 38, then we examine the attributes on the tag and take one of four actions: (1) if the token represents a C++ keyword, then the keyword is inserted into the syntaxStack (lines 30 through 33); (2) if the token represents a C++ literal, then the type and representation must be determined, and a literal token is inserted into the syntaxStack and the value of the literal is inserted into the semanticStack (lines 26 through 28); (3) if the token represents a user-defined identifier, then an id token is inserted in the syntaxStack and the identifier is inserted into the semanticStack (line 35); and finally (4) if the token represents a symbol, then the appropriate symbol is inserted into the syntaxStack (lines 36 through 38). Lines numbered 24 and 25 are skipped in this algorithm, the reason for this will be address in later in Chapter

6.4.

Finally, the algorithm examines end tags, lines 41 through 56. If buffering is currently on, then the end tag is added to the current parse tree buffer (lines 42 and 43). If buffering is currently off, or if buffering is on but the declaration for template being buffered has not yet been determined, then (1) perform the semantic action for the current production, as determined by the current close tag (left-hand side) and the list of symbols on top of the syntaxStack (right-hand side); (2) pop the current production off the syntaxStack and its semantic values off the semanticStack; and (3) insert the semantic value of the production, as determined by step (1), into the semantic value list on top of the semanticStack (line 44 through 49). If parse tree buffering is currently on and the declaration for the template definition being buffered was determined in the previous semantic action function, then the declaration and a handle to the current parse tree buffer are added into the parsetreeBuffer dictionary (lines 52 through 54). And, finally, if the close tag is the match to the open tag that started buffering a template declaration parse tree, then buffering is turned off (lines 55 and 56).

6.4 Library Analysis in User Applications

For source code analysis, the user may or may not be interested in comprehension or testing of the library or libraries that are used in the program. For example, when tracking memory leaks in a program using Valgrind, Purify or Insure++, the user is interested in a determination of whether or not the leak is in user code or library code because, if the leak is in user code, the developer can find and fix the leak. If the memory leak is in the library, the developer must decide (1) if they are using the library properly [61], (2) to live with the leak, or (3) choose an alternative library.

However, there are other analysis activities where the user is completely disinterested in comprehension or testing of the library code, and more importantly, inclusion of header files for some libraries obscures and obfuscates comprehension and analysis of the user application code. For example, consider libraries that make heavy use of the template construct in C++, such as the C++ Standard Library, or the Boost C++ libraries[10]. The results that we present in Chapter 6.9 illustrate a large disparity in the magnitude of an ASG for a "Hello World" program that includes type information and executable statements for library code, as compared with an ASG that only includes user code and excludes library code as part of the ASG.

This disparity motivated our inclusion of an option in the Hylian system to exclude library code as part of ASG construction. This option resulted in ASGs that are built more efficiently, are easier to comprehend and, therefore are more useful in analysis of applications that utilize heavily templated library code. However, this additional option further complicates name lookup resolution and requires construction of a schema that incorporates additional kinds of vertices and edges that appear in the semantic graph. These additional ASG elements build on the original schema and include information that is used as a placeholder for information that describes names used in the library. We refer to

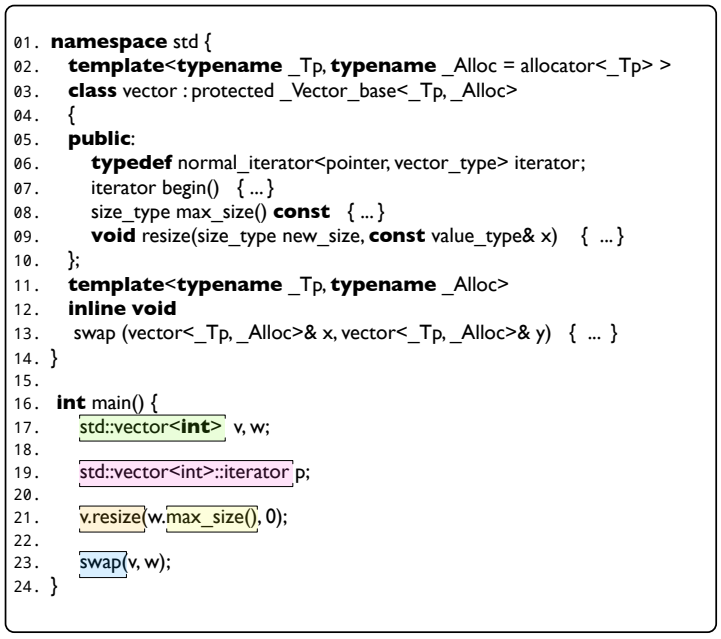
these placeholder elements as phantom elements and we refer to the ASG that contains these elements as a *phantom ASG*. Our results illustrate that, typically, a phantom ASG is smaller and therefore potentially facilitates easier analysis and comprehension of the user application, but required more effort and resources in development.

To generate phantom ASGs, we first need to know which declarations come from system header files and which come from user code. To accomplish this, we modified the parser tree generator (introduced in Chapter 5.1) to attach to each token the file name and line number it came from, and a Boolean indicating if that file is considered a system file. It is not enough to examining the path of the file for two reasons: (1) sometimes we only have a relative path name to the file, and (2) the user can specify on the command line what include file search paths are considered to be system directories with the `-isystem` switch (as opposed to the more familiar `-I` switch). With this new metadata in the parse tree, we can update the algorithm listed in Figure 6.2 to check for the "in-system-header" attribute, and mark the tag if needed.

```
23    if ( getAttribute(tag, "in-system-header") is true ):
24        mark tag as a phantom token
```

This update to the algorithm marks every token that comes in from a system header file as a phantom token. Every semantic action has to be sure to check for it before applying its actions. Only declarations that appear at namespace scope are considered, and we only keep track of their name. When we find a class, and its name is tagged as a phantom token, we do not parse its base specifiers, template declarations, or any members that appears in the class's body. The same applies for functions that are defined at namespace-scope. We identify, and store its name, but its return type, parameter declaration, and function body are not parsed.

To generate phantom ASGs, we need to augment the ASG schema we presented in Chapter 6.2. While we don't need to store definitions that appear in system header files,



```
01. namespace std {
02.   template<typename _Tp, typename _Alloc = allocator<_Tp> >
03.   class vector : protected _Vector_base<_Tp, _Alloc>
04.   {
05.   public:
06.     typedef normal_iterator<pointer, vector_type> iterator;
07.     iterator begin()  { ... }
08.     size_type max_size() const  { ... }
09.     void resize(size_type new_size, const value_type& x)  { ... }
10.  };
11.  template<typename _Tp, typename _Alloc>
12.  inline void
13.   swap (vector<_Tp, _Alloc>& x, vector<_Tp, _Alloc>& y)  { ... }
14. }
15.
16. int main() {
17.    std::vector<int>  v, w;
18.
19.    std::vector<int>::iterator p;
20.
21.    v.resize(w.max_size(), 0);
22.
23.    swap(v, w);
24. }
```

Figure 6.3: Phantom Parse This figure illustrates the important points of interest for a program that uses the vector class template from the Standard C++ Library.

we do need to provide nodes in the graphs as placeholders to those definitions. Figure 6.3 shows an excerpt of the vector header file, and small sample program that makes use of it. There are five snippets of code colored in user-code segment. The table in Table 6.1 lists the additional ASG nodes required to generate phantom ASGs. The first column, "ASG Schema Element" lists the name of the new node. The second column, "Cognition Level" lists how much information we may infer about the node. We use three Cognition Level types: (1) a "known-known" declaration is a declaration that appears in user-code, we know all there is to know about it; (2) a "known-unknown" declaration is a declaration that we known exists, because we have its name and where it was declared, but that is all that is known about it; and (3) an "unknown-unknown" declaration is a declaration that we did not expect, we know nothing about it. The third column, "Description" gives a quick description about how the ASG node is used. And, the fourth column, "Example Use", references an example use of the node in Figure 6.3.

The schema element in the first of the table, a known-unknown PhantomReference, is used to hold names of non-type values found in namespace-scope of system header files. These values are commonly function names, but occasionally a variable at namespace-scope is declared (the most common being std::cout). In Figure 6.3, the function *swap*, defined on Lines 11 through 13, and used on Line 23, is an example of a known-unknown PhantomReference. The second schema element, a unknown-unknown PhantomReference, is used to hold declarations that were never parsed, but their name can be inferred from its use. Because the bodies of phantom classes are not parsed, this element is used for accessing class members. In Figure 6.3, the use of *resize* on Line 21 creates an unknown-unknown PhantomReference. It was not found when the class template vector was parsed, but we know it from its invocation. The third schema element, a known-unknown PhantomTypename, is used to hold type declarations found at namespace-scope inside system header files. These declarations are classes, struct, union, and typedefs. In Figure 6.3, the use of std::vector on Line 17 to declare the variables *v* and *w* is an example of a known-unknown PhantomTypename. The fourth schema element, an unknown-unknown PhantomTypename, is used to hold type declarations that were defined at class-scope, but their name is inferred from its use. These declarations are generally nested classes, or typedefs that appear in the body of a class. In Figure 6.3, the variable declaration on Line 19, std::vector<int>::iterator is an example of an unknown-unknown PhantomTypename. We know the name std::vector is a PhantomTypename, and from the use of iterator as the type in a variable declaration we know it must name a type. The schema element in the fifth row of the table, PhantomType, is used to hold the return type of phantom functions, and any expression that involves phantom types and yields a value. This element is similar to PhantomTypename reference, except that it is used when we encounter types without names. This schema element is used in the expression-level analysis of user-code that gets mixed with declaration involving phantom

types. For example, in Figure 6.3 on Line 21, there is a call to the max_size method of the *w* object. Because the type of *w* is a known-unknown PhantomTypename, the function max_size is a unknown-unknown PhantomReference, and therefore its return type is an PhantomType.

Another addition to the schema, that is not listed in the table, is a new node, SizeofExpr, which is required to handle the sizeof operator. In the absence of phantom types, this is always immediately resolvable into a number. But if sizeof is evaluated on a phantom type, then this new ASG node type is needed in order to store the expression in the graph.

	ASG Schema Element	Cognition Level	Description	Example Use
1	PhantomReference	Known-unknown	Reference to a variable or function in a library header	swap on Line 23
2	PhantomReference	Unknown-unknown	Reference to a member of a phantom class	resize on Line 21
3	PhantomTypename	Known-unknown	Reference to a class\union\struct or type alias in namespace-scope	vector on Line 17
4	PhantomTypename	Unknown-unknown	Reference to a class\union\struct or type alias in class-scope	iterator on Line 19
5	PhantomType	Unknown-unknown	Return type of a phantom function, or the type of an expression involving phantom types	Value of max_size on Line 21

Table 6.1: New schema elements for phantom ASGs

6.5 Two Illustrative ASG Examples

In Chapter 6.3, we introduced an algorithm for building an abstract semantic graph. The majority of this algorithm is devoted to reconstructing a parse tree, evaluated with a top-down SAX parser, into a structure that is evaluated in a bottom-up fashion, which is more familiar to parsing languages defined in BNF. The actual construction of the ASG is handled by the semantic actions executed on Line 45 of Figure 6.2. Because there are 488 right-hand sides in the grammar definition, a complete explanation of the grammar is beyond the scope of this thesis. But we do present two code segments, and describe the semantic actions required to construct an ASG. First, we show how to build an ASG for binary expressions when user defined data-structures, overloaded methods, function templates, and phantom types are involved. And, secondly, we describe how an ASG is built when evaluating an expression that takes advantage of C++'s template metaprogramming feature to calculate a Fibonacci sequence at compile-time.

6.5.0.6 Evaluating Binary Expressions

Figure 6.4 lists the semantic action involved in a bottom-up parse of a binary expression. This algorithm accepts three arguments: a string representing of the operator, *op*, the left-hand side operand, *lhs*, and the right operand, *rhs*; which produces a new ASG node. On Lines 2 and 3, the algorithm acquires the types of the left- and right-hand side operands, and then removes any type qualifiers, using the getType and stripQualifier algorithms, respecively, and stores the results in the variables *ltype* and *rtype*. Type qualifiers may include const, or volatile, and are not, for the most part, necessary in resolving binary expressions. Next, on Lines 4 and 5, the algorithm checks if either of the types of the operands are phantom types. If so, then there is no other semantic information that may be obtained; therefore, a new binaryExpr object is created, with *op* as its operator, and *lhs*

```
1   algorithm CreateBinaryExpr(op, lhs, rhs):
2       ltype ← stripQualifiers(getType(lhs))
3       rtype ← stripQualifiers(getType(rhs))
4       if isPhantomType(ltype) or isPhantomType(rtype):
5           return binaryExpr(op, lhs, rhs)
6       if isPrimativeType(ltype) and isPrimativeType(rtype):
7           if isContantExpr(lhs) and isContantExpr(rhs):
8               return reduceConstant(op, lhs, rhs)
9           else return binaryExpr(op, lhs, rhs)
10
11      opfunc ← "operator " ∘ op
12      candidates ← lookupFunctions(getCurrentNamespace(), opfunc)
13      if not isPrimativeType(ltype):
14          extend candidates with lookupMembers(ltype, opfunc)
15          extend candidates with lookupFunctions(getNamespace(ltype), opfunc)
16      if not isPrimativeType(rtype):
17          extend candidates with lookupFunctions(getNamespace(rtype), opfunc)
18      f ← resolveOverload(candidates, getType(lhs), getType(rhs))
19      if f is not null:
20          if isMemberOf(f, ltype):
21              ma ← createMemberAccessExpr(lhs, f)
22              return createFunctionCallExpr(ma, rhs)
23          else:
24              return createFunctionCallExpr(f, lhs, rhs)
25
26      if not isPrimativeType(ltype):
27          foreach c in getTypeConversions(ltype):
28              if typesConvertable(getType(c), rtype):
29                  ma ← createMemberAccessExpr(lhs, c)
30                  cast ← createFunctionCallExpr(ma)
31                  return binaryExpr(op, cast, rhs)
32      if not isPrimativeType(rtype):
33          foreach c in getTypeConversions(rtype):
34              if typesConvertable(ltype, getType(c)):
35                  ma ← createMemberAccessExpr(rhs, c)
36                  cast ← createFunctionCallExpr(ma)
37                  return binaryExpr(op, lhs, cast)
38
39      foreach p in getBaseClasses(ltype):
40          if isPhantomType(p):
41              return binaryExpr(op, lhs, rhs)
42      foreach p in getBaseClasses(rtype):
43          if isPhantomType(p):
44              return binaryExpr(op, lhs, rhs)
45      return null
```

Figure 6.4: Algorithm for evaluating the semantics of a binary expression.

and *rhs* as its operands. On Line 6, the algorithm checks if the unqualified types of both operands are primitive, or language-defined, types. If so, the algorithm further checks, on Line 7, if both of the operands are constant expressions (that is, known values). If both operands are constants, then, on Line 8, the algorithm reduces both operands to a single value, using the operator, *op*. While this step does provide more exact information in the semantic graph than a parse tree would, it is also crucial to properly evaluate templates, as we demonstrate see in Chapter 6.5.0.7. Line 9 of the algorithm is reached, if both operands' unqualified types are primitive, but the operands themselves are non-constant. On this line,

the algorithm creates a binaryExpr with *op* as its operator, and *lhs* and *rhs* as its operands. Any implicit casting to make the operands' types coherent with each other, or the operator, are handled in the creation of the binaryExpr object. These implicit casts may include convert an int to a double, in the case of 3.14/2; or converting the operand to a bool if *op* is a logical operator, such as in the expression ptr && ptr->next.

If the algorithm reaches Line 11, then one of the operands is a user-defined type, and the appropriate overloaded operator function must be located. On Line 11, a string is created with the concatenation of the literal "operator " and the string representation of the operator *op*, and stored in the variable *opfunc*. On Line 12, a new set is created and stored in the variable *candidates*. This set will hold all the references to the overloaded operator functions with the correct name in the correct scope. The set, *candidates*, is then initialized to all references to functions named *opfunc* that are defined in the namespace scope that the binary expression appeared in. On Line 13, the algorithm checks if the unqualified type of the left operand, *ltype*, is a user-defined type. If so, on Line 14, *candidates* is extended with all references to *opfunc* found as member functions of *ltype*. And also, on Line 15, all references to functions named *opfunc* found in the same namespace that *ltype* was defined in is added to the *candidates* set. This lookup facilitates Koening, or argument-dependent, lookup. Similarly, on Lines 16 and 17, if the unqualified type of the right-hand operand, *rtype*, is a user-defined type, then all reference to functions named *opfunc* found in the same namespace that *rtype* was defined in is added to the *candidates* set. On Line 18, the resolveOverload algorithm is invoked. This algorithm takes a set of function references and a list of the types of the actual parameters, and returns the reference to the function that most closely matches the arguments, if any. In this instance, the algorithm passes in *candidates* as the set of function references, and the complete types, not the unqualified types, of the binary expression's operands, *lhs* and *rhs*. The closest match, if found, is stored in the variable *f*. On Line 19, the algorithm checks if resolveOverload found a

function that is an appropriate match, by examining the variable *f* for nullness. On Line 20, if *f* is non-null, and *f* is a member function of the unqualified type of *lhs*, then the algorithm creates a new memberAccessExpr ASG node. A memberAccessExpr is used to store expressions using the dot notation in the ASG. On Line 21, the memberAccessExpr node is created, with *lhs* as the object, and *f* as its member, and then stored in the variable *ma*. On the next line, Line 22, a functionCallExpr node is created, with *ma* set as the callee function and *rhs* as its sole argument. On Line 24, if *f* is a valid function reference, but it is not a member of the *lhs* object, then it is standalone function in either the namespace of *ltype* or *rtype*. A new functionCallExpr node is created, with *f* as the callee function, and with *lhs* and *rhs* as its two arguments.

Lines 26 through 37 of the algorithm checks to see if either of the operands have defined conversion functions. A conversion function is a function, defined in a class, that allows that class to be implicitly used as another type. Consider the following code segment:

```
1    class S {
2        operator int() { ... }
3    }
4    void f(int);
5    S s;
6    f(s);
```

The method definition on Line 2, allows the variable *s* of type *S* to be passed as a parameter to the function *f* which only accepts a single parameters of type ***int***.

On Line 26 of Figure 6.4, the algorithm checks if the unqualified type of the left-hand operand is a user-defined type. If so, on Line 27, the algorithm starts iterating all the of conversion functions defined in *ltype*. On Line 28, the algorithm checks if the conversion function, *c* and the unqualified type of the right-hand operand, *rtype*, are convertable. If so, then on Line 29, a new memberAccessExpr node is created with *lhs* as the object and the conversion function, *c*, as the member being accessed. This node is stored in the variable

ma. On Line 30, a new functionCallExpr node is created, with *ma* as the callee function, and because conversion functions are parameterless, no additional arguments are passed in. The new functionCallExpr node is stored in the variable *cast*. Finally, a new binaryExpr node is created, with *op* as the operator, and *cast* and *rhs* as the left- and right-hand side operands. Similarly, on Lines 32 through 37, an appropriate conversion function for the right-hand operand is sought for. If a conversion function is found, then a new binaryExpr node is created with *lhs* as the left-hand operand, and a functionCallExpr node as the right-hand operand.

If no exit point has been reached, then the algorithm reaches line 38; so far we know that: (1) neither left- nor right-hand side operand is a phantom type, (2) both operands are not constants, (3) at least one operand is a user- defined type, (4) there exists no overloaded operator in the current namespace scope, the scope of the left-hand operand's class, the scope of the left-hand operand's namespace, or the scope of the right-hand operand's namespace, and (5) neither operand has a conversion function defined that make the two types compatible. Since all avenues have been explored, as one final check, on Line 39, the algorithm iterates all classes that the left-hand operand's type inherit from. If any of these parent classes, referenced by the variable *p*, are a phantom type, then the algorithm assumes that an appropriate overloaded operator or conversion function was never found because it was defined in a system header file that was never fully parsed. If this is the case, then, on Line 41, a new binaryExpr node is created with *op* as its operator, and *lhs* and *rhs* as its operands. Analogous to the action taken on Lines 4 and 5, no further semantic information can be obtained in presence of phantom types. If no phantom types were found in the left-hand operand's type hierarchy, then the same search is mirrored in the right-hand operand, on Lines 42 through 44.

We now present a source code example that exercises the CreateBinaryExpr algorithm previously described. We will trace the example twice: first using the phantom

parser, which only uses minimal information from system header files, and secondly, using the full parser, which treats compiler vendor code and user code equally. After each trace is evaluated, we will present an ASG representation of the expression. A simple "Hello World" example is presented here:

```
1    #include <iostream>
2    int main() {
3        std::cout ≪ "Hello, World!" ≪ std::endl;
4    }
```

The expression on Line 3 is composed of two binary expressions, with a total of three operands. The first operand, std::cout is defined in the ostream header file and is of type std::basic_ostream<**char**, std::char_traits<**char**>>. A partial definition of the std::basic_ostream class template is provided in Figure 6.1. The second operand, the string literal "Hello, World!", is of type const char $*$. The third, and final, operand is std::endl, which is a function template defined in the ostream header file. This function template is defined on Lines 36 through 39 in Figure 6.1.

First, we evaluate Line 3 from the above program using our phantom parser. The first invocation of the CreateBinaryExpr algorithm has its parameters set to: *op* is $\ll$, *lhs* is std::cout, and *rhs* is "Hello, World!". Since we are using our phantom parser, std::cout is simply a PhantomReference; we know the variable was defined in a system header, but we don't have any type information. Since both *lhs* and *rhs* are already at their most basic type, the calls to stripQualifiers have no effect. On Line 4, the algorithm checks if either arguments' type was a phantom type. In this case, the left-hand side is. An ASG node for the operator is created, one of type LShiftExpr, with its left- and right-hand side operands set to *lhs* and *rhs*. The ASG type of this node cannot be determined due to the presence of phantom types, so the type is a PhantomType.

The second invocation of the CreateBinaryExpr occurs with: *op* is set to $\ll$, *lhs* is the newly created LShiftExpr object, and *rhs* is std::endl. Just as with std::cout, only

the name std::endl was parsed from the ostream header file, so its type is PhantomReference. This time, both operands are phantom types, so on Line 5, a new LShiftExpr node is created. The new node has its left and right operands set to the previously created LShiftExpr and std::endl. The type of the expression is indeterminate, so it is set to PhantomType. The ASG for the "Hello World" program, using the phantom parser, is shown in Figure 6.5.

We will now retrace the algorithm in Figure 6.4, using the above "Hello World" program, this time using the full parser. On the first invocation of CreateBinaryExpr, *lhs* is std::cout and *rhs* is the string literal "Hello, World!". As neither operands' type is a phantom type, and *lhs* is not a primitive type, we can skip to Line 11 to collect all possible function overloads for operator≪. On Line 14, we collect seventeen operator≪ functions from the std::basic_ostream class; on Line 15, we collect five more from the namespace std. Lines 16 and 17 are skipped, because *rhs* is a primitive type and, therefore, has no namespace. On Line 18, from the twenty-two candidate functions, the closest match is found to be this function in the std namespace:

```
33      template<typename _CharT, typename _Traits>
        basic_ostream<_CharT,  _Traits>  &operator≪(basic_ostream<_CharT,  _Traits>
34    &__out,
35                                              const _CharT *__s);
```

A reference to this function is stored in the variable *f*. *f* is not a member of the std::basic_ostream class, so on Line 24, a new FunctionCallExpr ASG node is created, with *f* as the callee function, and *lhs* and *rhs* as its operands. The type of the node is the return type of *f*, so this node has type std::basic_ostream<**char**, std::char_traits<**char**>>&. In the second invocation of CreateBinaryExpr this newly created FunctionCallExpr node is the *lhs*, and a name reference to the function template std::endl is its *rhs*. On Line 2 of the algorithm, the reference-type qualifier (the &) is stripped from the type of *lhs*, and stored in the variable *ltype*. Neither *lhs* or *rhs* is a phantom type or primitive type, so we can go

to Line 11 to collect the function overload candidates for operator≪. On lines 13 through 15, the seventeen operator≪ functions from the std::basic_ostream class or added to the *candidates* set. On Line 15, we add the five instances of the operator≪ function defined the namespace std. On Lines 16 and 17, we collect instances of operator≪ function that are defined in *rhs*'s namespace; as this is again the std namespace, we already have them. On Line 18, the closest match for operator≪ is found to be the following method in the std::basic_ostream class:

```
19    inline __ostream_type&
20      operator≪(__ostream_type& (*__pf)(__ostream_type&));
```

A reference to this method is stored in the variable *f*. Since *f* is a member of of the std::basic_ostream class, on Line 29, we create a MemberAccessExpr ASG node. This node is used to represent a member access using C++'s dot notation. In this case, the FunctionCallExpr created from the first invocation of CreateBinaryExpr is the object, and *f*, the correct operator≪ function, is the member being accessed. We then create a new FunctionCallExpr ASG node, with the MemeberAccessExpr ASG node as the callee function, and std::endl as its only argument. Although not shown in the algorithm, the creation the of FunctionCallExpr causes the reference to the function template std::endl to have its template arguments fully resolved. Instead of inserting the name reference to std::endl in the ASG, the function std::endl<**char**, std::char_traits<**char**>>(std::basic_ostream<**char**, std::char_traits<**char**>>&) is instantiated, and it the function pointer to that instantiation that is used as the argument to the FunctionCallExpr. The ASG for the "Hello World" program, using the full parser, is shown in Figure 6.6.

6.5.0.7 Template Metaprogramming

In this section we provide a second look the semantic actions required to build an ASG in Hylian. We present a algorithm for evaluating template declarations. Unlike the

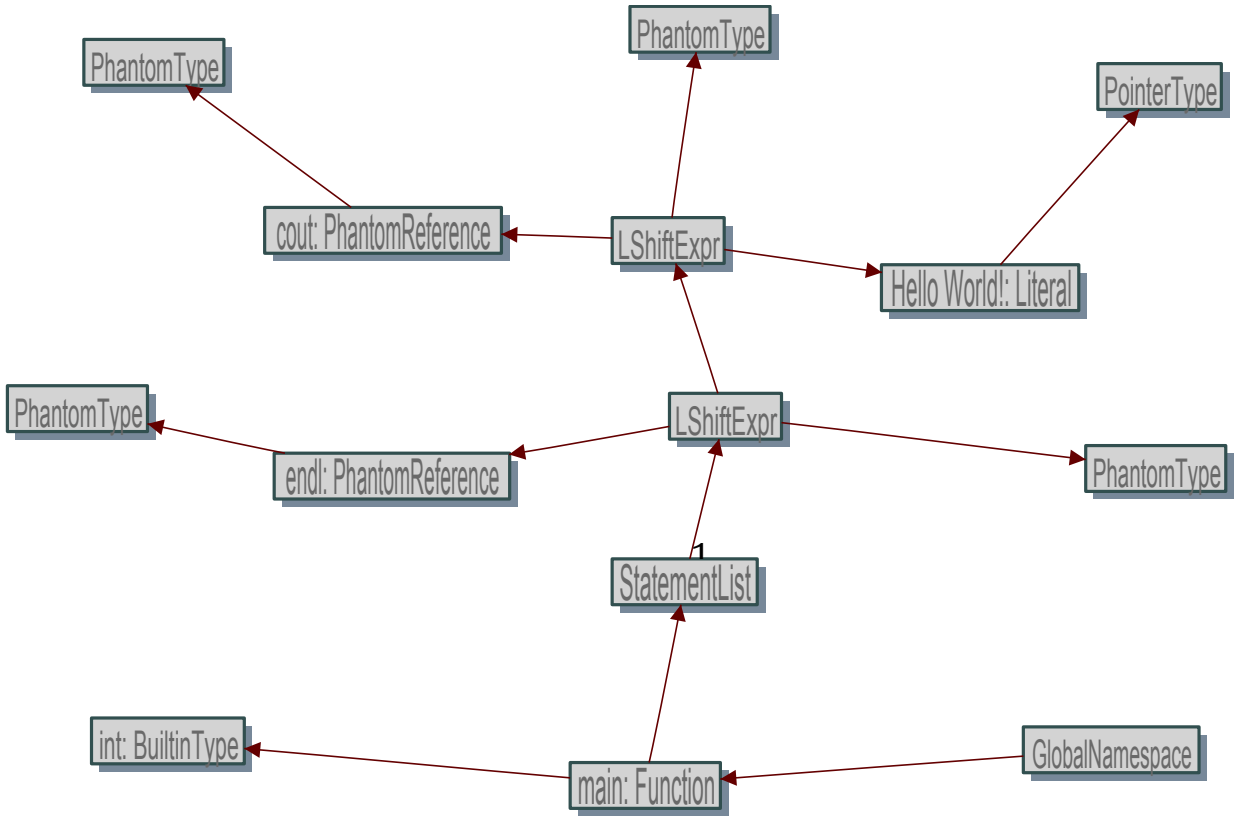


Figure 6.5: Hello World ASG using phantom parser

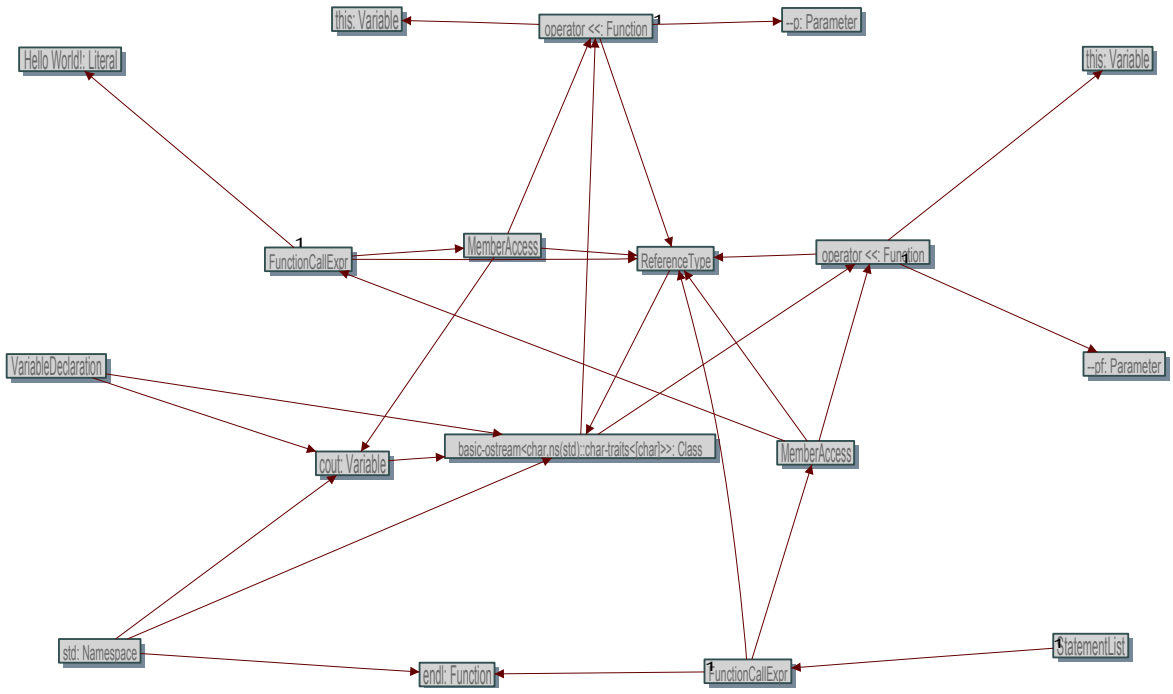


Figure 6.6: Hello World ASG using full parser

previous example, we do not provide a more detail view of this algorithm in pseudocode because handling template declarations affects most of the C++ grammar. Instead we present the algorithm in a higher level view here:

1. Store parse trees for templates declarations in template dictionary
2. If a semantic action requires a member of a *template-id* then
3. If a matching template specialization exists in symbol table then
4. Return symbol table entry for fully-formed class
5. Else
6. Build a new parser instance
7. Look for general definition of template declaration in *parsetreeBuffer*

8. Push provided template arguments onto new parser's scope stack

9. Evaluate any default template arguments for template declaration, push results onto new parser's scope stack

10. Set the new parser's input to the parse found in *parsetreeBuffer* on Step 7 or 8

11. Execute the parse, when complete the new class template instantiated will appear as a template specialization in the symbol table

12. Return symbol table entry for newly instantiated class template

As we step through he above algorithm, we refer to a *template-id* several times. A *template-id* is a grammar entity that has both a template definition's name, and a list of its template arguments. It is defined in the C++ grammar in the following way:

$$template\_id \rightarrow template\_name < template\_argument\_list >$$

Step 1 of the above algorithm was covered previously in Chapter 6.3; as the source code is being parsed in a bottom-up manner, only enough information needed to identify a template declaration is parsed, and the entire parse tree for the template declaration is stored in the *parsetreeBuffer*. Step 2 occurs in many of the semantic actions. In C++, when a *template-id* is encountered we may not instantiate it until it is invoked (in the case of function templates), one of its member's is accessed (in the case of class templates), or its size is needed (by way of the sizeof operator, or a variable of that type is defined). Consider the two following typedef declarations:

```
1  typedef std::vector<int> ContainerType;
2  typedef std::vector<int>::const_iterator IteratorType;
```

The definition of *ContainerType* only requires the *template-id*; nothing specific about the vector is needed, so the vector class template is not instantiated. However, the

second declaration requests the vector's *const_iterator* member, so the vector does have to be instantiated in order properly evaluate this typedef declaration.

For Step 3, the parser's symbol table is queried for an existing template specialization for the provided *template-id*. A template specialization is a class template that has been fully defined in the code, or a template declaration that already been instantiated because of a previous use. On Step 4, the template specialization found in the symbol table is returned to semantic action handler that requested it. If an exact match for the given *template-id* was not found in the symbol, then in Step 6, a new instance of the parser is created. This parser will have its own input stream and scope stack, but shares the same symbol table and parse tree buffer as the current parser handling semantic actions. In Step 7, the parse tree for the template declaration is retrieved from the *parsetreeBuffer*. In Step 8, all template arguments provided by the *template-id* are pushed onto the scope stack in the new parser instance. In Step 9, any default template arguments, that were not provided in the *template-id*, are evaluated and pushed onto the scope stack in the new parser instance. For Step 10, the parse tree found in the *parsetreeBuffer* is fed into the new parser's input stream. In Step 11, the parser is executed, when the parse is complete the new class is instantiated and, therefore, will appear as a template specialization is the parsers' shared symbol table. Finally, on Step 12, a template specialization for the *template-id* is guaranteed to be in the symbol table; its is retrieved and returned to to semantic action handler that requested it.

To exercise the template instantiation algorithm, given above, we are going to employ C++'s template metaprogramming technique to calculate the Fibonacci sequence. The familiar Fibonacci sequence is an integer sequence defined by the following recursive relationship:

$$F_0 = 0 \tag{6.1}$$

$$F_1 = 1 \tag{6.2}$$

$$F_n = F_{n-1} + F_{n-2} \tag{6.3}$$

This sequence is defined in C++ as metaprogram with following source code.

```
1   template<int N>
2   struct Fib {
3       enum { value = Fib<N-1>::value + Fib<N-2>::value };
4   };
5
6   template<>
7   struct Fib<1> {
8       enum { value = 1 };
9   };
10
11  template<>
12  struct Fib<0> {
13      enum { value = 0 };
14  };
15
16  int main() {
17      int f = Fib<4>::value;
18  }
```

On Lines 1 through 4, the template class *Fib* is defined. *Fib* has a single constant-template parameter, N, and defines an enumeration, *value*, set to the sum of *Fib<N-1>* and *Fib<N-2>*'s *value* enumeration. On Lines 6 through 9, a specialization for the class template *Fib* is defined for the case $N = 1$. Instead of calculating a value for *value*, it is assigned 1. Similarly, on Lines 11 through 14, a specialization for the class template *Fib* is defined for the case $N = 0$; in this specialization, *value* is set to 0.

In the main function, on line 16 through 18, the local variable f is initialized with the *value* enumeration of the class Fib<4>; this class is the class template *Fib*, instantiated with its template parameter N set to 4. Following the algorithm we presented at the be-

ginning of this section, the parse tree for *Fib* is saved in the *parsetreeBuffer*, and template specializations for Fib<1> and Fib<0> exist in the symbol table. To evaluate the expression Fib<4>::value, the symbol table is queried for the template specialization for class Fib<4>. Since it does not exists, *Fib* must first be instantiated with the template argument $N = 4$. A new parser instance is created, and the parse tree for *Fib* is parsed with $N = 4$. While parsing Fib<4>, to evaluate *value* requires the expression Fib<N-1>::value, or Fib<3>::value, to be resolved. This expression means we have to restart the algorithm with the *template-id* of Fib<3>. Fib<3> is not in symbol table, so another parser instance is created, and the parse tree for *Fib* is again parsed, this time with the template argument $N = 3$. As with Fib<4>, evaluating the *value* enumeration of Fib<3> requires the expression Fib<N-1>::value, or Fib<2>::value, to be resolved. Fib<2> is not in the symbol table either, so we create a fourth instance of the parser, set its input stream to the parse tree for *Fib*, and parse with the template argument $N = 2$.

Parsing the template class *Fib* with its template argument N set to 2, requires the evaluation of the expression Fib<1>::value. Fib<1> is a class in the symbol table, so no further instantiation is required for this expression. The value of Fib<1>::value is defined, in the above source code on Line 8, as 1. Further parsing the value enumeration for the instantiation of Fib<2>, we encounter the expression Fib<N-2>::value, which is equivalent to Fib<0>::value. The *template-id* Fib<0> is in the symbol table as well, and its *value* enumeration is set to 0, so we can completely resolve the expression Fib<N-1>::value + Fib<N-2>::value in Fib<2>, which is $1 + 0$, or 1. This class is now instantiated, so the class Fib<2> is inserted into the symbol table.

With the parse of Fib<2> complete, we jump back to the parse of Fib<3>. Its initialization of the *value* enumeration requires the *value* enumeration of the class Fib<2>. It is in the symbol table, so we can use the value Fib<2>::value, which is 1. Further evaluation of Fib<3>::value needs Fib<N-2>::value, or Fib<1>::value. The class

Fib<1> is in the symbol table, so we lookup its value of the *value* enumeration and get a 1. Fib<3>::value can now be resolved to $1 + 1$, or 2; the class Fib<3> is instantiated and inserted into the symbol table.

Similarly, now that Fib<3> is instantiated, we go back to the evaluation of Fib<4>::value. The first piece of the expression is Fib<3>::value, we have from the previous instantiation of Fib<3>, which is 2. The second piece of the expression, Fib<2>::value, requires the instantiation of Fib<2>, which we have stored in the symbol table from the instantiation of Fib<3>. Fib<2>::value is equal to 1, so can resolve Fib<4>::value, which has a value of $2 + 1$, or 3.

The ASG for this program is shown in Figure 6.7. This ASG shows the main function, with a single VariableDeclaration node. The node is for the declaration of the variable *f*, which is shown being initialized to the value of 3. Through template metaprogramming, only semantics of the program matter; the template class *Fib* and its five instantiations are not referenced from the main function at all.

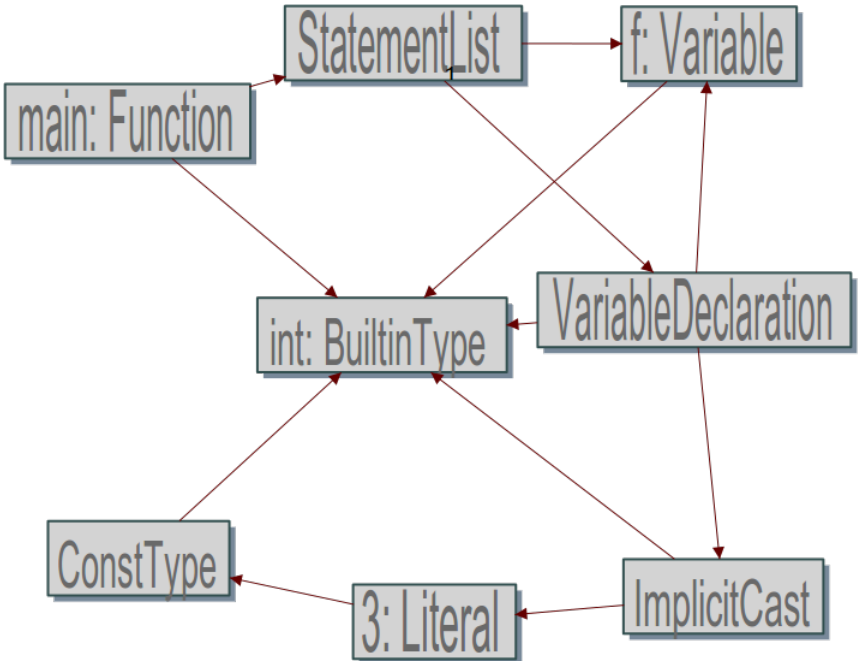


Figure 6.7: ASG for the Fibonacci template metaprogram

6.6 The Technique For Generating C++ Code From an ASG

In Chapter 4, we introduced all the modules that comprise the Hylian system. The rectangles numbered VI and VII in the overview summary diagram in Figure 4.1 both rely on the ability to convert an ASG into C++ code. In the "ASG Verification & Validation" stage, our approach to ASG validation entails generation of C++ source code from an unmodified, linked, ASG to ensure that the resulting code exhibits the same behaviour as the original program. And second, in the "ASG Transformation" stage, to perform dynamic analysis of the source code, we must perform transformations on the generated ASG, and then generate source code from the transformed ASG. In this section, we present an algorithm for generating C++ code from an ASG and walk through an example to demostrate how the algorithm works.

```
1   algorithm GenerateCodeFromASG(asg):
2       todo ← empty stack
3       depGraph ← empty graph
4       global ← lookupChildNodeByType(asg, "asg.gxl#GlobalNamespace")
5       main ← lookupChildNodeByName(global, "main")
6       push main onto todo
7       while todo not empty:
8           node ← top of todo
9           pop todo
10          deps ← generateCodeFromASGNode(node)
11          foreach dep in deps:
12              if dep not in depGraph:
13                  push dep onto todo
14              if dep in depGraph and nodeHasType(dep, "asg.gxl#Class"):
15                  decl ← generateCodeForClassDecl(dep)
16                  addDirectedEdge(depGraph, node, decl)
17              else:
18                  addDirectedEdge(depGraph, node, dep)
```

Figure 6.8: Algorithm for generating C++ source code from an ASG

The algorithm listed in Figure 6.8 traverses an ASG, creating C++ source code for high-level structures and building a dependency graph to ensure the code is written in the proper order. On Line 2, an empty stack is initialized, named *todo*, which will be used to maintain a collection of high-level structures whose source code has not yet been generated. On Line 3, an empty graph is initialized, named *depGraph*, which will be used to

enfore the dependency relationships among the high-level structures. On Lines 4 and 5, the algorithm searches for the entry point into the application. First, the ASG node of type "GlobalNamespace", representing the global namespace, is found. The schema requires that there is one, and only one, node of this type. Then, the algorithm searches the global namespace for the ASG node with an attribute named "name" set to the value of "main". On Line 6, the handle to the main function is pushed onto the *todo* stack, since it will be the first high-level structure for which code is generated. On Line 7 the main loop of this algorithm starts, the remainder of the algorithm will be repeated for every high-level structure found until the *todo* stack is exhausted.

On Lines 8 and 9, a reference to the high-level structure at the top of the *todo* stack is stored in the variable *node*, and then popped from the stack. On Line 10, the source code for the high-level structure referenced by *node* is generated. These high-level structures are defined in namespace-scope and include language constructs such as class definitions, function definitions, or variable declarations. As source code for this structure is generated, a list of external dependencies is maintained and returned to the callsite. An external dependency may be another function that is called, or a variable that is used, but not defined, inside a function. Further specifics for generating the source code will be uncovered later in this section. Next, on Lines 11 through 18, the algorithm examines the list of dependencies returned by the code generator. On Lines 12 and 13, the dependency is checked to see if it has not yet appeared in *depGraph*. If not, then it is added to the *todo* stack, so that it's code will be generated on a later iteration of the main loop. On Line 14, the dependency, *dep*, is checked to see if it is already in *depGraph*, and may also be a recursive entity (such as a class or function). If so, on Lines 15 and 16, a declaration for the high-level structure is generated, since its definition has been created previously, and added as a dependency for the source code generated on Line 10 to *depGraph*. Finally on Line 18, if a declaration was not inserted into the dependency graph, then the definition is

added.

```
1   algorithm GenerateCodeForType(in node, out code, inout deps):
2     if nodeHasType(node, "asg.gxl#BuiltinType"):
3       code ← getNodeAttribute(node, "name")
4     else if nodeHasType(node, "asg.gxl#ReferenceType"):
5       basetype ← getNodeFromEdgeType(node, "asg.gxl#BaseType")
6       code ← GenerateCodeForType(basetype, deps) ∘ " &"
7     else if nodeHasType(node, "asg.gxl#PointerType"):
8       basetype ← getNodeFromEdgeType(node, "asg.gxl#BaseType")
9       code ← GenerateCodeForType(basetype, deps) ∘ " *"
10    else if nodeHasType(node, "asg.gxl#ConstType"):
11      basetype ← getNodeFromEdgeType(node, "asg.gxl#BaseType")
12      code ← "const " ∘ GenerateCodeForType(basetype, deps)
13    else if nodeHasType(node, "asg.gxl#ArrayType"):
14      basetype ← getNodeFromEdgeType(node, "asg.gxl#BaseType")
15      code ← GenerateCodeForType(basetype, deps)
16    else if nodeHasType(node, "asg.gxl#Class"):
17      code ← getNodeAttribute(node, "name")
18      append node to deps
```

Figure 6.9: Algorithm for generating C++ source code from an ASG Node representing a C++ type

The algorithm listed in Figure 6.9, GenerateCodeFromType, accepts an ASG node referencing a type expression and returns both a string representation of its C++ code, in the parameter *code*, and a list of dependencies defined outside the type expression, in the parameter *deps*. This algorithm is used in generating code for constructs such as variable declarations, parameter declarations, and cast expressions. On Line 2, the type of the ASG node is checked to see if it is a "BuiltinType". An ASG node of type "BuiltinType" has two attributes, *name* and *size*. These are types that are defined by the language grammar, such as **int** or **float**, so the code representing this type expression is simply the *name* attribute. On Lines 4 through 6, code for representing nodes of type "ReferenceType" is generated. First, the outgoing edge from *node*, of type "BaseType", is found. The ASG node at the head of that edge, which represents the base type of this type expression, is stored in the variable *basetype*. On Line 6, the code of this type expression is set to the code of *basetype* concatenated with symbol &. Lines 7 through 9 generate code for type epxression nodes of type "PointerType", and is very similar for the "ReferenceType" node. The base type of the type expression is found, by following the "BaseType" edge, and stored in the *basetype*

variable. The code for *basetype* is generated, appended with the symbol $*$, and stored in the variable *code*. Similarly, on Lines 10 through 12, when generating the code for type expressions of type "ConstType", the algorithm follows the outgoing edge of type "BaseType", and stores the resulting node in the *basetype* variable. The code for this type expression is the keyword **const** concatinated with the C++ code that represents the node referenced by *basetype*. On Lines 13 through 15, code generation for nodes of type "ArrayType" is partially handled. Because generating the code completely for an array declaration requires tokens after the declarator identifier (such as a variable name, or a typedef alias), only the code for the base type of the array is generated here. The code for the array bounds will have to be handled specially later on. Finally, on Lines 16 thrugh 18, the algorithm generates code for handling type expressions of type class. Because this algorithm is used for type expressions, only the class's name attribute is needed to represent it in code. But to be sure that the definition for the class, if it exists, appears earlier in the code, the algorithm adds a reference to the class in the dependency list, *deps*.

```
1   algorithm GenerateCodeForVariableDecl(in node, out code, inout deps):
2       type ← getNodeFromEdgeType(type, "asg.gxl#Type")
3       code ← GetCodeForType(type, deps)
4       if getNodeAttribute(node, "name") is not null:
5           code ← getNodeAttribute(node, "name")
6       if nodeHasType(type, "asg.gxl#ArrayType"):
7           size ← getNodeFromEdgeType(type, "ArraySize")
8           if size is not null:
9               code ← "[" ∘ GetCodeForExpression(size, deps) ∘ "]"
10          else: code ← "[ ]"
11      append type to deps
```

Figure 6.10: Algorithm for generating C++ source code from an ASG Node representing a variable declaration

---

The algorithm shown in Figure 6.10 generates code for ASG nodes of type "Variable" or "Parameter", which reference variable and parameter declarations. As with the previous algorithm, GenerateCodeForVariableDecl returns both a string representation of its C++ code, in the parameter *code*, and a list of dependencies defined outside the declaration, in the parameter *deps*. On Line 2, the node reference the declaration's type is found

by following the outgoing edge of type "Type", and stored in the variable *type*. On Line 3, the code *type* is generated, by using the previously described algorithm, GenerateCode-ForType, and stored in the variable *code*. On Line 4, the algorithm checks to see if the ASG node for the declaration has a *name* attribute. Parameters in C++ are allowed to be abstract, or nameless, so a name may not exist; if there is a name for the node, then on Line 5, the name is appended to the variable *code* [1]. On Line 6, the algorithm checks to see if *type* has an ASG type of "ArrayType", meaning the declaration is for an array. If so, on Line 7, the ASG node representing the array's size declaration is look for, by following the outgoing edge of type "ArraySize", and is stored in the variable *size*. On Line 8, the algorithm checks if the *size* variable is non-null. If so, then on Line 9, the code representing the *size* expression is generated, surrounded by square-brackets, and appended to the end of the *code* variable. If no "ArraySize" edge was found coming from the declaration node, then the code representing an abtract array, that is empty square-brackets, is appended to the *code* variable. Finally, on Line 11, the type of the declaration is appended to the dependency list to ensure that the type is defined in the source code before this declaration occurs.

The algorithm shown in Figure 6.11 generates code for ASG nodes of type "Function", which references function definitions. On Line 2, the *code* variable is initialized to the empty string. On Line 3, the algorithm checks if the linkage attribute of the ASG node is non-null. If not, then on Line 4, the keyword **extern** along with the linkage attribute value is appended to the *code* variable. On Line 5, the algorithm checks if the is_static attribute is set to **true**. If so, then on Line 6, the algorithm appends the keyword **static** to the variable *code*. On Line 7, the algorithm checks if the is_inline attribute is set to **true**. If so, then on Line 8, the algorithm appends the keyword **inline** to the variable *code*. On

[1] Although we show code for writing tokens, adding code to show where whitespace is needed would be superfluous.

```
1   algorithm GenerateCodeForFunction(in node, out code, inout deps):
2       code ← empty string
3       if getNodeAttribute(node, "linkage") is not null:
4           code ← "extern " ∘ getNodeAttribute(node, "linkage")
5       if getNodeAttribute(node, "is_static") is true:
6           code ← " static "
7       if getNodeAttribute(node, "is_inline") is true:
8           code ← " inline "
9       retType ← getNodeFromEdgeType(node, "asg.gxl#ReturnType")
10      code ← GetCodeForType(retType, deps)
11      code ← getQualifiedName(node) ∘ " ("
12      parameters ← getOutgoingEdgesOfType(node, "asg.gxl#ParameterDeclaration")
13      foreach p in parameters:
14          if getEdgeFromOrder(p) > 0:
15              code ← ","
16          param ← getEdgeHead(p)
17          code ← GetCodeForVariableDecl(param, deps)
18      if getNodeAttribute(node, "variable_parameters") is true:
19          code ← ", ..."
20      code ← ")"
21      if getNodeAttribute(node, "is_const") is true:
22          code ← " const "
23      code ← "{"
24      body ← getNodeFromEdgeType(node, "asg.gxl#FunctionBody")
25      stmts ← getOutgoingEdgesOfType(body, "asg.gxl#Statement")
26      foreach s in stmts:
27          stmt ← getEdgeHead(s)
28          code ← GetCodeForStatement(stmt, deps) ∘ ";"
29      code ← "}"
30      append retType to deps
```

Figure 6.11: Algorithm for generating C++ source code from an ASG Node representing a function

Line 9, the algorithm looks up the node at the head of the outgoing edge of type "Return-Type", and stores it in the variable *retType*. Using the GetCodeForType algorithm, the code for the variable *retType* is generated and appeneded to the *code* variable, on Line 10. On Line 11, the algorithm evaluates the qualified name of the function, the function's name prepended with its complete namespace specifier, and appends it to the *code* variable. An open-parenthesis is also added to *code* to signify the start of the parameter declaration list. On Line 12, the algorithm collects all outgoing edges from the function ASG node, of type "ParameterDeclaration", storing the result in the variable *parameters*. Lines 13 through 17 iterate the edges making up the parameter declaration list, *parameters*, using the loop control variable *p*. On Lines 14 and 15, if the "from-order" of the edge is greater than zero, that is, not the first edge, then a comma is appended to the end of *code*. On Line 16, the ASG node at the head of the *p* edge, is stored in the variable *param*. This ASG node is of type

"Parameter" and references a parameter declaration. The code for *param* is generated and appended to *code*, on Line 17. On Lines 18 and 19, after examining the parameter declaration list, the algorithm checks the function declaration node for the "variable_parameters" attribute. If set to **true**, a comma and an ellipsis is appended to the *code* variable. Finally, on Line 20, the closing parenthesis is added, concluding the parameter declaration. On Line 21, the algorithm checks if the is_const attribute is set to **true**, indicating that this is a const-method and cannot modify its class's member variables. If so, then on Line 22, the algorithm appends the keyword **const** to the variable *code*. On Line 23, an open curly brace is added to *code*, starting the function's body. On Line 24, the outgoing edge of type "FunctionBody" is followed, and the "StatementList" node at its head is stored in the *body* variable. Next, all edges of type "Statement" coming from the *body* node are collected into the *stmts* variable. On Lines 26 through 28, the statement list, *stmts* is iterated with the loop control variable *s*. On Line 27, the variable *stmt* is set to the statement node at the head of the edge *s*. Next, the code for the statement is generated using the GetCodeForStatement algorithm; the statement code and a statement-terminating semicolon are appened to the *code* variable. Finally, on Lines 29 and 30, the closing curly brace is added to the *code* variable, and the function's return type node, *retType*, is inserted into the dependency list.

We have presented four algorithms thus far that make up the Code Generator module shown in Figure 4.1. There are many more; we have not included the algorithms for generating C++ source code for classes and expressions, for example, even though they are referenced in the described algorithms. To explain the process for converting every ASG node type defined by the schema into C++ is beyond the scope of this thesis, and wouldn't be very interesting.

We will now walk through a simple "Hello World" code example that exercies this module. Assume an ASG for the following source code has been built:

```
1    #include <cstdio>
2    int main() {
3        fprintf(stdout, "Hello World!\n");
4    }
```

Refering back to algorithm listed in Figure 6.8, we start by locating the reference to main function in the ASG and add it to the *todo* stack. On the first iteration of the loop *todo* has one item, main, so we regenerate the source code for the main function and collect its external dependecies.

```
1    int main() {
2        fprintf ( stdout , "Hello World!\n");
3    }
```

The two hilighted items, fprintf and stdout, are defined externally with respect to main. Both references are pushed onto the *todo*, and both references are inserted into the *depGraph* dependency graph, as dependcies of the main function. On the next iteration of the loop, stdout is popped off the *todo* stack. Its source code is generated and its dependencies are collected.

```
1    extern "C" _IO_FILE *stdout;
```

This time there is only one dependcy. _IO_FILE is pushed onto the *todo* stack and it is inserted into *depGraph* graph as a dependency of stdout. On the next iteration, _IO_FILE is popped from the *todo* stack and source code is generated.

```
1   struct _IO_FILE
2   {
3       int _flags;
4       char *_IO_read_ptr;
4       char *_IO_read_end;
5       char *_IO_read_base;
6       char *_IO_write_base;
7       char *_IO_write_ptr;
8       char *_IO_write_end;
9       char *_IO_buf_base;
10      char *_IO_buf_end;
11      char *_IO_save_base;
12      char *_IO_backup_base;
13      char *_IO_save_end;
14      _IO_marker *_markers;
15      _IO_FILE *_chain;
16      int _fileno;
17      int _flags2;
18      long int _old_offset;
19      unsigned short int _cur_column;
20      signed char _vtable_offset;
21      char _shortbuf[1];
22      void *_lock;
23      long int _offset;
24      void *__pad1;
25      void *__pad2;
26      void *__pad3;
27      void *__pad4;
28      unsigned long int __pad5;
29      int _mode;
30      char _unused2[40];
31  };
```

This class has two dependencies: _IO_marker and itself. Because we now have the source code for _IO_FILE, we only need to push _IO_marker to the *todo* stack. _IO_marker is also added to the *depGraph* graph as a dependency of _IO_FILE. For the _IO_FILE dependency, a declaration is created instead, and that declaration is added to *depGraph* as a dependency of _IO_FILE. On the loop's next iteration, _IO_marker is popped off the stack and its source code is generated.

```
1  struct _IO_marker
2  {
3      _IO_marker *_next;
4      _IO_FILE *_sbuf;
5      int _pos;
6  };
```

This class has the same two dependencies as the previous class: _IO_marker and _IO_FILE. The source code for both of these classes have already been generated, so they are not pushed onto the *todo* stack. For the same reason, only declarations for these two classes are added to the *depGraph* graph as dependencies of _IO_marker. On the next iteration of the loop, fprintf is finally popped off the *todo* stack. The source code for this function is now generated, but because the body resides in a library, only the function's declaration is created.

```
1  extern "C" int fprintf( _IO_FILE *__stream, const char *__format, ... );
```

The single dependency for fprint is the _IO_FILE class, whose source code has already been generated. It is not pushed on the *todo* stack, but a dependecy on _IO_FILE for fprintf is created in the *depGraph* graph. The *todo* stack is now empty, so the loop terminates. The final state of the dependency graph is shown in Figure 6.12. When the graph is traversed in a depth-first order, the complete source code can be arranged in the proper order. The complete regenerated source code for this "Hello World" example is shown in Figure 6.13.

6.7 Verification & Validation of the ASG

6.7.1 Verifying Observable Behavior

One method for verifying the correctness of the ASGs produced is to employ a blackbox test. In this test, the original C++ input program is compiled and executed. An

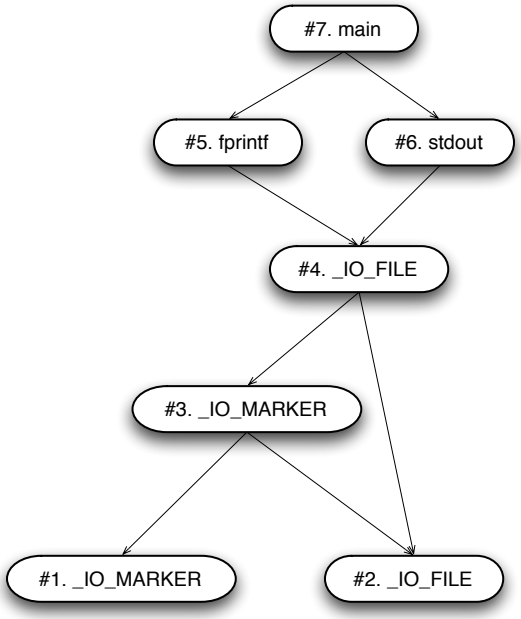


Figure 6.12: Dependency Graph

ASG is built from that same source code. C++ source code is then generated from the unmodified ASG, and the new source code is compiled and executed with the same input as the original program. Although the two programs will not be token-for-token identical, the output from both executable should the same; discounting, of course, nondeterministic values such as timers, random numbers, and memory addresses. If the two outputs differ then we track down issues in ASG Generator module. With a testsuite that provides good coverage on the source code, we gain confidence that the ASG builder is correct. In this section we provide one instance where the blackbox test failed, leading us to discover a significant error in the way our lookup rules were implemented.

Consider the same "Hello World" program we used in Chapter 6.5.0.6 to demonstrate the CreateBinaryExpr algorithm:

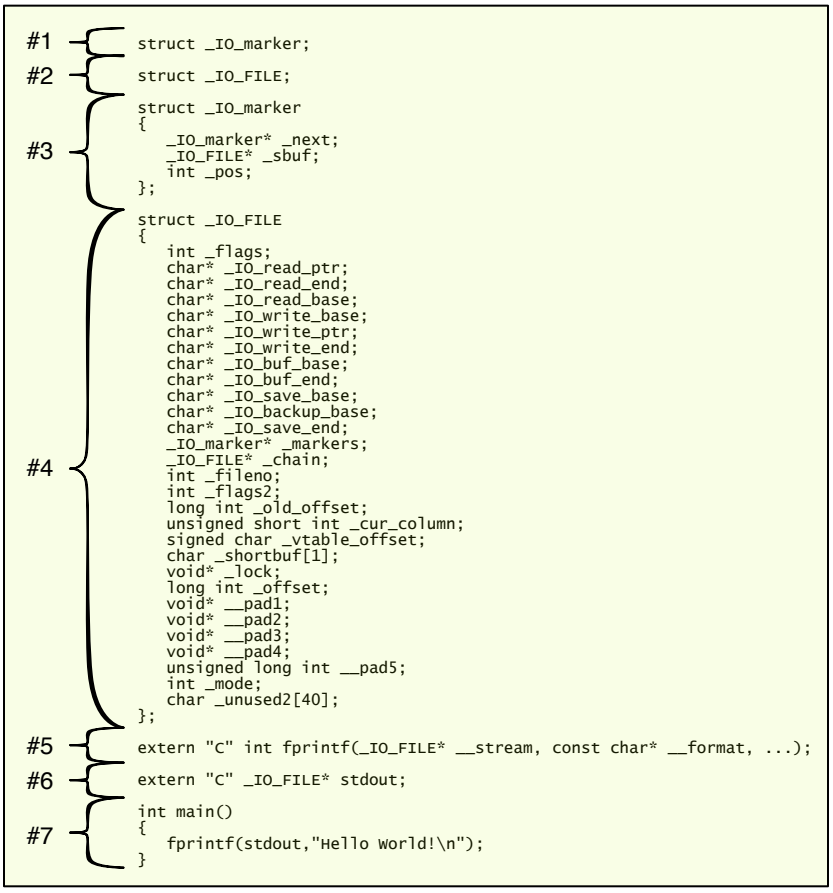


Figure 6.13: Regenerated Code

```
1   #include <iostream>
2   int main() {
3       std::cout ≪ "Hello, World!" ≪ std::endl;
4   }
```

When we first tested this program, the Code Generator module gave back the following C++ source code for the main function (the rest of the code that came from the system header files is not included):

```
1   int main() {
2     std::cout.operator≪("Hello, World!").operator≪(std::endl);
3   }
```

When we ran the program compiled from original source code, the string Hello, World! was printed to the terminal, as expected. However, when we ran the program that was compiled from the above source code, a memory address was printed. Upon tracing the which function was matching the operands' types in the binary expression, we were matching the method that appears on Line 29 in ostream excerpt listed in Figure 6.1:

```
29          __ostream_type& operator≪(const void* __p);
```

The above function that was being selected accepts any possible pointer, and sends its memory address to the output stream. From there we know what happened, but not why. Reviewing our lookup rules revealed that we were prematurely resolving the overloaded function without a complete set of candidates. The proper method requires the collection of all possible candidate functions from the left-hand side operand's class scope and namespace scope, and the right-hand side operand's namespace scope, then the correct function is selected in the resovleOverload algorithm. Instead we looked for the correct function in only the left-hand operand's class scope. If it was not found, then the operand's namespace scope was examined. And if it was not found there either, only then was it looked for the right-hand side operand's namespace code. The incorrect algorithm is shown in the listing below:

```
13    f ← null
14    if not isPrimativeType(ltype):
15        f ← resolveOverload(lookupMembers(ltype, opfunc))
16        if f is not null:
17            f ← resolveOverload(lookupFunctions(getNamespace(ltype), opfunc))
18    if not isPrimativeType(rtype) and f is null:
19        f ← resolveOverload(lookupFunctions(getNamespace(rtype), opfunc))
```

Fixing the lookup rules yields the correct implementation of BuildBinaryExpr that's listed in Figure 6.4. When we rerun the Code Generation module, we get the following C++ source code, which does produce the correct output to the terminal.

```
1    int main() {
2        std::operator≪(std::cout, "Hello, World!").operator≪(std::endl);
3    }
```

6.7.2 Interactive GXL Viewer

In the previous section, we described a technique for testing the observable behavior of an ASG by taking advantage of Hylian Code Generation module. However, performing blackbox testing may not always be sufficient. For example, every expression node in the ASG has to have a C++ type associated with it. When we generated source code for an ASG, we have no way to enforce the type of every intermediate value. We have to inspect the ASG manually to ensure each expression node is linked to the correct C++ type. Also, the implicit construction and destruction of intermediate objects cannot be expressed in source code. Consider the following code sample:

```
1    struct Lock {
2        Lock() { acquireLock(); }
3        ∼Lock() { releaseLock(); }
4    };
5
6    int main() {
7        Lock(), doThreadUnsafeAction();
8    }
```

In the above code, the class *Lock* acquires a lock in its constructor and releases the lock in its destructor. On Line 7, a temporary, nameless *Lock* object is created before the *doThreadUnsafeAction* function is called. Only when the function is complete is the temporary *Lock* object deleted. If we were to build an ASG for this program, and regenerate its code, the two programs would work the same. However, we would not know if the implicit calls to *Lock::Lock()* and *Lock::∼Lock()* are inserted in the ASG. This would be important if we wanted generated a call graph from the ASG to track the flow of data in the program.

A GXL file is stored internally in an XML format, which make it is inherently difficult to read a graph that is represented in a tree-like structure. To help gain insight into how our GXL graphs were arranged, we created an interactive GXL viewer. The viewers that were freely available only produced static images, which made it difficult to comprehend large graphs. Our viewer is built upon the Gtk+ windowing toolkit, and uses neato from the GraphViz project [28] to manage the graph layout algorithm.

The Hylian GXL Viewer allows the user to navigate the graph either by selecting node from a list of node IDs, or to traverse the graph manually with their mouse, showing as many or as few of the node's neighbors as they desire. The viewer also has an attribute explorer, to view all the attributes associated with a node or an edge. And there is a search feature, will allows the user quickly find nodes and edges by type, attribute name, or attribute value.

The ASGs shown in Figures 6.5, 6.6, and 6.7, and the ORD shown in Figure 6.15, are all generated with the GXL viewer.

6.8 Demonstrating Interoperability Through Schemas

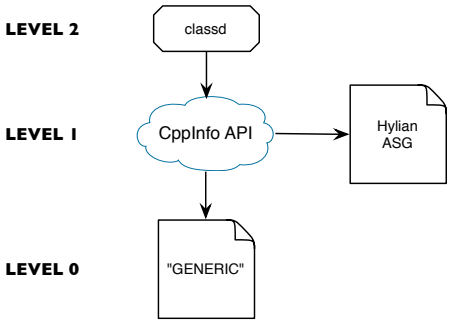


Figure 6.14: Interoperability

An important feature in our approach to generation of abstract semantic graphs (ASGs) is to ensure that our ASGs conform to a schema so that we can reuse previously developed analysis tools that accept conformant ASGs [47, 48]. Figure 6.14 illustrates dependencies in our approach to demonstrating this interoperability with a previously existing tool, classd, from the g^4re suite. classd is a program that accepts ASGs written to conform to the g^4re Level I schema, and generates an object relation diagram (ORD) conforming to a Level II schema. classd is shown at the top of Figure 6.14, and relies on the CppInfo API to query and gather information about an ASG. In the diagram we also show two possible ASG instance types that CppInfo can read. The first ASG instance, GENERIC, corresponds to a g^4re generated ASG that is conformant to the g^4re Level 0 GENERIC ASG schema. The second ASG instance is a Hylian generated ASG that is conformant to the CppInfo schema. Using classd we generate ORDs using ASGs created by Hylian without modification to the original classd source code. To verify this interoperability, we exercised the test suite included in the g^4re suite using ASGs from both g^4re and Hylian. The generated ORDs were identical, and Figure 6.15 shows a screenshot of the GXL Viewer rendering an ORD from the g^4re test suite.

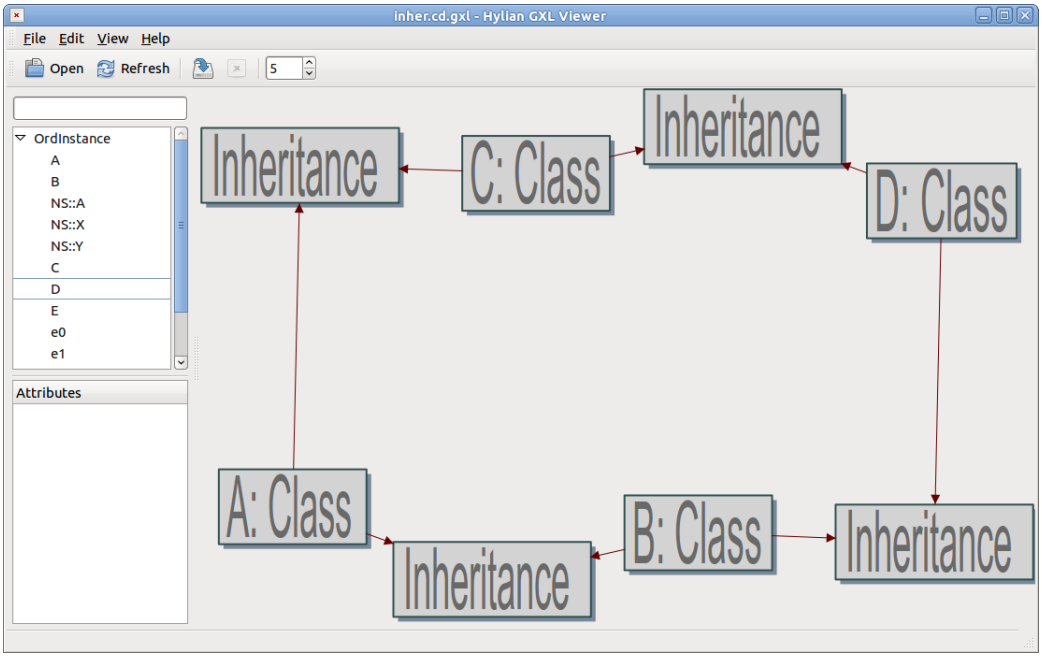


Figure 6.15: Interoperability

6.9 Results for the Construction of Abstract Semantic Graphs

In this section we provide some interesting results for ASGs that we have built using Hylian. Table 6.2 summarizes these results for three programs in our test suite: a Hello World program written in C and C++, and AlephOne, a rather large video game translated into C++ that uses the Simple Direct Media Layer (SDL) API, as well as the Lua embedded scripting engine. There are two sections of data in the table: the top section that summarizes results for the construction of Parse Trees, and the bottom section that summarizes results for the construction of ASGs. We are comparing parse trees and ASGs in the top and bottom sections to show the reduction in the size of ASGs compared to parse trees, and the reduction in size when the ASGs for each compilation unit in a program are *linked*, or merged, into a single ASG for the entire program.

In comparing the various results, we first consider the results for the three "Hello World" programs. The first column in the table describes the information in the particular

	Hello World			Video Game
	In C	**In C++**	**In C++**	**AlephOne**
Parse Trees				
No. Compilation Units	1	1	1	176
No. System Terminal	2,930	132,122	132,122	30,339,853
No. User Terminal	13	16	16	3,441,975
No. Total Terminal	2,943	132,138	132,138	33,781,828
No. Non-Terminals	13,404	518,788	518,788	128,818,128
Max Branch Length	223	374	374	3,671
Avg. Branch Length	123.47	215.33	215.33	652.71
Abstract Semantic Graph				
Using Phantom Parser	N	N	Y	Y
No. Vertices	984	21,861	38	1,914,869
No. Vertices (linked)				478,642
No. Edges	2,098	45,444	39	3,632,959
No. Edges (linked)				1,003,413
No. Statements	137	2,263	1	1,512,638
No. Statements (linked)				115,534

Table 6.2: ASG Statistics.

row of the table, and the next three columns represent information for the "Hello World" programs written in C and C++. Our first interesting result shows that both the C and C++ versions of "Hello World" consist of a single Compilation Unit, as shown by the first row of data in the table for columns two, three and four; however, the No. System Terminals in the C version has 2,930 terminals in the parse trees, but the C++ versions have 132,122 terminals in their parse trees, so that the C++ version has almost two orders of magnitude more terminals than the C version. This is due to the fact that the C++ version of "Hello World" includes the iostream library and much of the C++ standard library. Also, the No. System Terminals in the AlephOne program is 30,339,853 (first row, last column of the table), illustrating the large number of system terminals needed for a standard video game. The remaining rows in the top section of Table 6.2 reflect similar comparisons.

The bottom section of Table 6.2 summarizes results for the ASGs that Hylian builds. The second and third columns in the bottom section compare the ASG for a full parse of a C and a C++ "Hello World" program. For example, the third row of the bottom section of

the table shows that there are 984 vertices in the C version, and 21,861 vertices in the C++ version, an order of magnitude increase. However, the number of vertices in an ASG build by the phantom parser of Hylian shows only 38 vertices, at least two orders of magnitude reduction over the full ASG for C++ "Hello World," and an order of magnitude reduction in the full ASG for the C "Hello World." The fourth row of the bottom section shows that the ASG for the phantom parser required only 39 edges in the ASG for the C++ version, but the full ASG required 45,444 nodes, which is almost three orders of magnitude larger. This illustrates the considerable reduction in the size of ASGs built with the phantom parser and illustrates the reduction in the cognitive burden placed on a maintenance engineer using the Hylian system for program comprehension or analysis. Other rows of the bottom section of Table 6.2 further illustrate this comparison.

Chapter 7

Tranformations on an ASG

Developing and maintaining reliable object-oriented software requires a precise understanding of how individual classes must be used. Unfortunately, for many systems, especially those that are large, the available documentation is inadequate. Developers are frequently confronted with incomplete information concerning the allowable set of call sequences that each class can accommodate. Techniques for reverse engineering this information and presenting it to developers in an intellectually scalable manner are critical. In references [20] and [61] we presented a runtime trace collection system for large C++ applications along with a methodology for reverse engineering interface protocols from these collected trace data.

There are several problems with the approach described in the paper. Our primary data structure in gathering information about the programs under study, and the transformations, were performed on each compilation unit's parse tree. First, we located a branch in the parse for every defined class in a program and inserted a member variable to store each of its objects' "ticket". This was accomplished by adding a subtree into the parse tree that represented a member declaration. Second, for each class, we identified all of the constructor methods for the class and inserted another subtree that represented the initial-

ization of the ticket to some unique value. Third, for every non-static method of a class, we inserted another subtree that represented a printf call that printed the object's ticket number and the fullname of the function. The last insertion is particularly problematic, since it was difficult to decide if a method defined outside of its class was a static method or a member of the class. For example, consider the following code segment:

```
1   class MyClass {
2       static void someFunction();
3   };
4   void MyClass::someFunction() {
5       ...
6   }
```

In this example, *someFunction* is a static member of *MyClass*, but that is not obvious by examining only the site of the definition. It is not enough to keep track of the names of static function declarations that appear in the class body. If we modify the above code sample to the following:

```
1   class MyClass {
2       static void someFunction();
3       void someFunction(int);
4   };
5   void MyClass::someFunction() {
6       ...
7   }
8   void MyClass::someFunction(int) {
9       ...
10  }
```

In this modified example, we show that is possible to have multiple overloaded definitions for a function, where some are static and others are not. In order to correctly probe these functions, not only do we have to track the functions' names, but also their formal parameter declaration list. This is increasingly difficult to accomplish with just a parse tree, since it requires semantic information to correctly determine type equivalency. Faced with that problem, our script that inserted probes incorrectly probed static methods.

This would cause a compiler error in the regenerated source code, since a static method was trying to access an object's member, requiring manual removal of the probe for each erroneous insert.

By taking advantage of the Hylian ASG Generator we alleviate this problem. Using an ASG to determine which functions need to be probed requires only two checks: (1) that a Function ASG node has an incoming edge of type "Contains" from a Class node, and (2) that the function's ASG node has its "is_static" attribute set to false. There are added benefits to using an ASG that are not addressed in our previous research. First, because it is easy to determine where, in scope, a class resides, we can be selective in which classes we want to probe. This was difficult to determine with just a parse tree, so we probe every class in the program. And, second, we can only probe methods that have public accessibility. Since we are reverse engineering the interface protocol of the class, we are only interested in the operations that are available to clients of the class. Having the ability to not trace methods with protected or private accessibility would enhance the quality of our interface protocols.

Chapter 8

Concluding Remarks and Merits of Hylian

In this thesis, we have described our system, Hylian, for statement-level analysis, both static and dynamic, of a C++ application. In particular, we have described and demonstrated our approach for building an abstract semantic graph, ASG, that links the ASGs for each compilation unit in an application. In addition, to alleviate the burden of information that may hinder comprehension of the generated ASG, we have developed an extension to Hylian to build an abbreviated ASG that incorporates information about user code, but not about compiler provided library code. We performed various verification and validation metrics to the ASG, including a viewer that can visualize any graph in GXL format, to provide assurance for a developer that the ASGs that Hylian builds correctly represent the program under investigation.

The value of our system can be viewed in three phases. Chapter 5 details the Parse Tree Generation Phase. In this chapter, we detailed the generation of parse trees by augmenting the *gcc* compiler's C++ parsing subsystem, and the postprocessing work required to backpatch tentatively parsed language constructs, the removal of incomplete, tentatively

parsed subtrees, and the recovery of left-recursive grammar productions that were parsed iteratively. These parse trees were verified for their structural correctness, and validated against our grammar schema. By exercising the parse tree generator with an extensive testsuite, we were able to recover the *gcc* dialect of the C++ programming language. We represent this grammar in three formats: (1) a human-readable LaTeXdocument, (2) a Relax-NG schema, an XML schema for validating regular languages, which is used to validate parse trees represented in XML, and (3) a YACC specification program, which was used as a starting point in parsing parse trees for the second phase in Hylian.

Chapter 6 covers the generation of Abstract Semantic Graphs. In this chapter, we detail extensions to the ASG schema, first defined in the g^4re project, to handle statement- and expression-level language consutrcts. We then provided two algorithms for constructing ASGs. The first, an approach to represent every program artifact for an application in a single graph; including all user-defined and system-provided data structions, all class- and function-templates instatiated, all constants reduced, all type aliases resolved, all explicit- and implicit-function calls resolved, and every expression-level ASG node decorated with its appropriate type. The second approach to ASG construction, a superset of the first, builds an abbrivated version of the ASG that excludes compiler-provided library files, and only considers user developed source code. This second approach required additional extenstions to the ASG schema. We provided further detail to the construction of the ASG by describing the algorithm we used to evaluate binary expressions and class template instanatiated, along with example traces. We provided an algorithm and examples for transforming a complete ASG into C++ source code. We provided two approaches to ensuring the validity of our ASGs: (1) using the source code generator, we validate that the observable behavior of an ASG is the same as its input source code; and (2) we developed a visualization tool to manually inspect ASG for correctness beyond what may be determined by black-box testing. We developed a compatibility layer conforming to the CppInfo API.

This allows tools from the g^4re project to accept ASGs produced by Hylian as input. We demonstrated the usefulness of this compatibility layer by generating object relation diagrams using g^4re's classd program without modification. Finally, we presented statistics about input programs to the ASG builders.

Chapter 7 covers the problems we encountered in previous research by relying only on parse trees to transform C++ source code. We discuss the inadequacy of parse trees, for certain problem domains, such as source code transformations to enable dynamic program analysis, and the need for an type-complete ASG. We also explore possibilities for improving the research that were not considered when the idea Hylian was conceived.

Bibliography

[1] GCC-XML. http://www.gccxml.org, February 2005.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[4] G. Antoniol, M. Di Penta, G. Masone, and U. Villano. Compiler hacking for source code analysis. *Software Quality Journal*, 12(4):383–406, December 2004.

[5] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, SE-8(4):343–353, July 1982.

[6] Bell Canada Inc. *DATRIX - Abstract Semantic Graph Reference Manual*. Bell Canada Inc., Montreal, Canada, 1.4 edition, May 2000.

[7] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1):23–41, 2006.

[8] Blitz. http://www.oonumerics.org/blitz/.

[9] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *The second annual object-oriented numerics conference (OON-SKI)*, pages 122–136, Sunriver, Oregon, USA, 1994.

[10] Boost. http://www.boost.org/.

[11] E. Bouwers, M. Bravenboer, and E. Visser. Grammar engineering support for precedence rule recovery and compatibility checking. In *LDTA'07*, pages 82–96, March 2007.

[12] A. Celentano, S. Reghizzi, P. Della Vigna, and C. Ghezzi. Compiler testing using a sentence generator. *Software – Practice and Experience*, 10:897–913, 1980.

[13] C. L. Cox, E. B. Duffy, and J.B. von Oehsen. FiSim: An integrated model for simulation of industrial fibre and film processes. *Plastics, Rubber and Composites*, 33(9-10):426–437, November 2004.

[14] T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a c++ extractor. In *Working Conference on Reverse Engineering*, October 2001. www.cppx.com.

[15] Joel E. Denny. Pslr(1): Pseudo-scannerless minimal lr(1) for the deterministic parsing of composite languages, 2010. PhD Dissertation.

[16] Derek du Preez. C++ clear winner in google language tests, 2011.

[17] E. B. Duffy and B. A. Malloy. A language and platform-independent approach for reverse engineering. *Proceedings of The 3rd ACIS International Conference on Software Engineering Research, Management & Applications*, August 2005.

[18] Edward B. Duffy and Brian A. Malloy. An automated approach to grammar recovery for a dialect of the c++ language. In *Proceedings of the 14^{th} Working Conference on Reverse Engineering*, pages 11–20. IEEE Computer Society, 2007.

[19] Edison Design Group. C++ Front End. http://www.edg.com/cpp.html, December 2000.

[20] Jason O. Hallstrom Edward B. Duffy and Brian A. Malloy. Reverse engineering interface protocols for comprehension of large C++ libraries during code evolution tasks. In *Proceedings of the The 20th International Conference on Software Engineering and Knowledge Engineering*, 2008.

[21] R. Ferenc and A. Beszedes. Data exchange with the columbus schema for c++. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 59–66, Budapest, Hungary, March 2002.

[22] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, October 2002.

[23] Fluxbox. http://www.fluxbox.org/.

[24] Free Software Foundation. GNU Compiler Collection. http://www.gnu.org/software/gcc/, January 2002.

[25] FreePOOMA. http://www.nongnu.org/freepooma/.

[26] FTensor. http://www.oonumerics.org/FTensor/.

[27] K. Gallagher and D. Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 316, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[30] GCC, the GNU Compiler Collection. http://www.gnu.org/software/gcc/.

[31] GCC Testing Efforts. http://www.gnu.org/software/gcc/testing/.

[32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, third edition, 2005.

[33] LOOSE Research Group. iPlasma: An integrated platform for quality assessment of object-oriented design, 2007. http://loose.upt.ro/iplasma/.

[34] T. Gschwind, M. Pinzger, and H. Gall. Tuanalyzer – analyzing templates in c++ code. In *WCRE'04*, pages 48–57, Washington, DC, USA, 2004. IEEE Computer Society.

[35] M. J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. In *Technical Report OSU-CISRC-5/96-TR32, Department of Computer and Information Science, The Ohio State University (OSU-CISRC-5/96- 1996)*, May 1996.

[36] M. Hennessy, B. A. Malloy, and J. F. Power. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. In *Proceedings of International Workshop on Program Comprehension*, pages 298–299, Portland, Oregon, USA, May 2003. IEEE.

[37] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proceedings of the 20^{th} IEEE/ACM International Conference on Automated Software Engineering*, pages 104–113, November 2005.

[38] W. Homer and R. Schooler. Independent testing of compiler phases using a test case generator. *Software – Practice and Experience*, 19(1):53–62, January 1989.

[39] ISO/IEC JTC 1. *International Standard: Programming Languages – C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[40] J. Clark and M. Makoto. RELAX NG Specification. *http://www.oasis-open.org/committees/relax-ng/spec-20011203.html*, 2007.

[41] P. Klint, R. Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[42] G. Knapen, B. Lague, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, May 5-7 1999.

[43] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[44] N. A. Kraft, B. A. Malloy, and J. F. Power. g^4re: Harnessing gcc to reverse engineer C++ applications. In *Seminar No. 05161: Transformation Techniques in Software Engineering*, Schloss Dagstuhl, Germany, April 17-22 2005.

[45] N. A. Kraft, B. A. Malloy, and J. F. Power. Toward an infrastructure to support interoperability in reverse engineering. In *Proceedings of the Twelfth Working Conference on Reverse Engineering*, Pittsburgh, PA, USA, November 2005. IEEE Computer Society.

[46] Nicholas A. Kraft. *An Infrastructure to Support Interoperability in Reverse Engineering*. PhD thesis, Clemson University, 2007.

[47] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. An infrastructure to support interoperability in reverse engineering. *Information and Software Technology*, 49(3):292–307, 2007.

[48] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. A tool chain for reverse engineering C++ applications. *Science of Computer Programming, Special Issue on Experimental Software and Toolkits*, 2007. (accepted for publication).

[49] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. A tool chain for reverse engineering C++ applications. *Science of Computer Programming*, 69(1–3):3–13, December 2007.

[50] R. Lämmel. Grammar testing. In *FASE*, pages 201–216, 2001.

[51] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *SP&E*, 31(15):1395–1438, December 2001.

[52] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software: Practice and Experience*, 31(15):1395–1438, October 2001.

[53] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[54] S. Lapierre, B. Lague, and C. Leduc. Datrix source code model and its interchange format: Lessons learned and considerations for future work. *ACM SIGSOFT Software Engineering Notes*, 26(1):53–56, January 2001.

[55] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[56] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

[57] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for integration testing of C++ applications. In *International Symposium on Software Reliability Engineering*, pages 353–364, Denver, CO, USA, Nov 2003.

[58] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 33(1):19–39, 2003.

[59] B. A. Malloy, T. H. Gibbs, and J. F. Power. Progression toward conformance for C++ language compilers. *Dr. Dobbs Journal*, pages 54–60, November 2003.

[60] B. A. Malloy, T. H. Gibbs, and J. F. Power. Progression toward conformance for C++ language compilers. *Dr. Dobbs Journal*, pages 54–60, November 2003.

[61] B. A. Malloy, E. L. Lloyd, J. O. Hallstrom, and E. B. Duffy. Capturing interface protocols to support
comprehension and evaluation of C++ libraries. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*.

[62] B. A. Malloy and J. F. Power. An interpretation of purdom's algorithm for automatic generation of test cases. In *Proceedings of 1st Annual International Conference on Computer and Information Science (ICIS '01)*, Orlando, Florida, USA, October 2001.

[63] B. A. Malloy and J. F. Power. Program annotation in XML: A parser-based approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 190–198, Richmond, Virginia, USA, October 2002. IEEE.

[64] Brian A. Malloy, Tanton H. Gibbs, and James F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33(1):19–39, 2003.

[65] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[66] S. McPeak. Elkhound: A GLR parser generator and Elsa: An Elkhound-based C++ parser, 2007. http://www.cs.berkeley.edu/ smcpeak/elkhound/.

[67] S. G. McPeak. Elkhound: A fast, practical glr parser generator. Technical Report UCB/CSD-02-1214, EECS Department, University of California, Berkeley, 2002.

[68] The Mozilla Project. http://www.mozilla.org/.

[69] V. Murali and R. K. Shyamasundar. A sentence generator for a compiler for PT, a pascal subset. *Software – Practice and Experience*, 13:857–869, 1983.

[70] N. A. Kraft. g^4re reverse engineering infrastructure, version 0.0.1, 2007. Available at http://g4re.sourceforge.net.

[71] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, pages 57–68, Sydney, Australia, November 2000.

[72] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *37th International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS Pacific 2000)*, pages 57–68, Sydney, Australia, November 2000.

[73] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[74] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, April 1972.

[75] S.P. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[76] J.A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

[77] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of cobol/cics legacy systems. *Sci. Comput. Program.*, 45(2-3):193–243, 2002.

[78] Source–Navigator Team. The Source–Navigator IDE. http://sourcenav.sourceforge.net, June 2005.

[79] SCLC. http://www.cmcrossroads.com/bradapp/clearperl/sclc.html.

[80] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[81] The Object Modeling Group, Inc. OMG XML Metadata Interchange Specification (Version 1.2). January 2002.

[82] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[83] M. Tomita and S.-K. Ng. The generalized lr parsing algorithm. In M. Tomita, editor, *Generalized LR Parsing*, pages 1–16. Kluwer, Boston, 1991.

[84] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *IWPC*, page 108, Washington, DC, USA, 1998.

[85] D. van Heesch. Doxygen. http://stack.nl/ dimitri/doxygen/.

[86] T. L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University, 2003.

[87] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.