Clemson University
TigerPrints

All Theses | Theses

5-2016

Construction of a Generic Program Representation for Automated Metric Computation

Zachary McNellis
Clemson University, zmcnell@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Recommended Citation
McNellis, Zachary, "Construction of a Generic Program Representation for Automated Metric Computation" (2016). *All Theses*. 2341.
https://tigerprints.clemson.edu/all_theses/2341

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Construction of a Generic Program Representation for Automated Metric Computation

---

A Master's Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
M.S. Computer Science
Computer Science

---

by
Zachary McNellis
May 2016

Accepted by:
Dr. Brian Malloy, Committee Chair
Dr. Joshua Levine
Dr. Sekou Remy

Abstract

Software code metrics provide a quantitative and qualitative measurement of a software component's ability to perform under a specific set of objectives. Different metrics have been developed for analyzing various aspects of the source code to gain insight into the overall quality of the code under study. There are a variety of open source tools available for computing metrics for applications coded in most of the popular programming languages. However, there is no single tool that computes software metrics for the popular programming languages in use today. To address this problem, we describe an approach to software metric computation that can be applied to the popular programming languages currently in use, including both compiled and interpreted languages. The approach entails leveraging existing parser tools to produce a generalized abstract syntax tree that captures the important syntactic categories required for metric computation. To demonstrate the utility of our approach, we exploit front-end parser tools for the Python and C++ programming languages to produce a generalized abstract syntax tree and then compute software metrics as a form of tree traversal. We describe our results for applying two commonly used metrics to three open source software projects and various code samples written in both Python and C++. The context of this process is then extended to computer programming education, with the specific goal of helping students and programmers improve the quality of their code.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

List of Tables . . . . **v**

List of Figures . . . . **vi**

List of Listings . . . . **vii**

1 Introduction **1**

2 Background and Related Work **4**
- 2.1 Software Metrics 4
- 2.2 Abstract Syntax Trees 10
- 2.3 Assessment of Online Code Instruction Tools 12

3 Methodology **17**
- 3.1 Overview of our Approach 18
- 3.2 Step 1: AST Construction 19
- 3.3 Step 2: Generic AST Construction 19
- 3.4 Step 3: Validating Structural Correctness 20
- 3.5 Step 4: Code Evaluator 20
- 3.6 Application: CodeR Online Evaluator 22

4 Results **26**
- 4.1 Evaluation of Open Source Projects 28
- 4.2 Evaluation of Solutions for Project Euler 29
- 4.3 CodeR Results 33

5 Conclusion and Future Work **38**
- 5.1 Future Work 40
- 5.2 Future of Online Code Instruction Tools 40

Appendices **41**

A Results for Extended McCabe Cyclomatic Complexity 42
References 43

List of Tables

2.1 Halstead Variables Defined 8

4.1 McCabe Metrics for PureMVC, ua-parser, and Apache Etch 28
4.2 Descriptions of the First Five Exercises for Project Euler 30
4.3 Posted Solutions Written in C++ 30
4.4 Posted Solutions Written in Python 31

List of Figures

3.1 Overview of the GAST System 18
3.2 Code Evaluator Implementation 20

4.1 Cyclomatic Complexity for Euler Problems 1-5 33
4.2 Halstead Difficulty for Euler Problems 1-5 34
4.3 CodeRx Prototype 35
4.4 CodeRx Results 36

1 Extended Cyclomatic Complexity for Euler Problems 1-5 42

List of Listings

2.1 SLOC Python Code 6
2.2 Hamming Distance v1 7
2.3 Hamming Distance v2 8
3.1 GAST XML Schema Definition 22

Chapter 1

Introduction

In the past four decades, research in the design of programming languages and environments has focused on making computer programming accessible to a wider range of users. Nevertheless, surveys show that programming remains a daunting task and that little progress has been made in the fundamental area of computer programming instruction [32, 39]. Numerous explanations for the difficulty of learning to write correct programs have been proposed, including: rigid language syntax, commands with seemingly arbitrary or confusing names, the challenge of identifying structured solutions to match program specifications, and an inability to comprehend how instructions are executed by a computer [4, 11, 15, 21, 28].

To address the problem of learning to program, a multitude of forums and tools that facilitate computer programming instruction have become popular, including *Codecademy*, *Code Wars*, and *Project Euler* [7, 9, 13]. These approaches foster computer programming by proposing problems to be solved and then permitting users to submit code solutions to be evaluated for correctness. These forums have been successful in promoting computer programming but neither they, nor any other forum, have developed techniques that promote automated evaluation of the quality of the

submitted code in an effort to foster the development of programs that are easy to extend and maintain.

The process of assessing software quality is well studied and the underlying objective remains the same: to analyze relevant attributes for a particular software component as a means to measure how well it meets certain requirements, while also providing an indicator for the overall quality of the software product as a whole [5, 31, 36]. The most commonly used approach toward measuring software quality is through the use of *software metrics*, which tend to reflect non-functional requirements, and their selection is largely influenced by the specific task or project at hand [6]. Often these quality attributes contradict each other, and so the emerging task of prioritizing them becomes non-trivial. As a result, there is no universal software metric that measures all of the desirable features of an application [10]. Since no single metric can provide a complete evaluation of an application, most developers employ a combination of many software metrics that together measure the quality of the features that are important to the project under development. Thus, both the selection and use of software metrics as a collection offers an important role in the quantification and measurement of software quality.

In addition to the problem of metric selection, there is also the problem of finding a metric tool that can accommodate the particular language that is used to develop the application. A common problem occurs when a developer decides on the specific set of metrics that are appropriate to evaluate the application but cannot find metric tools that can be applied to the language used to code the application.

To address the problem of software evaluation for the popular languages used in developing applications, we describe an approach to software metric computation that can be applied to the popular programming languages currently in use, including both compiled and interpreted languages. The approach entails leveraging existing

parser tools to produce a generalized abstract syntax tree that captures the important syntactic categories required for metric computation. In addition, we provide an interoperable framework for metric computation such that additional metrics can be performed on this generic program representation as validated by our schema definition. We refer to this generalized program representation as a generic abstract syntax tree (GAST), which introduces a further level of abstraction than the abstract syntax tree (AST) that is also language-agnostic. To demonstrate the utility of our approach, we exploit front-end parser tools for the Python and C++ programming languages to produce a generalized abstract syntax tree and then compute software metrics as a form of tree traversal. We describe our results for applying two commonly used metrics to three open source software projects and various code samples written in both Python and C++. The context of this process is then extended to computer programming education, with the specific goal of helping students and programmers improve the quality of their code.

The remainder of this thesis is organized as follows. In Chapter 2, we provide background about the techniques and terminology that we use. In Chapter 3, we describe our code evaluation strategy and its implementation. Chapter 4 provides an evaluation of this approach and our prototype for assessing the quality of student program submissions: CodeR . We draw conclusions and describe future work in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we review the tools and terms used in this thesis. In Section 2.1, we discuss various software metrics and their uses. Next, we explain the fundamentals of abstract syntax trees and existing tools used to generate them in Section 2.2. Finally in Section 2.3, we provide an assessment of a select number of online code instruction tools for the purpose of furthering computer science education.

2.1 Software Metrics

Software metrics play a significant role in the design, implementation, and testing of software [10]. In essence, they provide a quantitative measurement of the performance of a certain piece of software [31]. Effective use of software metrics can provide important information for the management of software at all stages of its life cycle. For example, certain metrics may help assess and predict cost, complexity, and overall effort. Some analysts choose to classify these metrics by their role in the development cycle, such as process, product, and resource metrics [5]. Another intuitive way to classify these metrics are by programming paradigm, as some metrics

may measure attributes like cohesion and coupling that may lend itself to an object-oriented approach. The metrics that are relevant for evaluating code quality are known as software code metrics, or those metrics that can be "directly countable from source code" [5]. The discussion about the selection of metrics for different software tasks and styles is thus important, yet it is beyond the scope of this thesis. The goal of this paper is instead to explore a more general strategy for automatic evaluation of code quality, that is language-agnostic and whose metrics can be interchanged. Next, we describe a few standard metrics that we will later implement.

2.1.1 Source Lines of Code

Perhaps one of the oldest, most commonly used, yet somewhat denigrated software metrics is the source lines of code (SLOC) metric. Its basic definition and implementation are rather simple, as it is used to quantify the size of a program by counting the number of lines in the source code. However, this method can become increasingly complex as there exist many variations on what to consider a *meaningful* line of code [5]. Although many agree that larger-sized functions are harder to maintain and thus considered more complex, it is often cited as a misconception for being a useful predictor for software quality [20, 35]. Despite its controversy, it is commonly used as an indicator of productivity or estimated effort [20]. Here, it is used primarily to introduce the concept of software metrics with a more simple example. The following code in Listing 2.1 shows a simple Python implementation of SLOC.

2.1.2 McCabe's Cyclomatic Complexity

In 1976, Thomas McCabe attemted to address the issue of "how to modularize a software system so the resulting modules are both testable and maintainable" [26].

```python
def SLOC(filename):
    count = 0;
    fd = open(filename, "r");
    for line in fd:
        count += 1;
    return count;
```

Listing 2.1: SLOC Python Code

The result is a commonly used metric known as cyclomatic complexity that attempts to quantify the complexity of a program module in a single numerical value. This is accomplished by counting the number of linearly independent paths through a flow graph of the program's source. In theory then, the number of test cases needed to sufficiently maintain a program will equal the cyclomatic complexity of the program. Therefore, cyclomatic complexity plays a useful role as an indicator of required effort in the maintainability of a software component.

McCabe's cyclomatic complexity is formally defined using a control flow graph of the given program [26]. A control flow graph is a type of directed graph, whose edges indicate the possible flow of control between nodes, or the basic blocks of the program. Then, let us denote E to be the number of edges, N the number of nodes, and P the number of disconnected pieces[1] of the graph. Following this representation of a structured program, McCabe [26] defines the cyclomatic complexity as:

$$V(G) = E - N + 2P$$

As stated earlier, however, no metric can capture the entire meaning of complexity in a single value [10]. In other words, the cyclomatic complexity might not always match an individual's interpretation of complexity. By design, McCabe's cy-

[1] It is common for a program to be structured into a calling program and one or more subroutines. Building a control graph, in this case, would result in having a disconnected graph for each subroutine and calling procedure. Thus, we define P in this way.

```
1 def foo(a, b):
2     distance = 0;
3     if (a[0] != b[0]):
4         distance += 1;
5     if (a[1] != b[1]):
6         distance += 1;
7     if (a[2] != b[2]):
8         distance += 1;
9     return distance;
```

Listing 2.2: Hamming Distance v1

clomatic complexity counts the number of control statements while ignoring the size of the code under each node [19]. Thus, low complexity is not always translated to high readability. Listings 2.2 and 2.3 show code segments written in Python that compute the Hamming Distance of two binary numbers of length 3. Note that both functions have equal cyclomatic complexity values, although their interpreted complexity appears to differ substantially.

Another important consideration when calculating the cyclomatic complexity of a function is how to construct the control flow graph. Unfortunately, McCabe does not explicitly define what control statements establish a branch [40]. Because of these ambiguities, there exist many variations of cyclomatic complexity [40]. For example, the complexities for Listings 2.2 and 2.3 may differ depending on how each of the control statements are managed in the formation of the control flow graph.

2.1.3 Halstead Formulas

Another well known set of complexity metrics is known as the Halstead complexity measures, as introduced by Maurice Howard Halstead in 1977 [18]. This suite of metrics includes several metrics marking different attributes of a program [18, 31]. First, we look at Halstead's definition of program Volume. A list of variables and their

```
1 def bar(a, b):
2     count = 2;
3     distance = 0;
4
5     while (count>-1):
6         distance = 0;
7         for i in range(count, 3):
8             if (int(a[i]) ^ int(b[i]) != 0):
9                 distance = distance + 1;
10        count = count - 1;
11    return distance;
```

Listing 2.3: Hamming Distance v2

Variable	***Meaning***
η_1	number of distinct operators
η_2	number of distinct operands
η_1^*	minimum number of distinct operators
η_2^*	minimum number of distinct operands
N_1	total number of operators
N_2	total number of operands
N_1^*	minimum number of operators
N_2^*	minimum number of operands
V	Volume
V^*	Potential Volume
L	Program Level
$\hat{L}$	Estimated Program Level
D	Program Difficulty

Table 2.1: Halstead Variables Defined

corresponding meanings are listed in Table 2.1. A program's Volume is a measure of a program's size in bits, defined by:

$$V = (N_1 + N_2) * log_2(\eta_1 + \eta_2)$$

Then the Potential Volume, or the volume of a procedure in its most succinct implementation, is defined as:

$$V^* = (N_1^* + N_2^*) * log_2(\eta_1^* + \eta_2^*)$$

Now, we can define the Program Level, L, as an indicator of a human's ability to understand the program. It is based on Volume, V, and Potential Volume, V^*. Only the most efficient implementation of the program can have a Program Level of 1. The formula for Program Level is:

$$L = V^*/V$$

This assumes that we are able to calculate the Potential Volume and thus able to know the most efficient implementation of the code. Since this is usually not known, we can calculate the Estimated Program Level as follows:

$$\hat{L} = \frac{2}{\eta_1} * \frac{\eta_2}{N_2}$$

Finally, we can calculate the Program Difficulty as the inverse of Program Level:

$$D = 1/L$$

As a program's Volume increases, the Difficulty increases. An increased number of operators and operands will also increase the Volume and therefore Difficulty. For

the sake of creating a general program representation, we refer to the operators and operands in the program as equivalent to the arithmetic operators and operands defined in the programming language. Much like complexity, a program's difficulty is often used as an indicator of software quality. In summary, by calculating the number of operators and operands of a program as interpreted in Table 2.1, we refer to a program's difficulty with the following formula:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

2.2 Abstract Syntax Trees

The role of a compiler front-end can be viewed as the transformation from a concrete representation of the source code to a more abstract structure [14]. It is this language-specific, abstract form that is useful for the analysis and transformation of programs.

More commonly, this abstract structure is viewed as a hierarchical data structure known as an abstract syntax tree (AST). The AST retains the overall structure of a program, while ignoring the concrete details of syntactic information [14]. Therefore, ASTs have an important role in semantic analysis, program correctness, and code generation [3]. For our purposes, we are most concerned with utilizing the AST as an analysis tool to facilitate comprehension and evaluation of software applications.

While our code evaluation strategy is detailed in the next chapter, the first step in doing this analysis process is the AST generation. Generating an abstract syntax tree from a program's source code is non-trivial. Fortunately, there exist many tools that support the analysis of programs in their native language. Two such libraries are the Clang front-end parser for C-family languages and the Python *ast* module

for Python programs. For most programming languages, there exist many tools for the parsing and analysis of their respective programs. An exception to this is for the C++ language [12, 25]. In order to address this problem and demonstrate our method in a variety of programming languages, we have chosen Clang and Python's *ast* module to showcase the ability to generate ASTs for both C++ and Python applications.

2.2.1 Clang

Clang is an open source, front-end for the LLVM compiler project [22]. It is designed to work with the C family of programming languages, including C, C++, Objective-C, and Objective-C++. Its modular design makes it easy to use and tap into the internals of the compilation process. Clang provides several tools and APIs that provide the user access to the syntax and semantics of a program. These include LibClang, Clang AST, and libTooling [25]. The most robust and stable tool is LibClang, which is a C-language API that uncovers a large number of mechanisms for the analysis of programs [22]. Most importantly, LibClang allows the user to gain access to the AST, which is built by the Clang parser.

LibClang also offers a number of Python bindings that support the analysis of code at the level of an AST [23]. However, the documentation for the LibClang tool and Python bindings are largely lacking. Fortunately, there are resources which help to uncover these APIs [12, 25].

Even though the number of parsing and analysis tools for applications in C++ is dire [25], Clang provides a complete parse of C++ applications. This makes Clang a powerful and mature tool, and is backed by major companies such as Apple and Google [22]. For our purposes, we use LibClang and its Python wrappers only

at the level of an AST to support the use of software metrics and further analysis of our C++ applications.

2.2.2 Python *ast* Module

Python's *ast* module provides an interface to Python's internal parser program [16]. This built-in Python library helps expose methods for visiting and analyzing the AST of Python programs. It makes the task of creating a scanner and parser simple with the single **ast.parse()** function. The user does not have to worry about traversing the tree, and can instead focus on performing the code analysis by implementing a class derived from the **ast.NodeVisitor** class. The user is then tasked with overriding member functions essential to the analysis of the program.

2.3 Assessment of Online Code Instruction Tools

There is a shortage of students in science and engineering disciplines across the world [37]. Although science and engineering have gained attention as an important area of study, the U.S. has fallen behind in taking action to meet its demands, especially in K-12 education [8]. According to the U.S. Bureau of Labor Statistics and Code.org Computer Science Education Initiative, the number of computer science jobs will soon far exceed the number of college graduates in this field [8, 29]. Even if one does not go into a field of information technology, computer science courses are still beneficial. The skills and knowledge earned in computer science courses are critical to understanding the underlying logic and science present in today's fast-changing technology. To meet these increasing demands, many universities and organizations have turned to online education due to increased accessibility and reduced cost [2, 30]. Similarly, several web-based educational tools such as *Codecademy*, *Code Wars*, and

Project Euler offer unique approaches to teach programming principles in addition to making computer science education more accessible [7, 9, 13].

We look to improve the user experience of these online educational tools by applying our code evaluation strategies. The comprehension and evaluation of code quality has a broad range of applications, especially during the development and maintenance of software. These applications also extend to the use of software code metrics to assess the quality of code in an online, educational environment. We look at several free, web-based problem solving tools in an attempt to further their efforts in providing an accessible and effective learning environment for computer science.

2.3.1 Codecademy

Codecademy was started in 2011 as an online programming instruction tool that has since grown in popularity and support [7]. It features free[2] step-by-step lessons and quizzes organized into lesson units. These modules are then grouped into courses for learning a large number of programming languages, web developer skills, and other programming goals. This makes it easily accessible for those who wish to become familiar with the syntax and fundamentals behind an array of popular programming languages, including HTML/CSS, JavaScript, jQuery, PHP, Python, and Ruby. Individual lessons typically feature a few paragraphs of text introducing a new concept, followed by a set of programming instructions that the user must complete. On this page is also located a text editor where the user enters the code necessary to complete the indicated objective. Upon submitting the user entered code, the page then verifies whether the task was completed correctly. An incorrect submission will cause the page to report the terminal's error message along with

[2] *Codecademy* also offers a premium account for additional content and courses, including a separate course introducing the fundamentals of Java.

helpful hints and further guidance.

Many sites like *Codecademy* use a technique known as sandboxing to capture and execute user code safely [33]. After doing this, the code must be checked for accuracy. In order to verify the correctness and completeness of a solution, the user must closely follow the naming conventions and attributes specified in the directions. This approach does not allow for code clones and thus can be considered somewhat limited as far as code comprehension.

2.3.2 Code Wars

Unlike *Codecademy's* lesson based approach to teaching programming fundamentals, *Code Wars* presents several programming challenges in a competitive training environment to help users learn and improve their programming skills. Currently, *Code Wars* supports several programming languages such as CoffeeScript, JavaScript, Python, Ruby, Java, Clojure, Haskell, and C# [9]. The website also mentions their effort to expand support to F#, Objective-C, PHP, C, and C++ in the future.

Once a *Code Wars* account is created, the user can choose between a list of challenges, or kata. Alternatively, a random kata will be selected based on the chosen language preferences using the "Train" button. Completing challenges will earn honor points based on the kata ranking. Also, completing higher ranked kata will then match the user with challenges based on relevant difficulty settings. In addition to many available social features, the user can author their own kata challenges to present to the online community.

To evaluate the correctness of the entered program, the site simply runs the code through several test cases and matches the user output with the correct solution. After the user submits the correct solution (or forfeits his or her ability to earn

honor points by prematurely unlocking the solution), a page appears with the correct solutions of other members. Users can then rate each solution by cleverness and best practices, along with viewing similar variations for each solution. Thus, the site implements a feature called "solutions grouping" where it attempts to "group similar solutions together so that they may be voted on and discussed as a group" [9]. It is unclear how exactly this analysis is performed, though they encourage feedback on how this process can be improved.

Interestingly, *Code Wars* implements a code quality evaluation strategy that is often overlooked as an effective technique. Instead of using software metrics or formal analysis strategies, *Code Wars* relies on the community to provide user submitted feedback on the quality of others' code. This, in combination with their solutions grouping technique, offers a viable alternative to software quality evaluation. However, the drawback of this approach is that it provides a non-uniform way of evaluating quality that is in turn very subjective. In other words, new content does not receive immediate feedback, and aspects of software quality are constrained to cleverness and best practices, which may invoke different interpretations across different users. In addition, it may take several days or months before a "great answer" is voted enough times where other users can see, or even unintentionally grouped as a variant of somebody else's answer.

2.3.3 Project Euler

Project Euler aims to "encourage, challenge, and develop the skills and enjoyment of anyone with an interest in the fascinating world of mathematics" [13]. It is an online collection of challenging problems with a focus in mathematics. Problems range in difficulty from easy to challenging, though most can be solved using a suc-

cinct algorithm. Creating a free account allows the user to track their progress and receive achievement levels and awards depending on the number of problems solved. It currently has over 500 problems, with only 51 members having solved all 500+ problems. Ever since its creation in 2001, *Project Euler* has increased in popularity, with over 500,000 registered members that have solved at least one problem [13].

Compared to *Codecademy* and *Code Wars*, *Project Euler* is much more problem-oriented and thus does not contain an online terminal or text editor. The problems are generic enough such that they do not lend themselves to any one programming language or solution strategy. Instead of displaying a unique answer, each question has a designated forum page that can be accessed only after the user has entered a correct solution. This forum specific approach provides a unique way of accessing other members' posted solutions and comments.

While the forum specific approach allows for user discussion and solution sharing, it becomes difficult to evaluate the individual quality of a solution. In addition, the awards for accomplishing various tasks could potentially be refined by awarding solutions that score high code quality evaluations.

These limitations of *Codecademy*, *Code Wars*, and *Project Euler* lend themselves to an evaluation approach for software quality that is both automated and customizable based on the intended focus of the website. For example, one website might favor a quality assessment that considers complexity more heavily than code styling for a beginner. In addition, the ability to easily support evaluation of code quality across many languages allow for a potentially improved problem solving environment that is independent of programming language or paradigm.

Chapter 3

Methodology

In this chapter we describe our strategy for automated code quality evaluation for a variety of popular languages. We first evaluate the program for syntactic and functional correctness before we begin the process of evaluating the quality of the code. In the next section, we provide an overview of our approach. In Section 3.2, we describe the data structure that captures the information that we need for metric computation. We review our approach in Section 3.3 for incorporating the information from this data structure into a modified, language-independent representation. In Section 3.4, we describe the process for validating the resulting data structure before proceeding to the metric evaluation step. In Section 3.5, we review the languages and metrics that we use for the evaluation of our methodology. Finally, we discuss the implementation of our approach in the CodeR prototype to explore how this code evaluation technique can be applied in an online programming education environment.

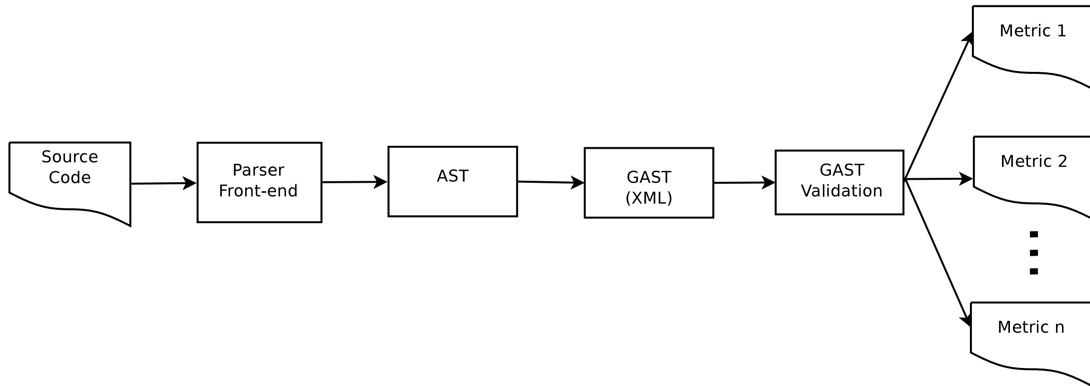

Figure 3.1: Overview of the GAST System

3.1 Overview of our Approach

Figure 3.1 contains an overview of the process that we use. First, the *source code* that is under review, shown on the left side of the figure, is used as input to a corresponding *Parser Front-end*, which builds a syntactic representation of the program called an Abstract Syntax Tree (AST). An AST is a pruned parse tree, but the AST might be decorated with semantic information to facilitate computation of the desired metrics. An AST is a language-dependent data structure so we extract the language specific attributes into a more generic representation that we refer to as a Generic Abstract Syntax Tree (GAST), shown in the middle of Figure 3.1. The GAST is expressed in the Extensible Markup Language (XML), as shown in the figure. We have developed a *schema*, described in Section 3.4 and shown in Listing 3.1, that we then use to validate the correctness of the generated GAST according to our definition of its structure. Finally, we compute the various software metrics on the GAST representation using an ordered tree traversal. Of course a set of metrics is required to provide a full evaluation and, in our final approach, we combine the scores to provide an overall evaluation of the source code.

3.2 Step 1: AST Construction

To build our AST representation, we leverage existing parser front-end APIs. In our implementation, we chose to support the Python and C++ programming languages using the Python *ast* module and Clang libraries, respectively [16, 23]. Python's built-in *ast* module provides several methods to tap into the Python parser internals, which we use to capture the program's AST. With slight modifications to the module's **ast.dump()** method, we can gather and format a Python program's AST. Similarly, we can use the LibClang library and Python wrappers to access the AST of a C++ program. Finally, we save the contents of the AST using the Extensible Markup Language (XML) to provide a standard and well-known output format.

Ultimately, this step allows us to gather the necessary syntactic and semantic information we need from the source code to perform our analysis. Rather than using the source code directly, we use an AST representation to provide us detailed information about the program that many software metrics may require. In addition, the existence of various parser front-end APIs for most languages makes this process adaptable to other programming languages [25].

3.3 Step 2: Generic AST Construction

The goal of this step is to use the XML representation of the AST generated in the previous step and remove all language-specific details to provide a common language AST for analysis. We call this modified AST a generic AST, or GAST. Once this step is completed, we have a new AST representation of the source code in XML that provides a common basis for code quality evaluation. This step facilitates

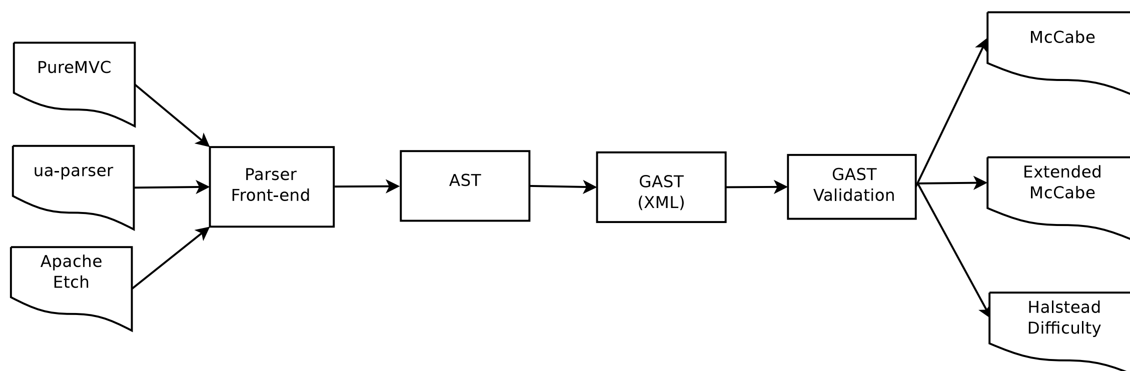

Figure 3.2: Code Evaluator Implementation

adaptation to a variety of programming languages.

3.4 Step 3: Validating Structural Correctness

Because the role of the GAST is to provide a common interface for evaluating code quality, it is important to explicitly define its structure to promote extensibility among other programming languages. In this way, researchers can apply our technique to other programming languages simply by providing its AST representation and a way to translate its entities according to the specification of the GAST. In Listing 3.1, we provide an XML Schema Definition to determine the validity of a generated GAST structure. In addition to allowing support for languages other than Python and C++, it is important that the output from Step 2 can be validated against our schema before proceeding to the code evaluation process in Step 4.

3.5 Step 4: Code Evaluator

Now that we have built a common format for problem analysis, we compute multiple software metrics using the GAST, and thereby measure the quality of the

code. In order to capture the code quality, we simply output the values returned by the cyclomatic complexity and Halstead difficulty metrics.

To investigate the practicality of our procedure, we computed the cyclomatic complexity using the GAST approach for three open source projects, each written in both C++ and Python. The three projects, PureMVC, ua-parser, and Apache Etch, are open source projects with their source code and language ports available on GitHub [1, 34, 41]. Figure 3.2 illustrates our code evaluator. Additionally, we aim to compare the values reported by cyclomatic complexity using our method against the known cyclomatic complexity measurement. The known cyclomatic complexity is obtained using the open source *metrics 0.2.6* software package written in 2010 to support calculation of metrics such as McCabe's cyclomatic complexity in C, C++, JavaScript, and Python programs [27]. Note that the *metrics* package does not calculate its included metrics using the approach proposed here, and instead relies on a more direct, text-based analysis of the source code.

However, the McCabe's cyclomatic complexity returned by the *metrics* package is defined differently than in our original implementation of the metric, which we have in turn adapted to match the software package's strategy. As mentioned in Chapter 2, the details on how to derive the control flow graph is vague on some details, and thus varies slightly from implementation to implementation [40]. Specifically, the McCabe metric used in the *metrics* package considers keywords like **assert**, **else**, **break**, and **continue** as branch statements, whereas our implementation does not. In order to distinguish these two, we continue to refer to the method proposed by the *metrics* package as McCabe's extended cyclomatic complexity, and our original implementation as simply McCabe's cyclomatic complexity. As a result, we calculated three different complexity measurements for each open source project: cyclomatic complexity, extended cyclomatic complexity, and the *metrics* tool cyclomatic com-

plexity.

After evaluating the accuracy of our approach, we aimed to determine its effectiveness through applying the McCabe cyclomatic complexity and Halstead difficulty metrics on several small-scale code exercises obtained from *Project Euler*. For each of the first five problems listed on the *Project Euler* problem archive, we obtain all user posted answers written in Python and C++, and then evaluate their quality using our approach and the Halstead difficulty, McCabe's cyclomatic complexity, and McCabe's extended cyclomatic complexity metrics. Finally, we again include the reported cyclomatic complexity from the *metrics* software package for verification.

3.6 Application: CodeR Online Evaluator

We apply the techniques described in this chapter to the CodeR online prototype [24]. CodeR is an online programming tool where users complete various programming tasks to learn the basics of either C++ or Python. Unlike other online programming instruction tools, we focus on providing the user feedback indicating the quality of his or her submitted code. Assuming they have answered the question correctly, their submission is then scored based on the Halstead and cyclomatic complexity metrics as mentioned earlier.

```
1 <?xml version="1.0"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4 <!-- simple element definitions -->
5 <xs:element name="TranslationUnit" type="CursorKind"/>
6
7 <!-- attribute definitions -->
8 <xs:attribute name="filename" type="xs:string"/>
9 <xs:attribute name="lineno" type="xs:positiveInteger"/>
10 <xs:attribute name="col_offset" type="xs:nonNegativeInteger"/>
11 <xs:attribute name="mccabe" type="xs:nonNegativeInteger"/>
12
13 <!-- GAST Cursor Kinds -->
```

```
14 <xs:complexType name="CursorKind">
15   <xs:choice minOccurs="0" maxOccurs="unbounded">
16     <xs:element ref="BinaryOperator" maxOccurs="unbounded"/>
17     <xs:element ref="UnaryOperator" maxOccurs="unbounded"/>
18     <xs:element ref="TernaryOperator" maxOccurs="unbounded"/>
19     <xs:element name="FunctionDefinition" type="CursorKind"
20       maxOccurs="unbounded"/>
21     <xs:element name="FunctionCallExpression" type="CursorKind"
22       maxOccurs="unbounded"/>
23     <xs:element name="ClassDefinition" type="CursorKind"
24       maxOccurs="unbounded"/>
25     <xs:element name="Constructor" type="CursorKind"
26       maxOccurs="unbounded"/>
27     <xs:element name="Assert" type="CursorKind" maxOccurs="unbounded"/>
28     <xs:element name="Num" type="CursorKind" maxOccurs="unbounded"/>
29     <xs:element name="Char" type="CursorKind" maxOccurs="unbounded"/>
30     <xs:element name="Str" type="CursorKind" maxOccurs="unbounded"/>
31     <xs:element name="Switch" type="CursorKind" maxOccurs="unbounded"/>
32     <xs:element name="While" type="CursorKind" maxOccurs="unbounded"/>
33     <xs:element name="For" type="CursorKind" maxOccurs="unbounded"/>
34     <xs:element name="If" type="CursorKind" maxOccurs="unbounded"/>
35     <xs:element name="Return" type="CursorKind" maxOccurs="unbounded"/>
36     <xs:element name="Case" type="CursorKind" maxOccurs="unbounded"/>
37     <xs:element name="Break" type="CursorKind" maxOccurs="unbounded"/>
38     <xs:element name="Continue" type="CursorKind" maxOccurs="unbounded"/>
39     <xs:element name="Try" type="CursorKind" maxOccurs="unbounded"/>
40     <xs:element name="Catch" type="CursorKind" maxOccurs="unbounded"/>
41     <xs:element name="Except" type="CursorKind" maxOccurs="unbounded"/>
42     <xs:element name="Unknown" type="CursorKind" maxOccurs="unbounded"/>
43   </xs:choice>
44   <xs:attribute ref="lineno"/>
45   <xs:attribute ref="col_offset"/>
46   <xs:attribute ref="mccabe" use="optional"/>
47 </xs:complexType>
48
49 <!-- complex element definitions -->
50 <xs:element name="BinaryOperatorMeta">
51   <xs:complexType>
52     <xs:choice minOccurs="0" maxOccurs="unbounded">
53       <xs:element name="rand" type="xs:string" maxOccurs="unbounded"/>
54       <xs:element name="tor" type="xs:string" maxOccurs="unbounded"/>
55     </xs:choice>
56   </xs:complexType>
57 </xs:element>
58
59 <xs:element name="BinaryOperator">
60   <xs:complexType>
61     <xs:complexContent>
62       <xs:extension base="CursorKind">
63         <xs:sequence>
64           <xs:element ref="BinaryOperatorMeta"/>
```

```
65        </xs:sequence>
66      </xs:extension>
67    </xs:complexContent>
68  </xs:complexType>
69 </xs:element>
70
71 <xs:element name="UnaryOperatorMeta">
72   <xs:complexType>
73     <xs:choice minOccurs="0" maxOccurs="unbounded">
74       <xs:element name="rand" type="xs:string" maxOccurs="unbounded"/>
75       <xs:element name="tor" type="xs:string" maxOccurs="unbounded"/>
76     </xs:choice>
77   </xs:complexType>
78 </xs:element>
79
80 <xs:element name="UnaryOperator">
81   <xs:complexType>
82     <xs:complexContent>
83       <xs:extension base="CursorKind">
84         <xs:sequence>
85           <xs:element ref="UnaryOperatorMeta"/>
86         </xs:sequence>
87       </xs:extension>
88     </xs:complexContent>
89   </xs:complexType>
90 </xs:element>
91
92 <xs:element name="TernaryOperatorMeta">
93   <xs:complexType>
94     <xs:choice minOccurs="0" maxOccurs="unbounded">
95       <xs:element name="rand" type="xs:string" maxOccurs="unbounded"/>
96       <xs:element name="tor" type="xs:string" maxOccurs="unbounded"/>
97     </xs:choice>
98   </xs:complexType>
99 </xs:element>
100
101 <xs:element name="TernaryOperator">
102   <xs:complexType>
103     <xs:complexContent>
104       <xs:extension base="CursorKind">
105         <xs:sequence>
106           <xs:element ref="TernaryOperatorMeta"/>
107         </xs:sequence>
108       </xs:extension>
109     </xs:complexContent>
110   </xs:complexType>
111 </xs:element>
112
113 <!-- root definition -->
114 <xs:element name="gast">
115   <xs:complexType>
```

```
116     <xs:sequence>
117       <xs:element ref="TranslationUnit"/>
118     </xs:sequence>
119     <xs:attribute ref="filename" use="required"/>
120   </xs:complexType>
121 </xs:element>
122
123 </xs:schema>
```

Listing 3.1: GAST XML Schema Definition

Chapter 4

Results

In previous chapters, we described our goal of developing a generic program representation that can be used for evaluating programs written in a multitude of popular programming languages. We have also described our approach for achieving this goal: the construction of a generic abstract syntax tree (GAST) that can capture the required information for automated metrics computation for each of the programming languages under study. In this chapter, we present some results from the implementation of our approach in an effort to evaluate the feasibility and utility of our goal.

In Section 4.1, we apply our approach to three medium sized, commonly used, open source programs: PureMVC, ua-parser, and Apache Etch, and we compute metrics for each of these applications [1, 34, 41]. The PureMVC framework facilitates writing applications based on the Model-View-Controller architectural pattern [34]. PureMVC implements several design patterns defined by Gamma et al. [17], including Facade, Command, Mediator, Observer, and Proxy. Versions of the framework are available for multiple languages, including Python and C++. In addition to its focus on design patterns, PureMVC also makes liberal use of template features. We also applied

our approach to ua-parser, a parser that can extract information about the platform, operating system, and cpu architecture, from a user-agent string, with relatively lightweight footprint [41]. The ua-parser is a commonly used application that has been ported to multiple programming languages including Python and C++. In addition, we applied our metrics to Apache Etch, an open source, cross-platform, language- and transport-independent framework for building and consuming network services [1]. We chose these applications because they are medium sized, commonly used, open source projects that include versions programmed in both Python and C++, the two languages that we wish to target for our evaluation.

The goals of our work also include the development of automated techniques to facilitate improvement of programming skills for students who major in both non computer science and computer science areas. There are a plethora of projects that either provide instruction in computer programming or test programs for validity, including *Project Euler*, *Codecademy*, and *Code Wars* [13, 7, 9]. However, there are no projects available that assess the quality of the student code and, in our experience, the quality can vary greatly. Thus, in Section 4.2 we describe some results obtained through our evaluation of the first five exercises found on the *Project Euler* web site. We captured all of the solutions posted for these first five exercises and used our metrics computation to evaluate the solutions. Finally, we have developed a web-based program evaluator, CodeR, described in Section 4.3, that permits users to enter programs, written in either Python or C++, for evaluation using the metrics that we have implemented. To illustrate the use of the web page in an online coding environment, we provide screen captures of the CodeR prototype.

Project	Language	SLOC	McCabe	Ext McCabe	Tool
ua-parser	C++	681	94	102	102
ua-parser	Python	542	91	106	106
PureMVC	C++	1991	154	162	162
PureMVC	Python	413	34	35	35
Apache Etch	C++	21125	1106	1302	1302
Apache Etch	Python	5023	377	422	422

Table 4.1: McCabe Metrics for PureMVC, ua-parser, and Apache Etch

4.1 Evaluation of Open Source Projects

To evaluate our generic metric computation for medium sized programs, we now describe our results for the three open source applications: PureMVC, ua-parser, and Apache Etch. Table 4.1 consists of of six columns and six rows of data. The first column lists the name of the project under study, the second column lists the implementation language for the respective project, and the third column lists the total *lines of code* required to implement the project, excluding comments and whitespace. Columns four and five list results obtained using our generic approach for computing McCabe's cyclomatic complexity, and for computing the extended McCabe's cyclomatic complexity. The sixth column of the table lists results obtained using an open-source metrics computation tool to compute McCabe's cyclomatic complexity. The values listed for cyclomatic complexity reflect the sum of the complexities of all of the functions in the application.

The fifth and sixth columns of Table 4.1 show that our generic computation tool that computes the extended McCabe cyclomatic complexity and the open-source metrics computation tool compute the exact same values, providing some confidence

in the validity of our results. Also, for the ua-parser application we observe that the C++ and Python versions consist of approximately the same number of SLOC, listed as 681 and 542 in column three, rows three and four. Similarly, the McCabe cyclomatic complexities of the ua-parser application are approximately the same for the C++ and Python versions, listed as 94 and 91 respectively.

However, the values for the PureMVC and Apache Etch applications list divergent results for the C++ and Python versions, where the values listed for the Python version of the application are consistently lower than the C++ version. For example, consider column three, rows three and four where the C++ and Python versions are listed as 1991 and 413 SLOC respectively. This reduction in SLOC for the Python version reflects the fact that the Python versions of PureMVC and Apache Etch are written in a *pythonic* manner, which is typically more succinct than the usual C++ coding style, which tends to be more verbose with separate files for the interface and implementation of classes. There are similar reductions in the results for McCabe's cyclomatic complexity where, for example, the McCabe values for C++ and Python are listed for PureMVC in column four as 154 and 34 respectively, illustrating the advantage of using Python for rapid prototyping.

4.2 Evaluation of Solutions for Project Euler

To further evaluate our generic metrics computation, we use our approach to evaluate the user solutions, written in C++ and Python, posted on the *Project Euler* website. Table 4.2 lists these first five exercises in two columns, where the first column lists a *tag* that we will use to refer to the exercise, and the second column lists the *full description* of the problem. For example, the second row of Table 4.2 lists the **Tag**, Even Fibonacci, and the **Full Description**, which asks the user to find the sum

Tag	Full Description
1. Multiples of 3 and 5	If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.
2. Even Fibonacci	Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.
3. Largest Prime Factor	The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143?
4. Largest Palindrome	A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 9199. Find the largest palindrome made from the product of two 3-digit numbers.
5. Smallest Multiple	2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder. What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

Table 4.2: Descriptions of the First Five Exercises for Project Euler

of the even Fibonacci numbers that are less than four million.

Tables 4.3 and 4.4 list the results of our metric evaluation for C++ and Python, respectively. For example, Table 4.3 contains five rows of data, one row for each of the five Euler exercises, and 11 columns. The first column lists the **Tag**, described previously, and the second column lists the number of solutions that were posted in C++ for each of the five Euler exercises. For example, the most solutions written in C++ were posted for the Largest Palindrome exercise, where 19 solutions were posted on the Euler website. The final nine columns list High, Low, and Avg values for each of the three metrics: Halstead, McCabe, and Ext McCabe.

Tag	**Solns**	**Halstead**			**McCabe**			**Ext McCabe**		
		High	Low	Avg	High	Low	Avg	High	Low	Avg
1. Multiples of 3 and 5	14	9.00	3.00	5.57	4.00	0.00	2.21	4.00	0.00	2.36
2. Even Fibonacci	17	10.00	3.00	5.53	5.00	0.00	2.59	5.00	0.00	2.76
3. Largest Prime Factor	11	14.00	3.00	9.59	11.00	2.00	6.36	15.00	2.00	7.09
4. Largest Palindrome	19	30.00	4.00	13.03	10.00	3.00	5.68	11.00	3.00	5.89
5. Smallest Multiple	16	10.00	2.00	5.44	14.00	1.00	4.38	16.00	1.00	4.69

Table 4.3: Posted Solutions Written in C++

Tag	Solns	Halstead			McCabe			Ext McCabe		
		High	Low	Avg	High	Low	Avg	High	Low	Avg
1. Multiples of 3 and 5	33	5.67	0.50	2.65	6.00	0.00	2.00	8.00	0.00	2.12
2. Even Fibonacci	34	5.71	1.00	2.72	6.00	1.00	2.26	6.00	1.00	2.41
3. Largest Prime Factor	46	8.40	1.00	4.44	10.00	2.00	4.30	12.00	2.00	5.00
4. Largest Palindrome	40	9.15	1.00	3.49	17.00	3.00	5.23	18.00	3.00	5.65
5. Smallest Multiple	26	7.80	1.80	4.26	20.00	1.00	5.92	16.00	1.00	6.85

Table 4.4: Posted Solutions Written in Python

For example, the values listed for the Halstead metric for the Multiples of 3 and 5 exercise are 9, 3, and 5.57 for the high, low, and average values. However, the most widely divergent values are those listed for the Largest Palindrome exercise, with values of 30, 4, and 13.03 for the high, low, and average values. These values illustrate the widely divergent metrics computed for the posted solutions to the Euler exercises. Even more interesting are the values listed for the McCabe metric for the first two exercises, where the low values for Multiples of 3 and 5 and Even Fibonacci are listed as zero for the low values. These zero values result from the fact that some of the posted solutions used mathematic formulae to compute their results and did not require loops or decision statements.

Recall that the extended McCabe complexity, Ext McCabe, extends the McCabe keywords with **assert**, **else**, **break**, and **continue**. Thus, the values listed for Ext McCabe are the same or higher than the values listed for McCabe. For example, the average value listed for the McCabe metric for the Even Fibonacci exercise is 2.59, while the average value listed for Ext McCabe metric for the Even Fibonacci exercise is 2.76, even though the high and low values are the same: 5 and 0 respectively.

Table 4.4 lists the metrics computations for solutions written in Python for the first five Euler exercises. Table 4.3 makes an interesting comparison with Table 4.4, where the solutions are instead written in C++. For example, there are more posted solutions in Python, with 179 solutions, than for C++, with 77 solutions. Also, the

Halstead metrics listed for the Python solutions are consistently lower than the values listed for the C++ solutions.

However, the results for the McCabe metrics list average values that are generally lower for the Python solutions, but the high values are generally higher for the Python solutions. For example, the average McCabe values for the C++ solutions are 2.21, 2.59, 6.36, 5.68, and 4.38; however, the average McCabe values for the Python solutions are 2.00, 2.26, 4.30, 5.23, and 5.92. These values illustrate that the Python solutions generally have lower average values than the C++ solutions, except for the Smallest Multiple exercise, where the average value for the C++ solution is 4.38, as compared to 5.92. However, the high values listed for the McCabe metric are generally higher for the Python solutions. For example, the high value for the Smallest Multiple exercise is 14.00 for the C++ solutions, but is 20.00 for the Python solutions. One explanation for these larger High values is that some of the Python solutions were submitted by beginning programmers, and this explanation is substantiated by some of the comments posted on the Euler website.

The results for the McCabe and Halstead metrics are further illustrated in Figures 4.1 and 4.2, respectively. Both figures attempt to highlight the differences in metric evaluation scores for all C++ and Python user solutions among the first five *Project Euler* problems. A similar boxplot visualization can be found in Appendix A for the extended McCabe metric calculations.

In Figure 4.1, the McCabe metric calculations for both C++ and Python user solutions depict similar patterns of high and low complexity values across the five problem sets. This likely reflects the varying difficulty of the *Project Euler* challenge problems, although they share a common user difficulty rating of 5% as listed on the website's Problem Archive [13]. Figure 4.1 also shows a large number of outliers, which further demonstrates the wide variety of user programming experience and

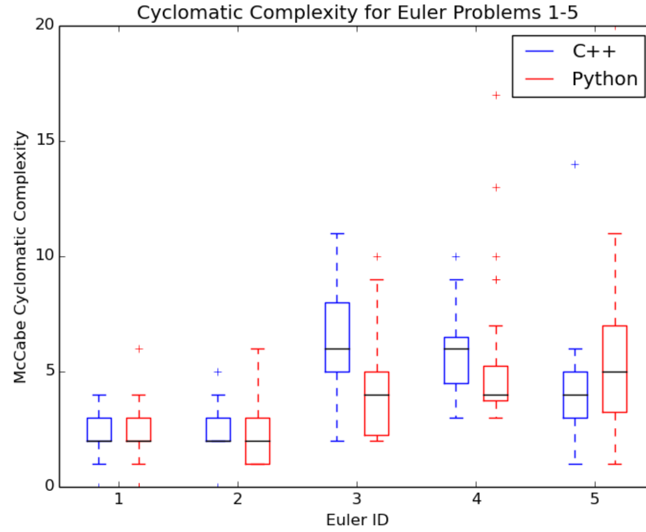

Figure 4.1: Cyclomatic Complexity for Euler Problems 1-5

posted solutions in both languages.

Figure 4.2 depicts a noticeable difference in Halstead difficulty measurements between C++ and Python user solutions. When considering the calculation of the McCabe cyclomatic complexity, the types of control statements share a similar structure in both C++ and Python programming languages. In comparison, the types and use of operators and operands when computing the Halstead difficulty measurement is largely influenced by the underlying grammar and intent of the programming language. Therefore, the succinct nature of the Python programming language may contribute to smaller Halstead Difficulty values as shown in Figure 4.2.

4.3 CodeR Results

Figures 4.3 and 4.4 show screenshots of an online code instruction tool prototype, CodeR, which performs the metric evaluation that we describe in this paper

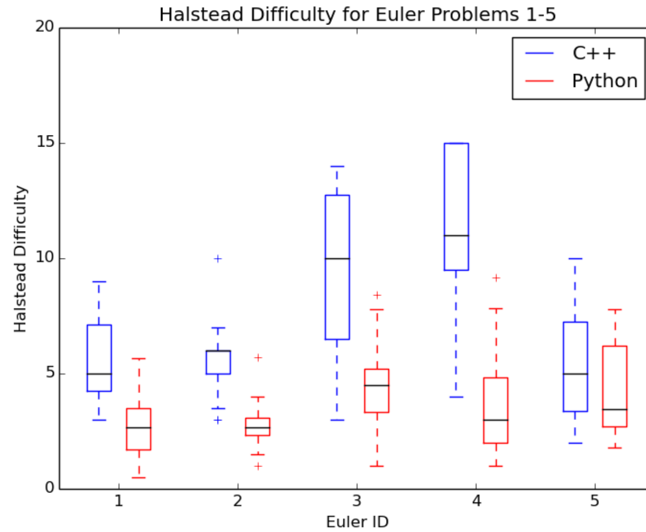

Figure 4.2: Halstead Difficulty for Euler Problems 1-5

[24]. Users can create an account and select from a sequence of challenge problems, including the first five *Project Euler* exercises. They can then write and execute code in the text editor to submit their attempted solution in either Python or C++. The submitted solution is evaluated for correctness by comparing the output of the submitted solution, which is shown in the console window, with the correct response that is stored in our database. Clicking on the *Evaluate* key will first verify that the submitted solution is valid, and then continue to give a detailed report on the cyclomatic complexity and Halstead difficulty scores for the submitted solution. Also, since students can better evaluate their performance by comparing their scores with the scores of other submissions, CodeR provides the user's best attempted scores for each exercise, as well as the average, minimum, and maximum scores of other students [38].

For example, Figure 4.3 illustrates a CodeR web page where a user has chosen to solve the Multiples of 3 and 5 exercise in Python, and has submitted their solution in

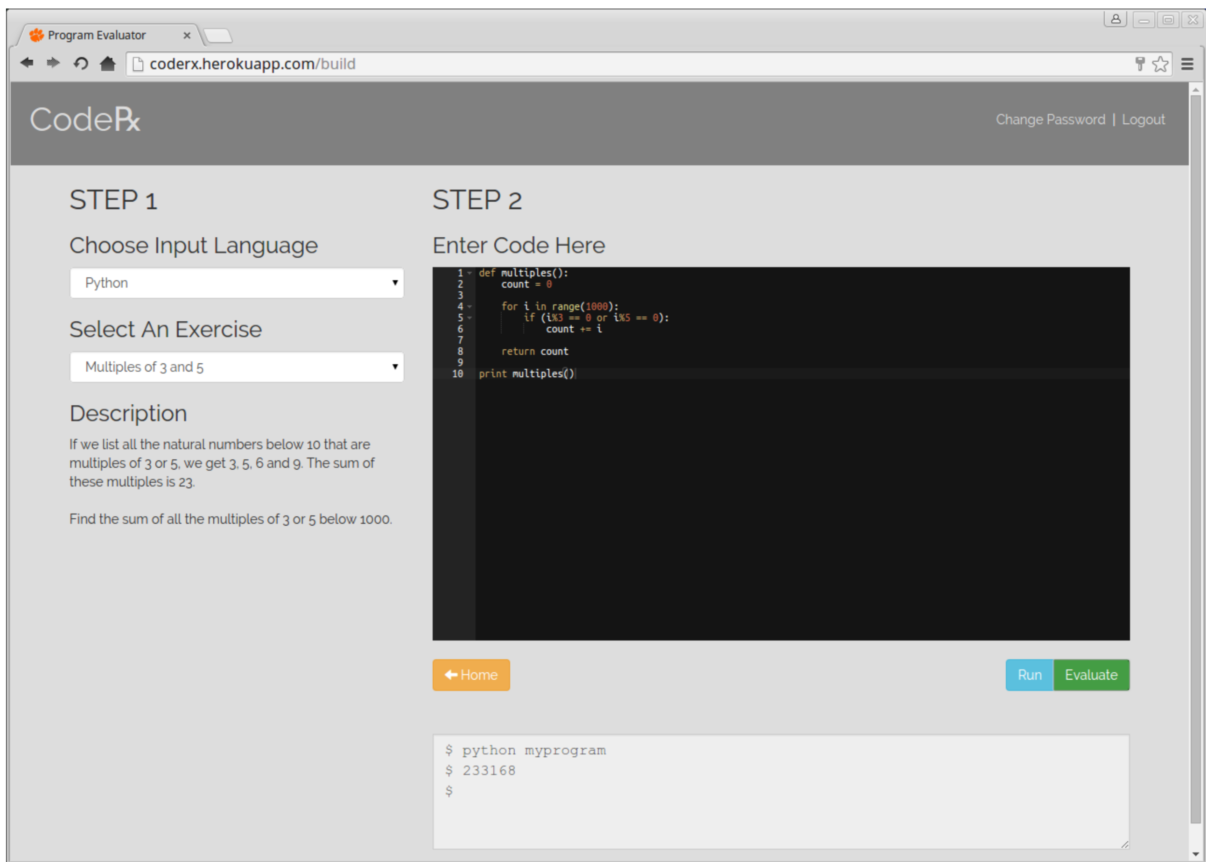

Figure 4.3: CodeRx Prototype

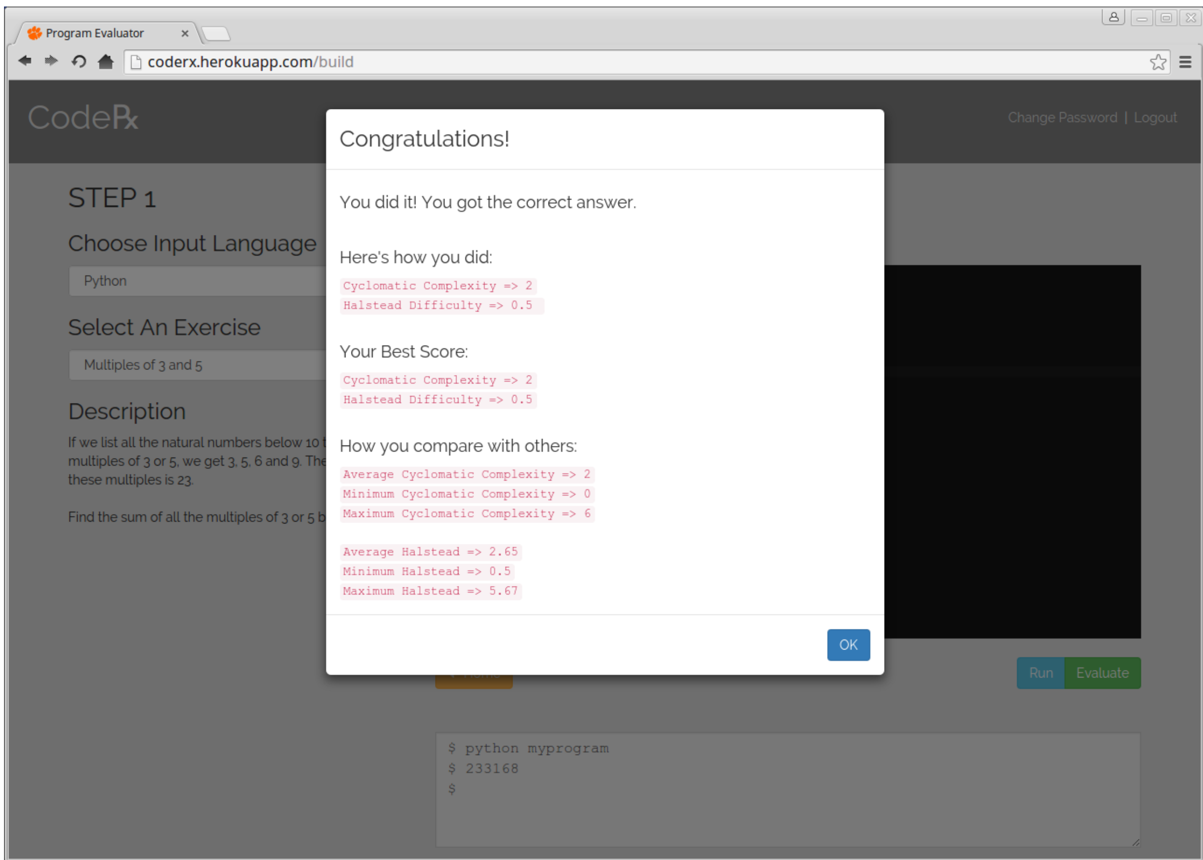

Figure 4.4: CoderR Results

the Enter Code Here window. Also, the bottom of the figure illustrates the output from the execution of the student submission, where the answer shown is 233168, which is the correct answer. Since the answer is correct, CodeR then provides comparison scores in Figure 4.4, where first the Cyclomatic Complexity and Halstead Difficulty are shown for the submitted solution as 2 and 0.5 respectively. The middle of the figure shows the student's best scores for this problem, also as 2 for Cyclomatic Complexity, and 0.5 for Halstead Difficulty. Finally, Figure 4.4 compares the submitted score with the scores of other students who have submitted solutions; these comparison scores are shown at the bottom of the figure.

Chapter 5

Conclusion and Future Work

In this thesis, we have described our strategy for developing a generic program representation that can be used for evaluating programs written in a multitude of popular programming languages. Our approach entails the use of existing parser tools to build an AST and, in our evaluation, we used the LibClang and Python *ast* front-end APIs. We described our use of the AST in the construction of a generic abstract syntax tree (GAST) that captures the required information for automated metrics computation for each of the programming languages under study. In our evaluation, we applied our approach to three medium sized, commonly used, open source programs: PureMVC, ua-parser, and Apache Etch, and we computed metrics for each of these applications [1, 34, 41]. We chose these applications because they are medium sized, commonly used, open source projects that include versions programmed in both Python and C++, the two languages that we target in our evaluation.

Since the goals of our work also include the development of automated techniques to facilitate improvement of programming skills for students who major in both non computer science and computer science areas, we also described results obtained from our evaluation of the first five exercises found on the *Project Euler* web site. We

captured all of the solutions written in Python and C++ posted for these first five exercises and used our metrics computation to evaluate the solutions.

Finally, we have developed a web-based program evaluator, CodeR, that permits users to enter programs, written in either Python or C++, for evaluation using the McCabe cyclomatic complexity and Halstead difficulty metrics. These metrics together provide a general indication of the quality of a program's code.

The contributions of our work include:

1. A generic representation of a program that is general and interoperable among programming languages. The GAST that we have developed can be used by other researchers to reproduce the metric results that we have computed, or the researchers can use the GAST to compute metrics of their choosing. This interoperability is facilitated by our representation of the GAST in XML, which is human readable, platform independent, and frequently used, so that there is likely to be an available XML parser coded in the same language used for project development. This interoperability is facilitated further through the use of the schema that we have developed, which captures the important constraints needed for metrics computation using the GAST.
2. An implementation, using two popular programming languages, Python and C++, which use the GAST to compute metrics on our test suite. We have validated some of our results through a comparison of the McCabe cyclomatic complexity values computed with an existing metric computation tool.
3. A web site, CodeR, which supports evaluation of the quality of student program submissions. The current implementation of the web page permits students to submit any program written in Python or C++, and also allows students to solve any of the first five *Project Euler* exercises, including a comparison of the quality

of the student submission with the quality of previous student submissions.

5.1 Future Work

Currently, our framework allows for metric evaluation for the C++ and Python programming languages. However, the GAST framework is designed to support any programming language with access to an AST representation. In addition, only the McCabe cyclomatic complexity and Halstead difficulty metrics were considered in our quality evaluation scheme. In order to increase the utility of our tool across many problem domains, the framework can be extended to cover a larger variety of programming language and metrics to facilitate a more wide appeal and utility.

5.2 Future of Online Code Instruction Tools

In order to engage students in computer science education, it is important to create a welcoming environment for educational innovation. To facilitate this welcoming environment, the web site might be extended to permit student programmers to submit additional problems to be solved, and to permit students to submit their solution for evaluation by other students and teachers.

Appendices

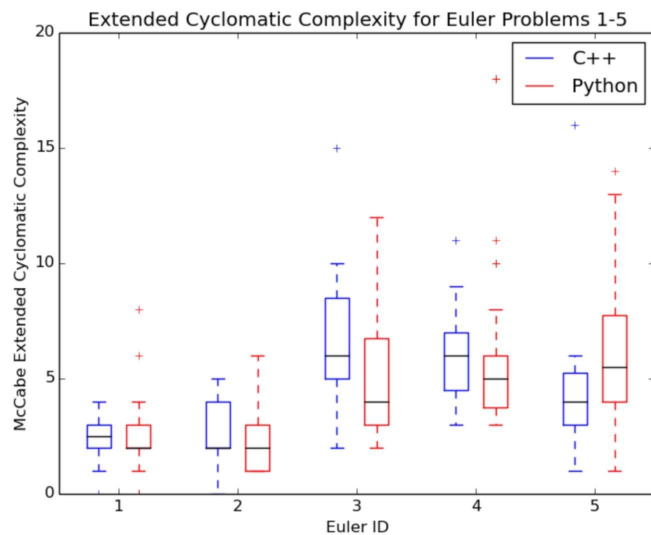

Figure 1: Extended Cyclomatic Complexity for Euler Problems 1-5

Appendix A Results for Extended McCabe Cyclomatic Complexity

References

[1] Apache. Apache Etch. https://etch.apache.org/, 2016.

[2] L.S. Bacow, W.G. Bowen, K.M. Guthrie, K.A. Lack, and M.P. Long. Barriers to adoption of online learning systems in us higher education.

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, November 1998.

[4] Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[5] H.R. Bhatti. Automatic measurement of source code complexity. *Luleå University of Technology*, 2010.

[6] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering.* Springer Science+Business Media, 2000.

[7] Codecademy. Codecademy: Learn to code. https://www.codecademy.com/.

[8] Code.org. Code.org: Anybody can learn. https://code.org/, 2015.

[9] Codewars. Codewars. http://www.codewars.com/.

[10] S. Datta. *Metrics-Driven Enterprise Software Development.* J. Ross Publishing, 2007.

[11] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 208–212, New York, NY, USA, 2011. ACM.

[12] E.B. Duffy, B.A. Malloy, and S. Schaub. Exploiting the Clang AST for analysis of C++ applications. In *The 52nd Annual ACM Southeast Regional Conference*, March 2014.

[13] Project Euler. Project euler.net. https://projecteuler.net/.

[14] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, July 1999.

[15] Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8):455–460, 1979.

[16] Python Software Foundation. ast - Abstract Syntax Trees. https://docs.python.org/2/library/ast.html, 2016.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] M.H. Halstead. *Elements of Software Science*. Amsterdam: Elsevier North-Holland, Inc., 1977.

[19] B. Hummel. Mccabe's cyclomatic complexity and why we don't use it. https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/, May 2016.

[20] C. Jones. Software metrics: good, bad and missing. *IEEE Computer*, 27(9):98–100, September 1994.

[21] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, 2005.

[22] LLVM. Clang: A C language family frontend for llvm. http://clang.llvm.org/, 2016.

[23] LLVM. Clang python bindings. https://github.com/llvm-mirror/clang/tree/master/bindings/python, 2016.

[24] B.A. Malloy and Z.S. McNellis. CodeR. http://coderx.herokuapp.com.

[25] B.A. Malloy and S. Schaub. Comprehensive analysis of C++ applications using the libclang api. In *Software Engineering and Data Engineering*, October 2014.

[26] T.J. McCabe. A complexity measure. *IEEE TSE*, 2(4):308–320, December 1976.

[27] metrics. metrics 0.2.6. https://pypi.python.org/pypi/metrics.

[28] Andreas Leon Aagaard Moth, Joergen Villadsen, and Mordechai Ben-Ari. Syntaxtrain: relieving the pain of learning syntax. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 387–387, New York, NY, USA, 2011. ACM.

[29] U.S. Bureau of Labor Statistics. Occupational outlook handbook. http://www.bls.gov/ooh/, December 2015.

[30] Massachusetts Institute of Technology. Mit online education policy initiative. https://oepi.mit.edu/.

[31] U.S. Department of Transportation Federal Aviation Administration. Software quality metrics. August 1991.

[32] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, December 2007.

[33] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference*, June 2001.

[34] PureMVC. Puremvc. http://puremvc.org/, [Online; accessed 15-January-2016].

[35] J. Rosenberg. Some misconceptions about lines of code. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, November 1997.

[36] D. Samadhiya, S. Wang, and D. Chen. Quality models: Role and value in software engineering. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, October 2010.

[37] A. Schäfer, J. Holz, T. Leonhardt, U. Schroeder, P. Brauner, and M. Ziefle. From boring to scoring a collaborative serious game for learning and practicing mathematical logic for computer science education. *Computer Science Education*, 23(2):87–111, 2013.

[38] P. Wesley Schultz1, Jessica M. Nolan2, Robert B. Cialdini3, Noah J. Goldstein3, and Vladas Griskevicius3. The constructive, destructive, and reconstructive power of social norms. In *Psychological Science: A journal of the Association of Psychological Science*, May 2016.

[39] Judy Sheard, S. Simon, Margaret Hamilton, and Jan Lönnberg. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop*, ICER '09, pages 93–104, New York, NY, USA, 2009. ACM.

[40] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, March 1988.

[41] ua parser. ua-parser: Community driver user agent string parser. http://www.uaparser.org/.