

# **Semi-Automated Software Restructuring**

**By**

**Santosh K Singh Kesar**

**Thesis submitted in partial fulfillment of the requirements for the**

**Degree of Master of Science in Computer Engineering**

**Advisor:**

**Dr. James Fawcett**

**DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER**

**SCIENCE**

**SYRACUSE UNIVERSITY**

**October 2008**

**Syracuse, New York**

# Table of Contents

Table of Figures and Tables .....	5
Chapter 1 Introduction .....	7
1.1 Motivation.....	7
1.2 Goals .....	11
1.3 Research Statement.....	12
1.3.1 Code Restructuring versus Refactoring .....	15
1.4 Related Work .....	16
1.4.2 Code Rearrangements during Compiler Optimization .....	17
1.4.3 Code Restructuring in Hardware Platforms.....	19
1.4.4 Table-driven Lexical Analysis.....	20
Chapter 2 - Restructuring Techniques .....	24
2.1 Syntax of Scopes and Data Declarations .....	25
2.1.1 Feasible Regions based on Control Scopes .....	26
2.1.2 Inclusion of Local Data Constraints .....	28
2.2 Extraction of Functions.....	30
2.2.1 Extracting functions with no parameters .....	30
2.2.2 Extracting functions with parameter passing.....	31
2.3 Extraction of Methods.....	33
2.3.1 Transforming local data into member data .....	36
2.4 Capturing Feasible Regions .....	38
2.5 Passing Parameters.....	39
2.6 Deciding which code to Extract.....	39

2.7 Our Lexical Analysis Tools .....	40
2.8 Grammar Construction and Parsing.....	44
2.7 Summary.....	47
Chapter 3 Code Analysis .....	48
3.1 Analysis.....	48
3.1.1 Parsing Scopes and Saving Scope information.....	51
3.1.2 Grammar Detectors.....	54
3.1.3 Parse Tree Construction.....	55
3.1.4 Parsing Data Declarations and Saving Data Spans.....	59
3.1.5 Selecting Feasible Regions .....	62
3.2 Implementation .....	66
3.2.1 Implementing Grammar Detectors .....	66
3.2.2 Implementing Parse Tree .....	68
3.2.3 Implementing Code Restructuring.....	74
Chapter 4 Semi-Automated Restructuring Results .....	78
4.1 Restructuring Functions .....	78
4.1.1 Extracting functions with no parameters .....	79
4.1.2 Extracting functions with one parameter .....	81
4.1.3 Extracting functions with many parameters .....	86
4.2 Restructuring Methods.....	89
4.2.1 Changes to Method Declared File.....	92
4.2.2 Changes to Method Defined File .....	93
4.2.3 Transforming local data into member data .....	94

4.3 Restructuring functions in multiple passes .....	95
Chapter 5 Contributions, Conclusions, and Future Work.....	99
5.1 Reviewing Research Statement.....	99
5.2 Contributions.....	100
5.2.1 Accomplished Work .....	101
5.3 Future Work .....	103
Appendix.....	105
A.1. Large Method.....	105
A.1.1. Restructured Method in First Pass .....	108
A.1.2 Restructured Method in Second Pass.....	111
Bibliography .....	114

# Table of Figures and Tables

Figure 1.1- Large Sized Functions in imaging research <sup>[2]</sup> source code .....	8
Figure 1.2 - Internal and External Dependencies in GKGFX Library, Mozilla 1.4.1 .....	10
Figure 2.1 – Syntax of Scopes .....	25
Figure 2.2 – User Entered Constraints .....	40
Figure 2.3 – Sample output from Tokenizer Module .....	42
Figure 2.4 – Sample output from Semi-Expressions Module.....	43
Figure 2.5 – Top Level Structure of Parse Tree.....	45
Figure 3.1 - Different Levels in Parse Tree .....	49
Figure 3.2 – Hypothetical view of Hierarchy Stack .....	53
Figure 3.3 – Different types of Nodes in Parse Tree .....	56
Figure 3.4 – Building of First Three Levels .....	57
Figure 3.5 – Top Level Containment diagram of Parse Tree .....	61
Figure 3.6 – Line number criteria for Feasible Regions .....	62
Figure 3.7 – Top down approach for determining parameters.....	64
Figure 3.8 – Bottom up approach for determining parameters.....	65
Figure 3.9 – Two-Way approach for determining parameters.....	65
Figure 3.10 – Class Diagram of Parsers using Utility Class.....	67
Figure 3.11 – Class Diagram of ICRNode Interface.....	68
Figure 3.12 - Class Diagram of RootObj .....	69
Figure 3.13– Class Relationship diagram of Parse tree Objects.....	70
Figure 3.14 – Class Diagram of Different Node Types .....	71
Figure 3.15 – Class Diagram of DataObject.....	73

Figure 3.16 – Class Diagram of TempContainer .....	74
Figure 3.17 – Class Relationship diagram of feasibleRegions and newFunctions .....	75
Figure 3.18 – Class Relationship diagram of FunctionParser and fileManager .....	76
Figure 4.1 – Restructuring source code in multiple passes.....	98

# Chapter 1 Introduction

## ***1.1 Motivation***

Software code structure is concerned with partitioning and organizing software source code in a logical fashion. This may be carried out by creating a top level partitioning of software into modules, and, in each module, separating processing into classes and, if the language supports it, global functions. Careful partitioning of software components makes development, testing and verification processes more effective.

Software Components are tested for correctness of functionality in unit tests, for compatibility with other project software in integration tests, and demonstrate that they meet their specified obligations in qualification tests. Software Development and testing becomes easier when working with well structured components and source code functions. It's hard to understand and correctly make changes to large source code files, especially if there are many files with large functions and no self documentation. It takes a lot of time and effort to trace through large functions to understand their functionality. This is especially so if there are many variables of interest whose changes need to be traced while the code is running. . It is quicker and easier to understand, debug, trace, and test the functionality of small components.

Maintenance and support is an ongoing process in software development once a product is released and in use. Reviews and requests from clients and their users may require changes to the source code, or verification that an observed behavior is correct. If source code is lengthy and hard to understand, it is more likely that new errors may be inadvertently added to the software during these maintenance activities. Considering all these issues,

software structures, and size of source code components, such as functions, play an important role in development and maintenance activities. .

An obvious question is: “Is software developed by researchers and professional developers likely to have structural problems”. Is software badly structured often enough to warrant research on ways to improve that?

Below is the summary of large function sizes of test<sup>[2]</sup> source code used at a facility<sup>[3]</sup> for medical imaging research.

File	Function Name	Number of lines
Weights_calculation.cpp	WeightTwoQuadrantsFactor	280
Weights_calculation.cpp	FactorsTwoRays	206
Weights_calculation.cpp	AreaWeightFactor	223
Weights_calculation.cpp	W_Calculate	377
Weights_calculation.cpp	Main	191
Weights_calculation.cpp	WeightBottomFactor	164
Mlr800fs.c	Emsid2_new	724
Mlr800fs.c	Main	851
Mlr800fs.c	Ect	<b>3608</b>
Mlr800fs.c	emsid3_new	<b>749</b>
Mlr800fs.c	emsid4	<b>516</b>

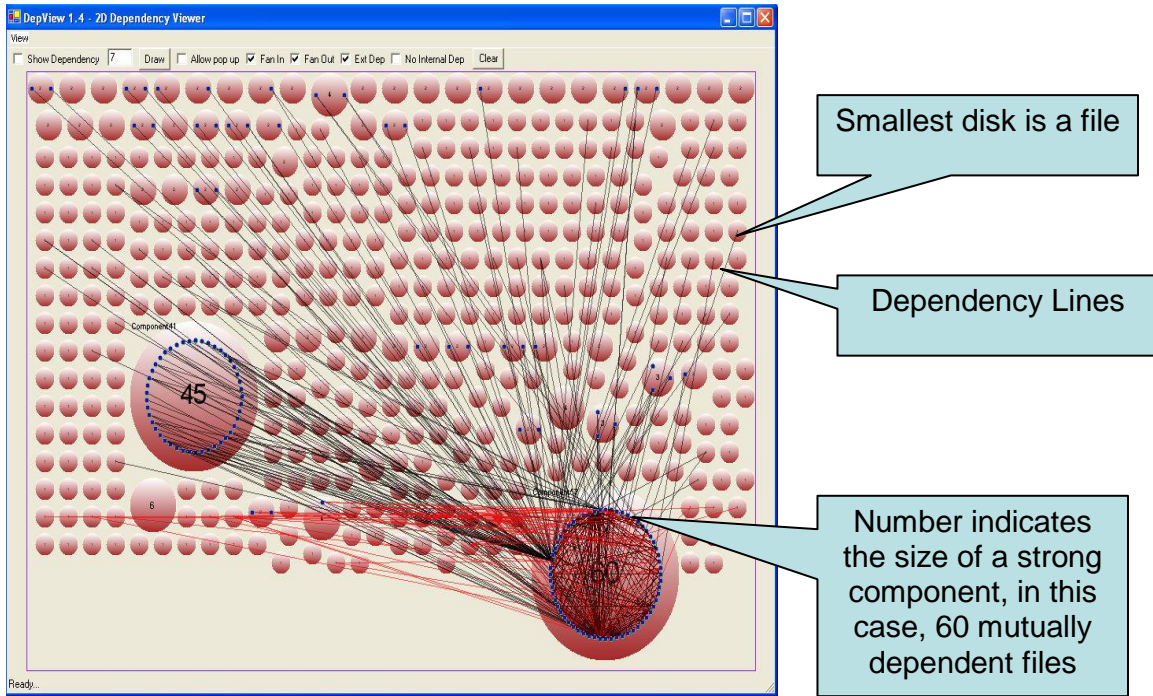
**Figure 1.1- Large Sized Functions in imaging research <sup>[2]</sup> source code**

An interesting analysis was performed on Medical Image Processing software code<sup>[2]</sup> that is in use at a facility<sup>[3]</sup>. This application is comprised of large source code files. For example, Referring to the figure 1.1, the file ‘mlr800fs.c’ has a main function with 851 lines, function



'ect' with 3608 lines, function 'emsid2\_new' with 724 lines, function 'emsid3\_new' with 749 lines, and function 'emsid4' with 516 lines. Given this application to test, debug or make enhancement. It's a daunting task to make any changes - very time consuming and prone to errors.

Another interesting analysis that was performed on an open source browser, Mozilla version 1.4.1 in the study Structural Models for large Software Systems<sup>[1]</sup>. The software system was composed of 6,193 files. To understand the complexity of these large files, a type based file-to-file dependency analysis was performed on one of the Mozilla libraries, GKAFX, responsible for browser rendering, with 598 files. The analysis determined that the GKAFX library, and many other libraries in that system, as well, contained large mutually dependent file sets. Also, Fan-in – The number of files dependent on a given file, and fan-out – The number of files a given file depends on were evaluated and found to be, in some cases, exceedingly large.



**Figure 1.2 - Internal and External Dependencies in GKGFX Library, Mozilla 1.4.1**

The above figure represents dependency relationships among files in the Mozilla GKGFX library. In this figure, the smallest disk represents individual source files; all larger disks represent strong components – set of mutually dependent files. The number at the center of each circle indicates the size of a strong component – the number of files. A line between circles shows dependency among files.

The above figure also shows internal and external dependency of the largest strong component within the GKGFX library. This figure reveals that the strong component uses services of many individual files and members of other strong components. In addition, the figure adds dependencies on files, outside the same strong component, on files inside, indicating services it provides to these files.

The above analysis reveals important issues with respect to development and maintenance of large software systems in general. If dependencies between sections of

source code are dense, that results in undesirable tight coupling between files. Changes made to one file of a strong component section in the source code would affect all the other files in that section. In making changes, one needs to know how the changes would affect every one of them. Large functions and files would add more difficulty understanding the dependencies and functionality of such source code.

In this section, we have discussed the analysis two software applications, one with a large number of mutually dependent files, and one with very large functions. . We conclude that working with large files and making changes, fixing bugs and understanding behavior in either of these applications is difficult and time consuming. It took us three weeks just to collect all the parts of Mozilla 1.4.1 and learn how to build it. Several student researchers - not the authors of the code - spent a lot of time with the imaging research <sup>[2]</sup> code and still do not completely understand how it functions.

## **1.2 Goals**

Our goal in the current research is to develop means to make semantic preserving transformations on source code that improve its structure, and to provide a framework – a software library - for this and future research on code restructuring. In this research, we perform syntactic analysis of existing source code and create a prototype implementation that performs semi-automated software restructuring of source code to create smaller, more manageable components from existing source code, without changing its behavior. This makes it easier for software developers and testers to look closely at the functionality of source code. It improves the ability to test functionality, document sections of source code precisely, and perform maintenance effectively.

Semi-automated software restructuring means a user guided restructuring of software source code in the form of extraction of functions and methods. In this Thesis, we use the term ‘function’ for a global function, and the term ‘method’ for a member function of a user-defined type, such as a Class or a Structure. Our implementation provides a framework for extracting smaller functions from large functions, and smaller methods from large methods, from an existing source code, by an automatic process that may be guided by user input. This restructuring process requires analysis of existing source code, determining feasible regions where a new function or a method can be extracted, and finally making suitable changes to the original source that preserves its original behavior. The changes include creating a new function or a method from the extracted region, which was determined to be a feasible region, and replacing the removed code with a function or method call.

We plan future work, based in part on the framework provided here, that extracts objects from procedural code and reduces file-to-file dependencies by repackaging code (essentially bringing dependencies into one file where that is practical), eliminating global data references, and isolating dependent file sets for later redesign. Our contribution in this thesis provides a framework for source code analysis and restructuring, which can be used in this future work to study large, complex software systems.

### ***1.3 Research Statement***

Our research goal is to attack one aspect of software structure, the size of its functions, in an attempt to improve it by extracting reasonably small functions, say a few scores of lines, from much larger functions, like those of the imaging research<sup>[2]</sup> code. We will create a prototype for semi-automated restructuring of software code that makes semantic preserving transformations on code to reduce the size of selected functions by these extractions. Thus,

by restructuring, we mean breaking down existing source code into smaller entities, without changing its external behavior. This process involves scanning source code functions, identifying the viable regions where functions can be extracted and finally extracting functions. Once functions are extracted, suitable changes have to be made to the source code so that its behavior is maintained. In this work, our targets are standard C and C++ language source code. We are making this process semi-automatic, which means the processing of source code is done automatically, however the user has some control over the processing.

Restructuring source code involves scanning and parsing the source and performing a syntactic analysis of the code. Syntactic analysis identifies program structure - sections of source code that can be changed in order to improve its structure.

Software restructuring starts by identifying feasible regions where functions can be extracted. These feasible regions are determined by control scopes within functions, in the source code. Also, feasibility may be affected by the extent of local data declarations. Identified feasible regions are further examined to determine the number of parameters that need to be passed if a function is extracted from the feasible region. It is desirable to pass only a very small number of parameters, in order to keep the extracted function easy to understand and test.

The last step in the Code Restructuring process is to extract functions from the source code and make necessary changes in the original source so that the original functionality of the code remains unchanged. Once the analysis is complete, a new file is written in a separate location with code extracted from original locations. This involves making calls into newly created functions from the original location where the function was extracted, and passing variables that are needed to preserve original semantics passed to the new

functions. The variables are passed by reference so that any changes to those variables inside the new functions will be reflected in the calling function. This maintains the same external behavior of the code as it was before code restructuring.

Thus, the restructured code has extracted feasible regions, based on the code restructuring criteria of number of lines from the feasible region and number of parameters to be passed. If the number of parameters to be passed exceeds a defined limit, and the function being decomposed is a member of a class or structure, the parameters are redefined as fields of the class or structure. This eliminates the complexity of passing a large number of parameters. In this approach to simplify factoring of legacy and large software code, we provide a semi-automated source code parser to identify sections of source code which can be restructured by extracting functions, then making suitable modifications to the original source to extract them without making semantic changes to the code.

Another interesting area where software code restructuring can be extended by future work is the use of semantic cues to more intelligently select lines of code from feasible regions. These topics are discussed in chapter 5 as a future enhancement.

Another future goal of our research is to provide the user with a lot of information about the consequences of extraction and enable human decision making to select from a continuum of feasible extractions, e.g., inserting human decision making into a highly automated process by providing sophisticated visualization processes.

Code refactoring is widely discussed in the literature and several code development environments support it. We address differences between our restructuring and code refactoring in the next section.

### 1.3.1 Code Restructuring versus Refactoring

Both code restructuring and refactoring are concerned with improving code structure in some sense, and both focus, largely, on logical structure. Both intend to modify code without changing its semantics. That is, by modifying syntax with semantic preserving transformations.

The restructuring process developed in this research, programmatically examines code within a large method, determines all of the regions which are feasible<sup>1</sup> for method extraction, and then attempts to minimize the number of variables that must be passed into the new method. Both of these activities require a considerable amount of machinery, which is the topic of this and subsequent chapters in this Thesis. Traditional refactoring requires manual selection of code to extract, while restructuring automates that entirely. So a difference between our restructuring and refactoring is in the level of automation applied to the task. Refactoring addresses other factors as well, so its scope of interest is broader than ours here [2]:

1. Changing names of interfaces, classes, functions, fields, variables, and moving classes and packages.
2. Turning anonymous classes into nested classes, moving nested classes to top-level classes, creating interfaces for concrete classes, moving methods and fields between sub and super classes.

---

<sup>1</sup> Feasible regions are determined by the syntax of scoping. Control constructs, for example, each have a scope of definition, and the way these scopes are distributed across a function's code determines where extraction is feasible. In large, poorly structured code there may be scores of control scopes with both sequential and nested placement in a large function, and it can be difficult to determine appropriate regions for extraction without a lot of automated support.

3. Changing the code within a global function or class: turning local variables into class fields, turning selected code in a method into a separate method, and turning fields into properties.

There are many platform specific tools, like Visual Studio and Eclipse that support software refactoring. Refactoring, supported by these tools, is based on user selection of code to be refactored, rather than automated analysis, as reported here.

In both the literature and in widely used code development environments, like Visual Studio<sup>[12]</sup> and Eclipse<sup>[13]</sup>, extraction of methods by refactoring requires users to decide which parts to extract, with no help from the environment, to select that part in a text editor, and then turn the selection over to the refactoring process to automate packaging of the extracted code into a method and call the method in place of the extracted code.

Restructuring, as defined in this research, has much narrower scope, but is much more ambitious. Our focus is exclusively on factoring out parts of large class methods and global functions into a composition of smaller functions. The difference between our goals and those of refactoring for this activity is in the level of automation applied to the transformation.

## ***1.4 Related Work***

There are many areas in which similar concepts and ideas are used to perform analysis and transformation activities to effectively use resources, and improve performance. In this section, we detail the areas in which ideas similar to the ones in this research are used.



## **1.4.2 Code Rearrangements during Compiler Optimization**

There are many areas where code rearrangements are performed, based on the source code compiler optimizations. Below is a discussion of various tools and packages such as compiler driven products and emulator software that perform software code restructuring.

### **1.4.2.1 Compiler-Directed Code Restructuring for Improved Performance**

Code Restructuring is performed extensively in compilers and hardware platforms for optimization and parallelism. One such implementation is Compiler-Directed Code Restructuring for Improved Performance of MPSoCs<sup>[4]</sup>. This study deals with code optimization for Multi-Processor-System-on-a-Chip (MPSoC) architectures in order to minimize the number of off-chip memory accesses. It deals with a strategy that reduces the number of off-chip references due to shared data. It achieves this goal by restructuring parallelized application code in such a fashion that a given data block is accessed by parallel processors within the same time frame, so that its reuse is maximized while it is in the on-chip memory space.

### **1.4.2.2 Compiler-directed code restructuring for reducing TLB energy**

Another application area where Code Restructuring is used in Compiler technology is Compiler-directed code restructuring for reducing TLB energy<sup>[5]</sup>. This is a software-based technique for data TLBs, has considered the possibility of storing frequently used virtual-to-physical address translations in a set of translation registers (TRs), and using them when necessary instead of going to the data TLB. The idea is to restructure the application code in such a fashion that once a TR is loaded, its contents are reused as much as possible.

### **1.4.2.3 Compiler-Directed Code Restructuring for Improving I/O**

#### **Performance**

A similar approach to Code Restructuring is used for Improving I/O Performance of applications through Compiler-Directed Code Restructuring<sup>[6]</sup>. It is a restructuring scheme for improving the I/O performance of data-intensive scientific applications. This implementation improves I/O performance by reducing the number of disk accesses through a new concept called disk reuse maximization. In this context, disk reuse refers to reusing the data in a given set of disks as much as possible before moving to other disks.

### **1.4.2.4 FORTRAN Vasto90 Tool**

There are many language specific tools available with some well known programming languages that perform software code restructuring. Among the prominent ones is the FORTRAN tool Vasto90<sup>[7]</sup> which is FORTRAN language tool to transform and simplify “spaghetti” code to Fortran 77 style code. Some tools are now becoming available to further transform Fortran 66 and 77 to take advantage of the new Fortran 90 syntax. There are various versions and optimizations such as Vast77to90.

### **1.4.2.5 Code Restructuring in Emulator Software**

Software code restructuring is also used in emulator software used in system analysis. Software emulating hardware for analyzing memory references of a computer program<sup>[11]</sup> is a method for analyzing a computer program stored in a memory, the computer program including a plurality of computer executable instructions for execution on a target hardware platform.

### **1.4.3 Code Restructuring in Hardware Platforms**

Still other Code Restructuring is performed on many hardware platforms to improve efficiency of software code for parallel execution.

#### **1.4.3.1 Improving Efficiency in Embedded DSPs**

Code Restructuring for improving execution efficiency, code size and power consumption for embedded DSPs<sup>[8]</sup> is a study to improve the performance of embedded DSP processors. In this approach, a reduction in generated code size and improved parallelism is achieved by exploiting the parallelism present in Instruction Set Architecture of DSP processors.

Restructured code runs in parallel thereby improving the performance, power consumption and increased throughput for devices for DSP Processors like Personal Digital Assistants, Cellular phones and pagers.

#### **1.4.3.2 Code Restructuring in Virtual Hardware Systems**

Code Restructuring is used in virtual hardware systems in the study Computer Storage exception handling apparatus and methods for virtual hardware systems<sup>[9]</sup>. In a design system using virtual hardware models, a filtering manager for filtering execution results and determining which software instructions are candidates for restructuring. This approach is targeted to improve device exceptions and reduce overheads or crashes. In some examples, illegal address range instructions are identified based on exception records and restructured software instructions may redirect memory access to an appropriate memory location thereby enabling the use of hardware device drivers in conjunction with hardware emulations, simulations or virtual models without requiring driver source code modifications.

### **1.4.3.3 Code Restructuring by Source code Transformation**

Code Restructuring is also used in Generating Hardware Designs by Source code transformation<sup>[10]</sup>. This study is targeted towards field-programmable gate arrays. The implementation is a user-customizable transformation system for a high level hardware description language.

### **1.4.4 Table-driven Lexical Analysis**

Lexical Analysis is our first step in software code restructuring. There are many approaches to do this, depending on the platform and language used. Among the prominent ones are Lex, Yacc, Flex, Bison, and Antlr. In the following sections, we discuss these alternate technologies and cite reasons why we chose to pursue our work using an “ad-hoc” grammar analyzer. We start with an introduction to each of the alternate technologies:

#### **1.4.4.1 Lex and Yacc**

Lex is a part of BSD Unix, used to break up an input file stream or user entered characters into meaningful elements that are recognized by a language. Lex is widely used in compiler design to identify specific characters that make up constructs of a language. This process of breaking up streams into identifiable characters is termed as lexical analysis and Lex is widely used in doing so. Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine. Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the

corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. At present, the only supported host language is C.

Yacc is also a part of BSD Unix and is used to analyze the structure of an input stream. It stands for "Yet another Compiler Compiler". This analysis of input structure is carried out for syntactical correctness of the input stream. Yacc works on words and not on individual characters. This process of parsing is carried out in compilers to check for correctness. In C Compilers, Yacc is to check for declarations, definitions of variables, to match closing braces and for checking the terminating statements. Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level

input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

#### **1.4.4.2 Flex and Bison**

Flex and Bison are variants of Lex and Yacc, available in GNU. Flex and Bison are more flexible and are available for other platforms like BSD Unix and Windows. They are used in lexical analysis and parsing of input streams and have advanced features like user defined type checks and optimization algorithms. Flex and bison are code-generating tools designed to aid in compiler development. In particular, Flex will take a sequence of characters and break them apart into tokens (words). It could be used to write code breaking this article into a sequence of words and punctuation. Bison will group words into sentences, the sentences into paragraphs and the paragraphs into sections.

#### **1.4.4.3 Antlr**

Antlr stands for “Another Tool for Language Recognition” and is a parser generator that is used to create language recognition tools like compiler front ends. In addition to native language recognition, there can be user defined rules for maintaining the syntax of a language. Antlr is mostly used in creating pattern and language support tools and currently it supports code generation for C, C++, Java, C# and Python. ANTLR, provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR automates the construction of

language recognizers. From a formal grammar, ANTLR generates a program that determines whether sentences conform to that language. There can be user-defined grammar checks and language recognition.

In the above section, we have introduced alternate technologies that we could have used to pursue our restructuring. The technologies discussed above have some limitations with respect to platform and compilers. Our framework would be platform dependent had we used these technologies. Furthermore, the grammar we need is a very small subset of the C and C++ languages, and the initial setup of these alternate tools isn't warranted for our simpler tasks.

Finally, the lexical analysis process that we have discussed in the above section is only a part of our restructuring process. Lexical Analysis is followed by a determination of the feasible regions where functions can be extracted, followed by extraction. By considering all these issues, we decided that using the alternate tools would not be effective for our work.

We have seen several areas where code restructuring can be used in order to enhance the performance of systems or make transformations in source code to use resources effectively. When dealing with large software systems, experience shows that a lot of time is spent on understanding the current functionality, determining changes needed in source code and testing for errors. As pointed out earlier, there is also a risk of introducing incorrect code. Looking from this point of view, one way to improve software production, quality, and development time would be to improve the understanding of software source code and decrease the time spent to reach an understanding.

## Chapter 2 - Restructuring Techniques

Code structure entails both logical and physical disposition of statements of the programming language used for implementation. Logical structure is determined by the way code is divided into global functions, classes, class hierarchies, relationships between classes, and the way data is factored into collections. Physical structure is determined by how the logical parts: functions, classes, and data structures, are placed into physical files and the way those files are compiled, e.g., into libraries or execution images.

In this chapter, we discuss the elements of source code such as control scopes and methods of identifying data and its usage. We present methods of identifying feasible regions, from which new functions can be extracted, and various constraints associated with determining feasible regions. We also present examples of extracting functions and methods, in different scenarios where parameters may be required for newly extracted function or methods. We conclude this chapter, by discussing two modules<sup>2</sup> - Tokenizer and Semi-Expression, which are used for extracting the contents of source code and performing lexical analysis. We also discuss construction of a simplified language grammar<sup>3</sup>, discussed in detail in Chapter 3.

---

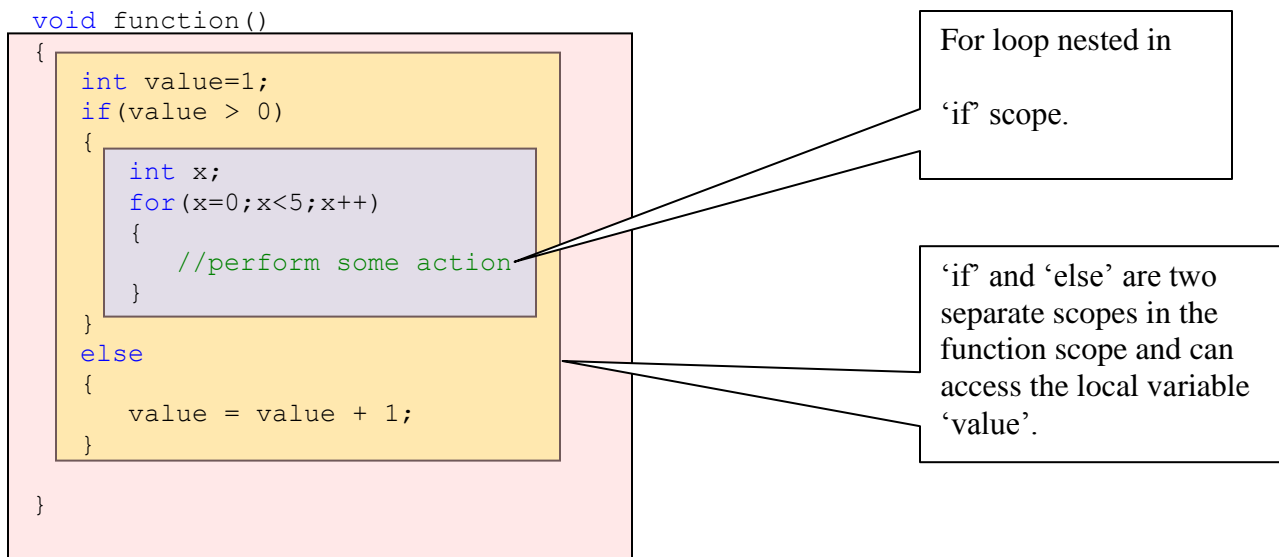
<sup>2</sup> A module is a logical separation in a software program, which may contain a header (.h) and a definition file (.cpp)

<sup>3</sup> Grammar is identifiable program elements like basic and derived types, declaration and definition of variables and data.



## 2.1 Syntax of Scopes and Data Declarations

A scope is a section of source code where the existence and usage of a variable or object declared in that particular scope is valid, and is invalid outside of it. Because of this, scopes are usually associated with ‘lifetime’ and ‘accessibility’ of variables or objects. Programming logic can also be included in scopes, such as conditional statements and repetitive source code steps to perform an activity. Control Scopes are defined by a region enclosed by the characters ‘{’ and ‘}’. Scopes can exist at the function level, within functions, or nested within other scopes. Scopes in the source code can be sequential, or can be nested within other scopes. Below is a figure depicting sequential and nested scopes.



**Figure 2.1 – Syntax of Scopes**

From the above figure, we can state the difference in accessibility of data variables that exist in scopes. Referring to figure 2.1, the integer variable ‘value’ can be accessed anywhere in the function. The scopes ‘if’ and ‘else’ are two sequential scopes that exist in the function, and ‘for’ loop is a nested scope that is contained in ‘if’. The variable ‘x’ can be accessed only in ‘if’ scope. Every variable in the source code belongs to a control span which

may be at global, function or control construct<sup>4</sup> level. Also, we notice that control spans don't overlap, each have their own region. For restructuring, source code is analyzed based on the control spans and variables in a given span. Information about control types, the variables declared in them, and their usage is stored in data structures that allow us to store and retrieve information about feasible regions easily. This turns out to be a parse tree, in a unique format. More information about our data structures is given in Chapter 3.

### **2.1.1 Feasible Regions based on Control Scopes**

Every region enclosed in a control span is identified by the parsing process, and defines a candidate region, which, if it matches user supplied criteria for selecting feasible regions, number of lines of code subtended and number of variables that must be passed if extracted as a function, are determined to be a feasible region. Every feasible region must be contained or be contained by local control scopes, which may belong to a function or other control types. Feasible regions cannot be identified across scopes. Below is an example showing a region that may be, if it matches the user criteria, is a feasible region.

---

<sup>4</sup> By control constructs we mean loops, branches, exception handling clauses, and switch statements.

```

void funTest(int arg1, int arg2)
{
    std::string inFile = "";
    try
    {
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}

```

Source code region can be analyzed to determine whether or not it's a feasible region

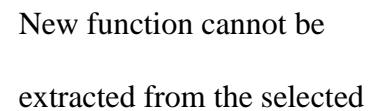
From the code fragment above, we notice that the region for analysis belongs to a ‘try’ block exclusively and is entirely contained within one scope. All data defined in a scope can be accessed in that particular scope itself. For enclosing scopes, variables declared in the outer scopes can be accessed in inner scopes. Each control statement in source code, like loops or conditional statements, define a scope and have a purpose of treating all code within the scope as a single block of processing. New functions cannot be extracted across two or more scopes as it breaks the code structure and the code can’t be recompiled after such code restructuring.

## 2.1.2 Inclusion of Local Data Constraints

To maintain the same external behavior of restructured code as that of the source code, the data values that the source code operates on, should be the same for restructured and original code. When deciding on feasible regions, an important criterion is data accessibility. Code restructuring works by extracting new functions, from existing functions. If a local data declaration is included in extracted function, the function must include the entire data span. Thus, if an extracted function contains a data declaration which is used subsequently in the calling function, the code won't compile. Below is an example to illustrate this:

```
void funTest(int arg1, int arg2)
{
    std::string inFile = "";
    try
    {
        std::string str="";
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
        str.append("string value");

    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}
```

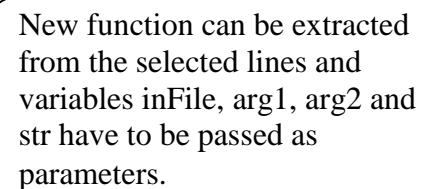


New function cannot be  
extracted from the selected

From the selected lines, a new function cannot be extracted as the string 'str' is used in the last line of 'try' scope. By extracting new function, the declaration and usage will be in different functions, and the source code won't compile.

Likewise, if a new function is extracted following the declaration of data variable, the variable has to be passed as a parameter to the newly extracted function. Considering the same source code shown above, if a new function is extracted from line 7, the variables 'inFile', 'arg1', 'arg2' and 'str' have to be passed to the new function, as it would be used in the new function.

```
void funTest(int arg1, int arg2)
{
    std::string inFile = "";
    try
    {
        std::string str="";
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
        str.append("string value");
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}
```



New function can be extracted from the selected lines and variables inFile, arg1, arg2 and str have to be passed as parameters.

Also, the newly extracted function might have data that is being changed in the body of the new function. The same data might have been accessed in the parent function, after the end of feasible region, which was extracted as a new function. The point where data is accessed after returning from the new function should have the latest value set by the newly extracted function. For example, if the value of a variable is changed in the newly extracted function, it should be reflected in the parent function when control returns from the new function.

Considering this, all parameters that are passed to the new function should be passed by reference, so that any changes in the extracted function would reflect a corresponding change in the calling function.

## 2.2 Extraction of Functions

Public functions are accessible from all other functions and classes, provided they have seen the function definition. If it is defined in a different file, it has to be included into the file from which a call has to be made. By restructuring functions, the newly extracted functions should also be accessible in a similar fashion. Maintaining this, the newly extracted functions will also have the same inclusion and accessibility properties of their parent functions, from which they are extracted.

Considering a global function, let's see some results of how restructuring is performed and what has to be considered during restructuring. First, we present the cases where, no arguments are passed to the newly extracted function and following that, arguments are passed to the new function.

### 2.2.1 Extracting functions with no parameters

Below is the source code of a global function – 'FunTest'

```
void FunTest(int arg1, int arg2)
{
    try
    {
        std::string inFile = "";
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception ex)
    {
        exit(1);
    }
}
```

Performing code restructuring on the above function, we extract a new function, FunTest\_1(). Below is the source code of the extracted function and changes that appear in the parent function, FunTest.

```
void FunTest_1()
{
    std::string inFile = "";
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
}

void FunTest(int arg1, int arg2)
{
    try
    {
        FunTest_1();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception ex)
    {
        exit(1);
    }
}
```

From the above source code, we see that a new function 'FunTest\_1' is extracted from the parent function 'FunTest'. In place where the body of extracted function was present in 'FunTest', before restructuring, a function call has been made to the new function. We also notice that no arguments need to be passed to the new function.

## 2.2.2 Extracting functions with parameter passing

When new functions are extracted, data defined before the extraction and used within the extracted code, determine whether or not parameters from the calling function have to be passed to the new function. Recalling that our goal of restructuring is that the restructured code should be able to build and behave, as before restructuring. In order to maintain that

behavior, it's important to consider the scenarios which we'll discuss below while passing arguments to new functions.

### 2.2.2.1 Scenario 1 – Argument declared before Function Call

In this section, we present an example where the extracted function requires one parameter to be passed as an argument. Considering the source code in the section 2.2.1, with a global function – FunTest, and performing restructuring, with the selected feasible region set to span 6 lines, we notice that the argument 'arg1' has to be passed to the newly extracted function.

```
void FunTest_1(int& arg1)
{
    std::string inFile = "";
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    arg1++;
}

void FunTest(int arg1, int arg2)
{
    try
    {
        FunTest_1(arg1);
        arg2++;
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception ex)
    {
        exit(1);
    }
}
```

In this scenario, we notice that an argument 'arg1' is passed to the newly extracted function. This variable is passed to the parent function 'FunTest' as an argument. We also notice that the variable is passed by reference, so that the changes made in the new function would reflect in the calling function.



### 2.2.2.2 Scenario 2 – Determining the number of arguments

We have seen in the previous section that a single argument is passed to the newly extracted function. Considering the same source code in Section 2.2.1, if we increase the number of lines in a feasible region to 7, we notice that two arguments have to be passed to the newly extracted function. Below is the restructured code:

```
void FunTest_1(int& arg1, int& arg2)
{
    std::string inFile = "";
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    arg1++;
    arg2++;
}

void FunTest(int arg1, int arg2)
{
    try
    {
        FunTest_1(arg1, arg2);
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception ex)
    {
        exit(1);
    }
}
```

We notice that as we increase the number of lines to be considered while analyzing a feasible region, the number of arguments also are affected. By this, we conclude that the number of lines in a feasible region and number of parameters to be passed are mutually dependent.

## 2.3 Extraction of Methods

We have seen the process of restructuring functions in Section 2.2.2. In this section, we discuss restructuring methods. As mentioned earlier, we use the convention of calling a member function of a user-defined type such as Class, Structure, Union or Enumeration, a

method. Parsing methods and identifying feasible regions are essentially the same as that of a function. However, restructuring of methods require changes in the method declared and class declarations. Methods may or may not be declared and defined in the same file. Hence, restructuring methods require two sets of changes – changes to the method declaring file and changes to the method defining file. The newly extracted function is defined in the same file as that of the method.

Below is the source code of a class definition, which includes two data members and one method declaration.

```
class test
{
    private:
        int x;
        int y;
    public:
        void funTest(int arg1, int arg2);
};
```

Below is the source code of method defined file –

```
void test::funTest(int arg1, int arg2)
{
    try
    {
        std::string inFile = "";
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        x = 300;
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}
```

Performing Restructuring on the above source code, with number of feasible region lines set to be 8, requires changes to the class definition and method definition.

Changes made to the restructured class definition:

```
class test
{
    private:
        int x;
        int y;
    public:
        void funTest(int arg1, int arg2);
        void funTest_1(int& arg1, int& arg2);
};
```

Changes made to the method defined file:

```
void test::funTest_1(int& arg1, int& arg2)
{
    std::string inFile = "";
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    arg1++;
    arg2++;
    x = 300;
}

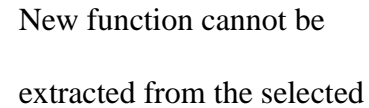
void test::funTest(int arg1, int arg2)
{
    try
    {
        funTest_1(arg1, arg2);
        int value = 1;
        int value2 = 2;
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}
```

We notice that the parameters ‘arg1’ and ‘arg2’ are passed to the newly extracted function by reference. We also notice the function prototype of the new function is added to the class definition. The other interesting result is that the variable ‘x’ that is used in the new function is not passed. This is because, both the functions belong to the same class and have access to it’s member data, to which x belongs.

### 2.3.1 Transforming local data into member data

Let's consider the source code below:

```
void test::funTest(int arg1, int arg2)
{
    try
    {
        std::string inFile = "";
        std::string str="";
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        arg1++;
        arg2++;
        int value = 1;
        int value2 = 2;
        str.append("string value");
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}
```



New function cannot be  
extracted from the selected

We have spoken about this scenario in Section 2.1.2, where a new function cannot be extracted from the considered feasible region declaring a local data variable, as the data variable is used following the feasible region. This will leave the declaration and definition of data in separate functions and the source code will not be compile. In such a case, the variable can be added as a data member of the class. The variable to consider for the example shown above is 'str'. The class definition for the above example is presented in Section 2.3. By executing the idea of transforming local variable into a data member of the class, the class definition will include a declaration of the considered variable in its class definition.

Below is the change that would occur in the class definition.

```

class test
{
    private:
        int x;
        int y;
        std::string str;
    public:
        void funTest(int arg1, int arg2);
        void funTest_1(int& arg1, int& arg2);
};

```

The changes to the method defined files are:

```

void test::funTest_1(int& arg1, int& arg2)
{
    std::string inFile = "";
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    arg1++;
    arg2++;
}

void test::funTest(int arg1, int arg2)
{
    try
    {
        funTest_1(arg1, arg2);
        int value = 1;
        int value2 = 2;
        str.append("string value");
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() << std::endl;
    }
}

```

We notice that the variable ‘str’, of type string is added as a data member of class ‘test’ and the declaration that was there in the extracted function is removed. We also notice that the function ‘funTest’ can access the variable, as the variable and the method belong to the same class. This idea is presented in Chapter 5 for future work.

## 2.4 Capturing Feasible Regions

The first stage in Code Restructuring is defining a language grammar and identifying source code elements that comply with the defined grammar. By grammar, we mean identifiable program elements like basic and derived types, declaration and definition of variables and data. For extracting code fragments from functions, we need only a quite simple grammar – much less than that required to define the entire language<sup>5</sup>.

For our implementation of Code Restructuring, we are only concerned with identifying the scope of control constructs, as they cannot be split across function boundaries, and with local data definitions and parameters passed into the parent function. Every identified data type is associated with a control scope – the region in the source code where a variable is defined and can be accessed. Every variable in the source code belongs to a control span which may be at global, function or control construct<sup>6</sup> level. A control span is defined by curly braces ‘{’ and ‘}’. All variables declared within two enclosed curly braces, belong to that span. Source code is analyzed based on the control spans and variables in a given span. Information about control types, the variables declared in them, and their usage is stored in data structures that allow us to retrieve information about feasible regions easily. This turns out to be a parse tree, in a unique format. More information about our data structures is given in Chapter 3.

Each control span that is identified in the parsing process defines a candidate feasible region for extracting functions. Our criteria for selecting one of these regions for extraction include number of lines subsumed by the region and the data that needs to be passed if a

---

<sup>5</sup> We focus on the C and C++ programming languages for this research.

<sup>6</sup> By control constructs we mean loops, branches, exception handling clauses, and switch statements.

function is extracted. The number of lines a control structure spans represents the length of the control span. That data is used to identify candidate feasible regions. Other optimization techniques are included to identify regions that most qualify for extraction, based on the number of parameters that must be passed, discussed below.

## ***2.5 Passing Parameters***

The determination of passing parameters to the newly extracted function depends entirely on whether or not a variable declared in the analyzing function would be used in the new function. This is affected by the number of lines that exist in a feasible region, and eventually a new function. Constraints on the number of lines extracted and number of allowed parameters are set by the user. If no candidate region satisfies the constraints, extraction fails.

If parameters are to be passed to a new function, all parameters are passed by reference, so that any changes made to the variable in the new function, would be reflected in the calling function. This is done to maintain the behavior of source code, as the variable that was changed in the new function, might be used in the calling function after its use in the new function. In such a case, the variable value should be the latest that was changed in the calling function. Passing variables by reference, also avoids creation of new copies of the variables, which is a property of variables being passed by value. This avoids the overhead of copying large data values like large objects and structures.

## ***2.6 Deciding which code to Extract***

For deciding what code to extract, we choose from candidates that satisfy user entered constraints. These are values entered by the user, before the restructuring process.

User entered constraints include the file to be restructured, the maximum number of parameters to be passed to an extracted function and maximum number of lines in an identified feasible region. These are entered by the user as command line arguments. Code Restructuring is carried out subject to these by an optimization process that attempts to minimize the number of passed parameters by adjusting the lines extracted within a feasible region. The process of filtering out feasible regions meeting these constraints is carried out by analyzing the built parse tree for feasible regions. All regions that don't meet these constraints are not identified as candidates for extracting functions.

Below is a screen shot of these user entered constraints –

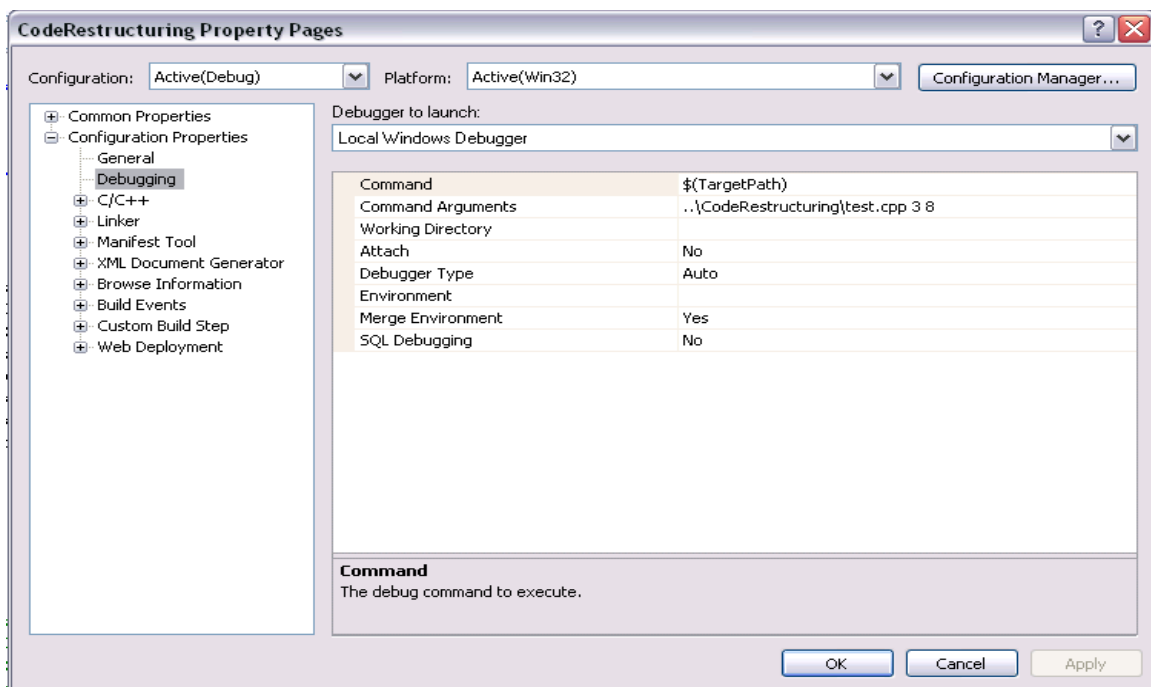


Figure 2.2 – User Entered Constraints

## 2.7 Our Lexical Analysis Tools

Our approach is built around a light weight parsing mechanism, in which we need only limited system resources. We make use of a memory resident parse tree instead of



building a dictionary of program elements or Program Execution Graphs (PEGs). In Chapter 3, we present a detailed discussion of the complete parsing mechanism, our parse tree data structure, building and using the parse tree and identifying feasible regions - regions in the source code that form candidates for extracting new functions.

Tokenizing is a process of getting identifiable character sequences from an input stream. The identifiable characters are the characters defined in the host language's character set. This is carried out by a module called the Tokenizer. By module, we mean a cohesive set of functionality, packaged as a unit and containing self-describing comments. Tokenizing is similar to lexical analysis that is discussed in the previous section. The Tokenizer extracts one word<sup>7</sup> at a time from an input stream which may be a file, user entered values or a string of characters. For our work we use a text stream for analysis. The Tokenizer is fundamental to the program and is our starting point in performing analysis on the source file or stream. The characters that are extracted from Tokenizer are called 'Tokens' and are passed to the next module that is Semi-Expressions. Below is a sample output from Tokenizer extracting 'tokens' from a source file:

---

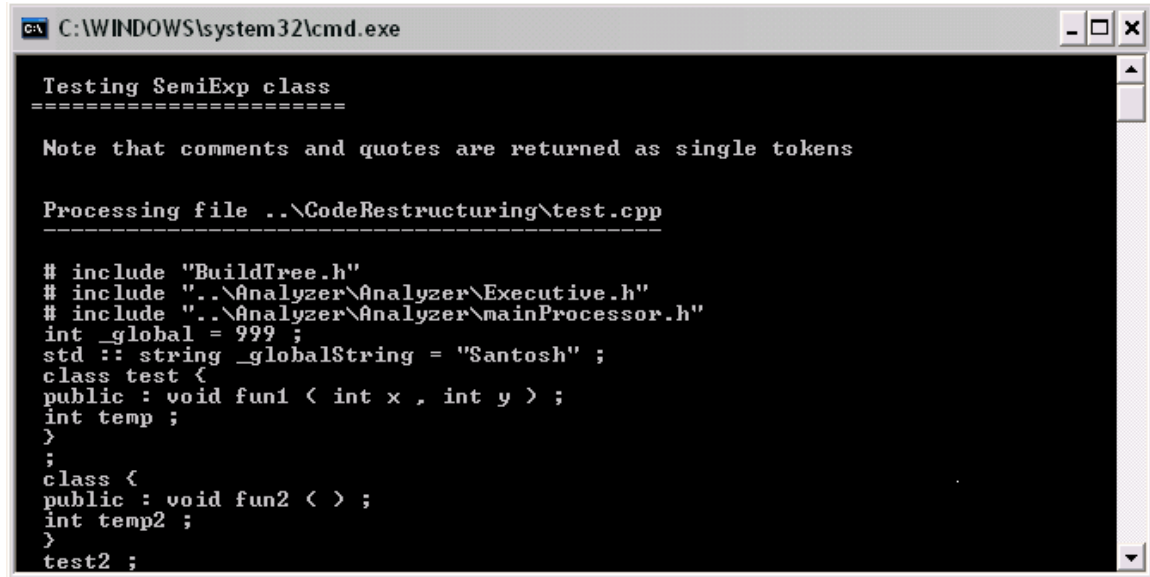
<sup>7</sup> A word is a contiguous sequence of alphanumeric characters or contiguous punctuator characters. Certain punctuators are identified as single character tokens, like '{', '}', and ';'.

```
C:\WINDOWS\system32\cmd.exe
ln: 71, br lev: 4, tok size: 1 -- <
ln: 71, br lev: 4, tok size: 1 -- &
ln: 71, br lev: 4, tok size: 8 -- newToken
ln: 71, br lev: 4, tok size: 1 -- >
ln: 71, br lev: 4, tok size: 1 -- ;
ln: 72, br lev: 4, tok size: 1 -- newline
ln: 72, br lev: 4, tok size: 41 -- //_typeParser->getUserDefinedTypes(root);
ln: 73, br lev: 4, tok size: 1 -- newline
ln: 73, br lev: 4, tok size: 11 -- _typeParser
ln: 73, br lev: 4, tok size: 2 -- ->
ln: 73, br lev: 4, tok size: 12 -- getTypeNames
ln: 73, br lev: 4, tok size: 1 -- <
ln: 73, br lev: 4, tok size: 1 -- >
ln: 73, br lev: 4, tok size: 1 -- ;
ln: 74, br lev: 4, tok size: 1 -- newline
ln: 74, br lev: 3, tok size: 1 -- }
ln: 75, br lev: 3, tok size: 1 -- newline
ln: 76, br lev: 3, tok size: 1 -- newline
ln: 76, br lev: 2, tok size: 1 -- }
ln: 77, br lev: 2, tok size: 1 -- newline
ln: 77, br lev: 2, tok size: 20 -- //_Building Objects..
ln: 78, br lev: 2, tok size: 1 -- newline
ln: 79, br lev: 2, tok size: 1 -- newline
ln: 79, br lev: 2, tok size: 3 -- for
ln: 79, br lev: 2, tok size: 1 -- <
```

Figure 2.3 – Sample output from Tokenizer Module

Comments and quoted strings are each returned as single tokens, or in the case of comments, optionally discarded.

Semi-Expression is a module that gathers extracted tokens from a stream and makes meaningful expressions required for analysis. There are single character tokens that determine the end of a syntactically meaningful set of tokens, which are called ‘terminators’, e.g., ‘;’, ‘{’, ‘}’, and ‘\n’ if its line starts with ‘#’. Each set of characters up to and including a ‘terminator’ is called a Semi-Expression. Whenever a terminator is encountered, all the tokens gathered till then becomes a Semi-Expression. All tokens that are extracted following this would become the next Semi-Expression. Each of the Semi-Expressions, are passed to the next module to perform grammar checking and analysis. The next set of tokens is not extracted till the parsing of a current Semi-Expression is complete. Below is a sample output from Semi Expressions that is generated from tokens extracted from a source file –



```
C:\WINDOWS\system32\cmd.exe

Testing SemiExp class
=====

Note that comments and quotes are returned as single tokens

Processing file ..\CodeRestructuring\test.cpp
-----

# include "BuildTree.h"
# include "..\Analyzer\Analyzer\Executive.h"
# include "..\Analyzer\Analyzer\mainProcessor.h"
int _global = 999 ;
std :: string _globalString = "Santosh" ;
class test {
public : void fun1 ( int x , int y ) ;
int temp ;
}
;
class {
public : void fun2 ( ) ;
int temp2 ;
}
test2 ;
```

Figure 2.4 – Sample output from Semi-Expressions Module

Tokens and Semi-Expressions can be extracted from a source file or a set of files in a directory, or in a directory tree. This gives us the flexibility of extending our Code Restructuring process across many files. We can also define different terminators for defining end of a Semi-Expression. This gives us flexibility in parsing different native languages and analysis of grammar can be performed based on the type of Semi-Expression that was extracted from the input stream. Semi-Expressions are used by the parser for building the parse tree by analyzing tokens that are contained in the Semi-Expression.

The next Step in Code Restructuring is parsing of Semi Expressions. This starts with identifying expressions defined in our grammar. These expressions include identifying classes, functions and data that are defined in the source code. Each identifiable expression becomes a node that forms a part of the parse tree. Properties associated with nodes are added, which include declared and defined file name, name of the expression, class or function name, and line number. Scope hierarchy is maintained when building the parse tree.

Functions and data which belong to a class become child nodes of that class in the parse tree. All scopes defined in functions, become child nodes of that function. The types of nodes in our parse tree are root node, class node, global function and data node, member function and data nodes, and scope nodes. All nodes are descendents of the root node.

## ***2.8 Grammar Construction and Parsing***

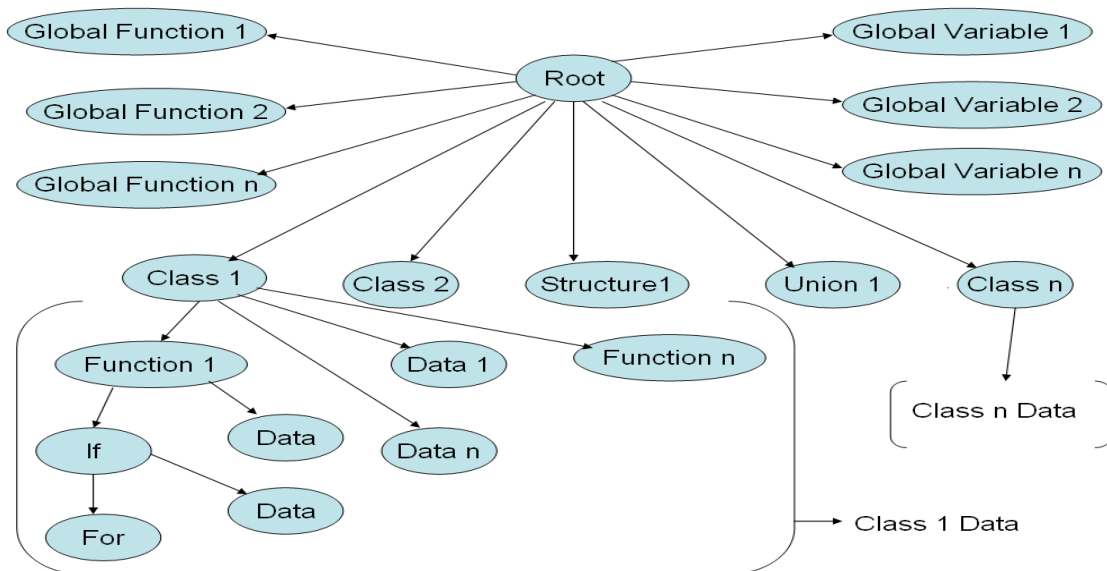
Our Framework for Software Code Restructuring is based on building an n-ary parse tree of program constructs and analyzing the parse tree. The parse tree consists of nodes, with each node representing a program element, which can be a class, global function or data, member function, private members, functions or scopes within functions. Nodes also contain information about where program element is declared and defined in a file. The declaration and definition of program elements may be in different files and different line numbers. The line numbers of respective program elements are chosen to be our point of reference during software restructuring. Nodes also contain information of where data is declared, defined and where it is used. This enables us to keep track of data that may need to be passed into an extracted function.

Software Restructuring is based on identifying feasible regions where functions can be extracted. The criteria for identifying such regions include number of lines of candidate region and the number of parameters that are to be passed, in order to preserve the semantics of the code after Code Restructuring. These feasible regions may be scopes within functions or sections of scopes, in the source code.

The last step in Code Restructuring process is to extract the feasible regions from the source code and make necessary changes in the source code so that the external functionality of the source code remains unchanged by creating a new function and calling it at the point

that its code was extracted. Once the feasible regions are identified, pointers based on the line numbers are maintained determining the span of the feasible region. Based on these pointers, a new restructured file is written in a separate location with feasible regions extracted from their original location and written separately. This involves making calls into the newly created function from the original location where the function was extracted along with passing all the required variables that are to be passed to the new function. The variables are passed by reference so that any changes to those variables will be reflected in the called function. Thus, maintaining the same external behavior of the source code as it was before code restructuring.

Below is a model of a hypothetical parse tree that is generated during the parsing of source code.



**Figure 2.5 – Top Level Structure of Parse Tree**

As we can see from the above figure, the root node is central to all nodes, and all other nodes are derived from the root node. Each of the nodes, have specific properties depending on the node type. The top level nodes are global functions, global data and User-

Defined types like classes, structures, unions or enumerations. The global function nodes have Scope nodes. The scope nodes determine the scope type and its span. Each of the Scope Nodes may have other scope nodes within them. Every Scope node is associated with a collection of data Objects, which we'll discuss in detail in the coming chapters. All properties associated with a scope, like the number of lines it spans and all the data that is declared and used, is stored in a Scope Node. The Global Data Node represents global variables in the source code and has properties that keep track of its declared and defined line numbers, file names and usage. The User-Defined type nodes are class, structure, union or enumeration nodes. These nodes have two components – Member Function Nodes and Member Data nodes. The member function nodes are Function Nodes that are members of the type and, member data nodes are data members of respective types. All properties of Function nodes apply to member function nodes and belong to a particular User-Defined type. The description of each of the node type and its significance is discussed in detail in the following chapters.

The parse tree is generated dynamically as we parse the files and information is added to nodes as the program elements are encountered. The determination of sections of files, where software code restructuring can be performed, is done during the generation of the parse tree. Once this determination is done, a new set of files is created with restructured source files. This way the source files are maintained as they are and new set of files are written to a separate location. This will also give a chance for the user to compare the two files.

## ***2.7 Summary***

In this chapter we have discussed steps in analyzing source code and determining feasible regions, and extracting functions. In the next chapter, we discuss the various techniques and implementation details needed to enable extraction of functions. Details of the data structures we use, the process of building our data structure, and its components are presented in Chapter 3.

## Chapter 3 Code Analysis

In this chapter, we detail all the steps and techniques we have used for identifying feasible regions, from which we perform Software Code Restructuring. We also detail the data structures we are using, the purpose and importance of them, and how the data structures are grouped and co-ordinate. We also present various parsing techniques and detectors we have used to validate source code elements with our grammar. The different types of data structures – persistent and temporary, which are used to hold scope and data information of source code, are detailed in this chapter.

This chapter also details each of the implementation techniques and details of components.

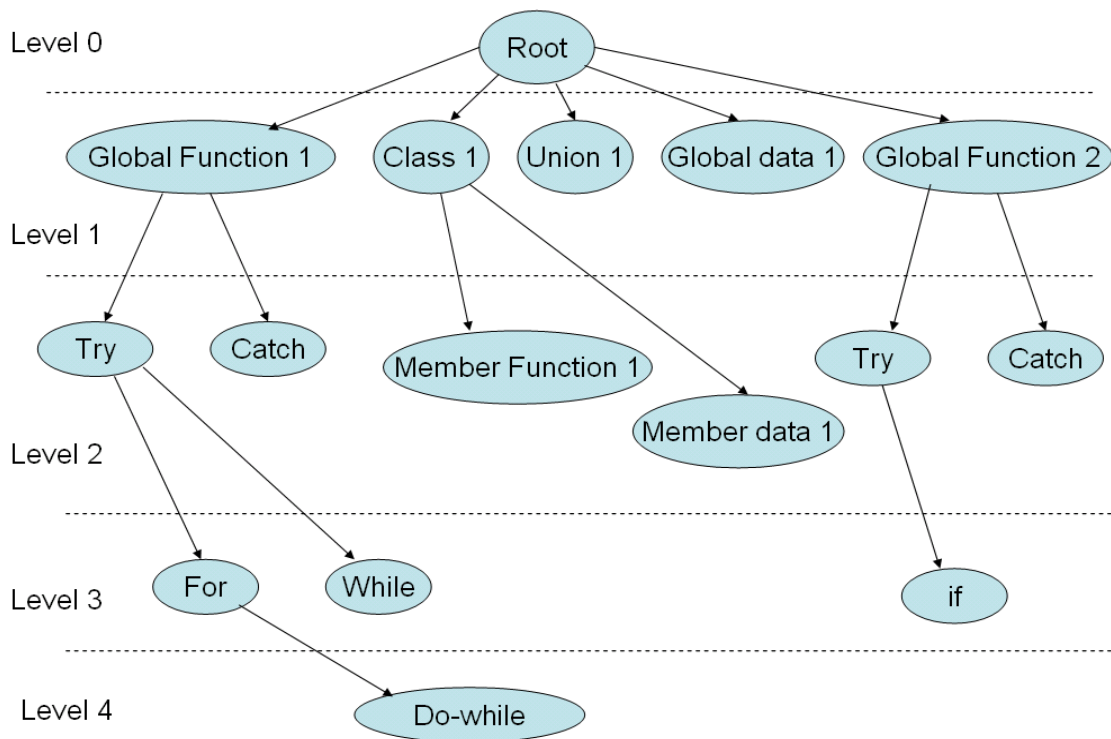
### **3.1 Analysis**

In the earlier chapters, we have discussed using tokens and Semi-Expressions, which are fundamental to parsing source code and, building our parse tree. The parse tree that is created by scanning source code is our central data structure and is an m-ary tree, which represents all the program elements of the source code. By program elements, we mean global functions and data, classes, member functions and member data, local data in functions, scopes in functions, and data in scopes. The parse tree also includes details of use of data, like the line number and location of occurrences of data. The parse tree is created dynamically by parsing through the Semi-Expressions and identifying elements that make the parse tree. The parse tree is built with connecting nodes. Each node is an identifiable component of the source code which may be a class node, function node or a scope node. Each of these nodes, have data objects, which are objects with associated properties which



exist in a particular node type. The details of data objects are described in the following sections.

The Semi-Expressions are parsed to identifiable program elements like classes, functions, scopes and data. Whenever an identifiable type is encountered, a new node of that particular type is created and added at respective position. The parse tree has a structure that is different from the order in which program elements are encountered from the source code. All nodes are derived from a 'root' node, which is at the 0<sup>th</sup> level. Level is term we use to represent the depth of the parse tree, at a depth i, where i can extend from 0 to n. All nodes are derived from the root node. Global functions, Global data, User-types like classes, structures, unions and enumerations form the 1<sup>st</sup> level nodes. Below is an example of a parse tree with different levels of representations.



**Figure 3.1 - Different Levels in Parse Tree**

Above is an example figure of parse tree, which is generated dynamically as and when identifiable program elements are encountered, at the same time maintaining the level order. Every node in the parse tree represents a User-Defined type, function or a scope. The root node is defined at the global level, which is termed as Level 0 the parse tree. Global Function and data, and User-Defined types like Classes, Unions, Structures and Enumerations are derived from the root node, and are termed as Level 1. All member functions and member data form Level 2 in the parse tree. All other scopes with in functions, like a for, while, do-while, if, else, else-if are descendents of Level 2, and start with level 3, going down the parse tree. The height of the tree is not balanced as there can different kinds of depths in scopes and functions.

The root node is always at level 0 and all other nodes are derived from the root node. All program elements are encountered dynamically, by parsing the Semi-Expressions are added at appropriate level and it's not required that the Semi-Expressions have to be fetched in the order in which the tree is built. Every time a Semi-Expression with an identifiable program element is extracted from the source code, respective node is created and added to the parse tree at appropriate level and position.

As we can see from the figure above, the root node is the parent node of all other nodes and occupies level 0. Global Functions and global data occupy level 1, together with User-Defined types like classes, structures, unions and enumerations. All member functions and member data of User-Defined types occupy level 2. All other scopes with in functions, and scopes with in scopes, are added to the parse tree, as it appears in the source code.

Every node in the parse tree has respective properties that are associated with the node type. There are properties specific to classes, functions and scopes. And since each of

the classes, functions and scopes, represent a different node type, each of the node type have properties that are associated with that particular node type. There are some properties that are common to all node types like declared line number, used line number, declared and defined files. In the following section, we will detail the information of the data structures that are used in our approach of Code Restructuring and the components of the parse tree. The method of parsing and building of parse tree, identifying feasible regions for extracting functions by parsing the parse tree are detailed in the following sections. The criteria used in identifying feasible regions, with the process of extracting functions and redefining the Restructured source code in a different file are detailed in the following sections. We begin the next section with an introduction to scopes and saving scope information.

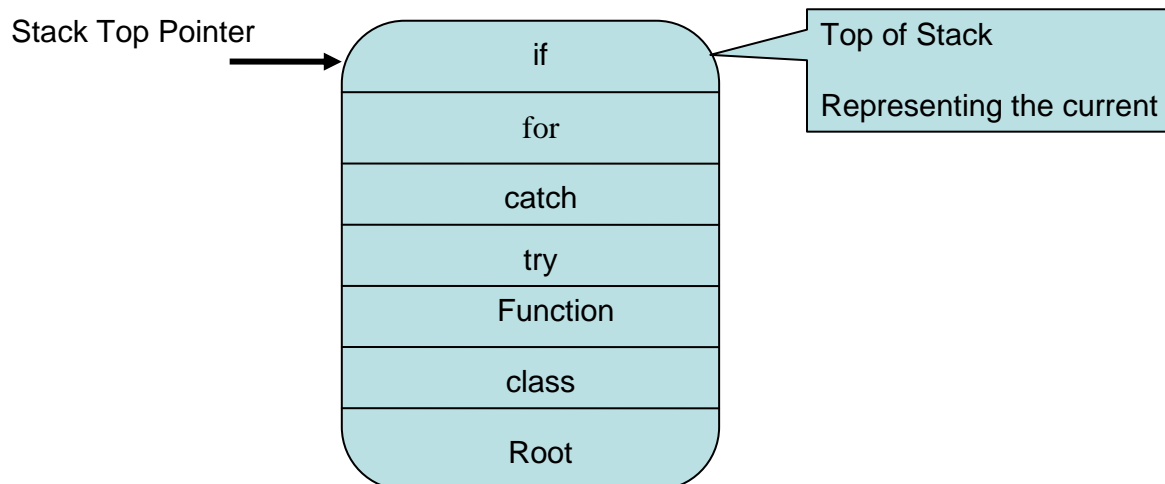
### **3.1.1 Parsing Scopes and Saving Scope information**

We have discussed scopes in Section 2.1. A Scope is defined as a region in the source code, where in data declared in that scope can be accessed and used in that particular scope and can't be accessed outside of it. Scope can be a function scope, scope by itself, or enclosed in another scope. It is represented as a region by characters between '{' and '}' in the source code. We have also mentioned in Section 2.1, that scopes belong to a control type, which represents the program logic of that section and can be data structures, functions, conditional or repetitive statements.

Parsing of scopes requires associating scope with its control type. For this, it is important to know the types of elements that form a scope. While parsing for scopes, we treat user-defined data structures as one category and functions, conditional and repetitive statements as another category. Every identifiable element in the source code is stored in our parse tree, the details of which will be presented in the later sections of this chapter. Scope

determination and identification is a two step process, which is closely connected to building of parse tree. The process of building of parse tree is detailed in Section 3.1.3. In order to determine scope, it is important to understand how the parse tree is constructed. We have introduced parse tree in the previous section and looking at figure 3.1, we have different levels in the parse tree. Parse tree is constructed partially in the first step, which are levels 0, level 1 and level 2, according to the figure 3.1 This forms the base line scopes of the source code, which includes the user-defined types, global data and global functions. User-defined types include data structures such as classes, structures, unions or enumerations, their data members and their member functions. All other scopes are derivatives of these scopes. The scanner performs type analysis to determine the elements of first three levels in the parse tree, which include type checks for user-defined types, global data – basic and derived types, and global functions. The association of data members and member functions is done during the processing of user-defined types. All other data and functions are referred as global functions and data.

Once this framework is laid out, the rest of the analysis is based on the ‘current’ scope. For this, we make use a temporary data structure called the ‘hierarchy stack’. In the process of scanning functions, our technique maintains a hierarchy stack for keep track of scopes and the current location in the source code. The hierarchy stack maintains the information that’s represented in the parse tree. Below is a figure depicting an example hierarchy stack as it is built while parsing a user-defined function –



**Figure 3.2 – Hypothetical view of Hierarchy Stack**

The figure above depicts an example of a Hierarchy Stack. When the function parsing starts, the root, the enclosing class, if any, and the current function are pushed on to the stack. Entries into the Hierarchy stack include pointers and type of entry, which is contained in a data structure, whose details are presented in the following sections. The parser considers an open brace ‘{’ as the start of scope. If the character ‘{’ is defined in quotes as a string or character literal like ‘{’ in source code, it is omitted. Only valid ‘{’ character with out any quotes is treated as a valid scope. There can also be anonymous scopes. Else every scope has an enclosing control type like if, for, else, else-if .etc. Once a valid scope is encountered, a *ScopeObj*, detailed in Section 3.2.2, is created on the heap and added to the parse tree. A pointer to the *ScopeObj* is pushed on to the stack. This maintains the information about the current scope that is being parsed. Whenever a valid closing brace, ‘}’, is encountered, it is treated as the end of scope. When end of scope is reached, all the properties of the *ScopeObj* are updated, like the end line number. The pointer to the *ScopeObj* that was stored on top of the hierarchy stack is popped.

### 3.1.2 Grammar Detectors

Determining source code structure involves identifying the semantics of the source code, through syntactical type analysis. This involves performing grammar checks to determine the source code elements and adding the user-defined types to grammar detectors. As we know, the entire source code structure is represented in our parse tree, and the parse tree is built in two passes. Like wise, there are two types of grammar checks – Type parsing and Function Parsing, which are performed in order to determine the entire structure of the source code. Each of these techniques have pre-defined language type checks for basic types such as integers, floats, double, vectors, sets etc.

Grammar detection also includes creating a repository of user-defined types such as classes or structures. Every time a new user-defined type is encountered, it is added to the repository. This process of determining the user-defined types is termed as type parsing. Type parsing also checks for data members and member functions. All basic type checks are included when determining the user defined types. Type parsing is done to build the first 3 levels of the parse tree.

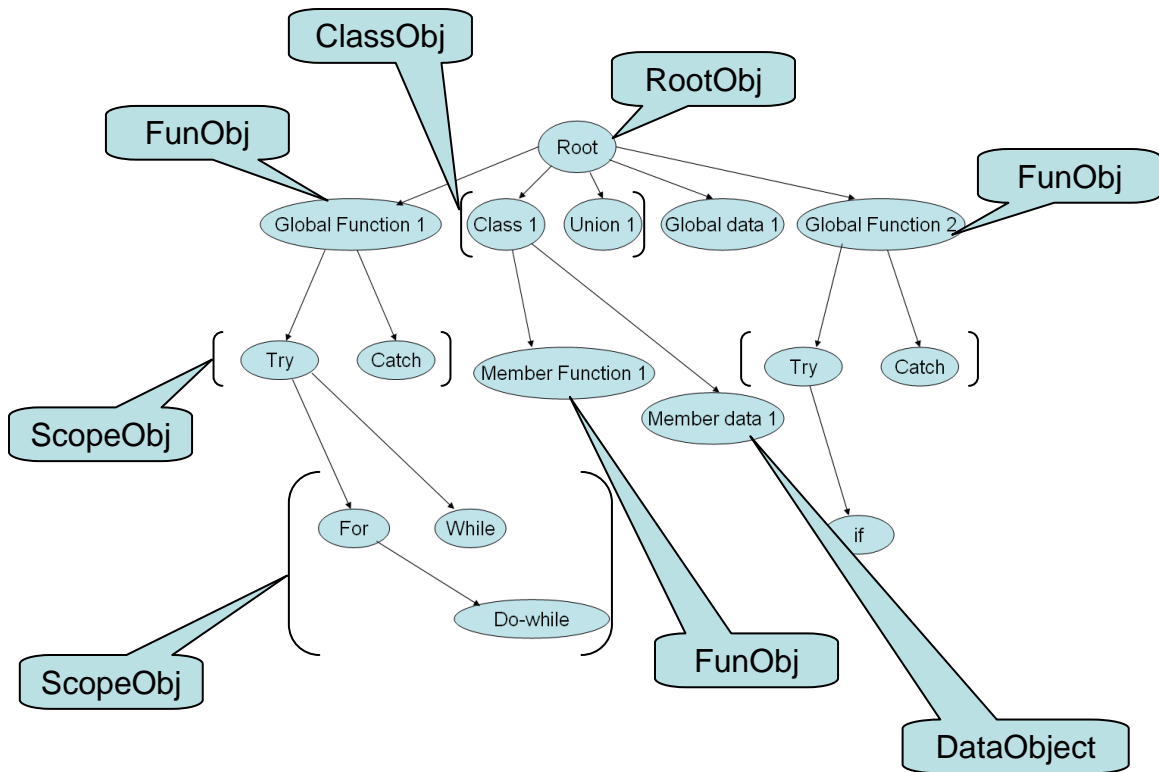
Function parsing is associated with completing the building of parse tree, downwards from level 3, which is done after the type parsing process is complete. Function parsing is carried out by parsing the functions, which include determining the local data and associating it with their respective scopes. This also gives us information about the life time and accessibility of data variables.

In both these processes, data is validated with the basic and derived types to determine their exact types.

### 3.1.3 Parse Tree Construction

The parse tree data structure is built dynamically and is done level by level. In this section, we discuss the building of the parse tree and its purpose. The parse tree is built to maintain the structure that exists in the source code. It is a connected data structure that is represented as an n-ary tree, with nodes forming the components of the tree and is connected by maintaining the relationships that are encountered by parsing the source code. The structure in the source code is represented in a hierarchical fashion that can be used to analyze the source code and identify feasible regions. Feasible regions are sections of source code which are best candidates for extracting functions from the source code. The parse tree maintains all the information that is required in understanding the structure of the source code and determining the feasible regions.

The parse tree is made up of connected components called ‘Nodes’. There are different types of nodes, depending on the program elements that are identified. The different types of nodes include a Root node, Class Node, Function Node and Scope Node. Each of these nodes, have common and specific properties, depending on the type of the node. Every Node type is associated with an object called the ‘Data Object’, which represents the data that is contained in or declared in that particular scope. The details of ‘Data Objects’ are described later in this chapter. All node types are derived from a common Root Node and there is only one Root Node in a parse tree. Also, there is only one parse tree for a Code Restructuring parse. Below is a sample figure of Parse tree –



**Figure 3.3 – Different types of Nodes in Parse Tree**

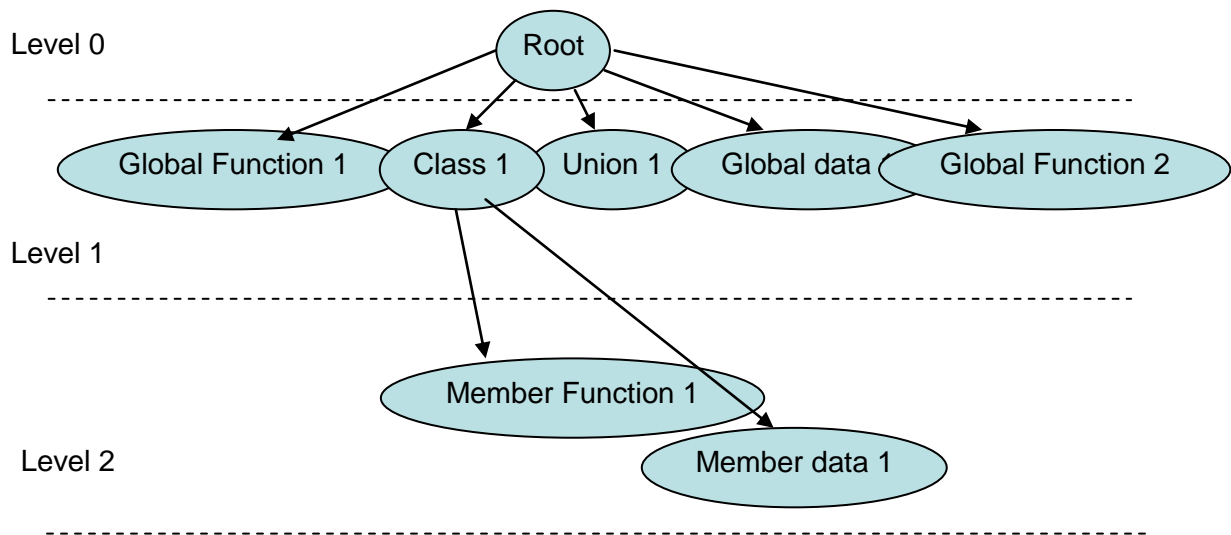
The representations in the above figure mention about *RootObj*, *ClassObj*, *FunObj*, *ScopeObj* and *DataObject*. These are different kinds of nodes that are represented as Objects in our approach in building the parse tree. More details about them are provided in the later sections, where we detail each of the Node Types.

Parsing the Semi-Expressions involve checking for language keywords and identifying the relevant source code expressions. The keywords that form the parse tree are User-Defined Variables, Functions and Types – Classes, Unions, Structures and Enumerations. The variable names and types are extracted from the Semi-Expressions and added to the parse tree as a node of that type. Initially, when the parsing begins, a root node of type *RootObj*, discussed in Section 3.2.2, is created. All other nodes are added to the parse tree deriving from the root node and maintaining the hierarchy. In the following section, we



discuss the building of parse tree from root node onwards, building the tree downwards from the root node.

The parse tree is built dynamically and is built in two stages. In the first stage, the first three levels of the tree are built. These include *RootObj*, Global Functions and Global Data nodes, *ClassObj* nodes, discussed in Section 3.2.2, which include User-Defined types like Classes, Structures, Unions and Enumerations, and member functions and member data. The parse tree up to this, form the first three levels in the data structure. The following figure depicts the first three levels of the parse tree and is built in the first parse.



**Figure 3.4 – Building of First Three Levels**

In the first parse of scanning the source file, the first three levels of the parse tree is built. The parser scans the Semi-Expressions extracted from the source code for built-in language keywords like 'class', 'int', 'for' and for User-Defined Functions. During this phase, the parse tree is built with User-Defined Global Functions and Global Variables, and User-Defined Types like Classes, Structures, Unions and Enumerations, and Member Functions and Member data. Levels are maintained as the structure in the source code where first the top level nodes like Global Functions and Global Data nodes are built, with User-

Defined Types. Also, member functions and data are added to the parse tree as child nodes to respective type nodes. All User-Defined types form the '*ClassObj*' node, all functions from the '*FunObj*' node, discussed in Section 3.2.2, and all Global Variables form the '*DataObject*' node, discussed in Section 3.2.2. Every Node type has a collection of child nodes. For Example, the Root Node has a collection of '*ClassObj*' nodes, and *ClassObj* node has a collection of member function '*FunObj*' nodes and member data '*DataObjects*'. The parse tree built up to this point finishes the first parse of the source code. The rest of the nodes are built in the next parse, where we focus on parsing source code functions from where we get the scope and data variables' information. The parse tree up to this point is built by *TypeParser* class, defined in the Section 3.2.1.

The next step in building the parse tree is to scan the defined functions in the source code. From this point onwards, the parse tree is built by *FunctionParser* Class, also discussed in Section 3.2.1. Scanning of functions gives us the scope information and information about variable life time and access regions. This part of building the parse tree is an extension to the first step. The parse tree is extended in this step to include rest of the information of the source code which includes information in functions. Other important information that is required in determining the location and usage of variables. From this step onwards, the focus is on building functions by parsing the source code. All data related information is stored in '*DataObjects*'. Each function has a collection of scopes, represented as '*ScopeObj*' and Function data represented as '*DataObjects*'. In the process of scanning functions, our technique maintains a hierarchy stack for keep track of scopes and the current location in the source code. The hierarchy stack, discussed in Section 3.1.1, maintains the information that's represented in the parse tree. Initially, the top of stack pointer will be

pointing to the current function, represented as '*FunObj*'. When the parsing of the function begins, a function object represented as '*FunObj*' is created on the heap and added to the parse tree. A pointer to this is pushed on to the stack. For all data variables declared in the function, '*DataObjects*' are created and are added to the function. In the earlier sections we had mentioned each of the function to maintain a collection of *DataObjects*, representing the data variables declared in that function. These variables can be used in anywhere in the function. All the properties associated with data are maintained in the '*DataObject*'. Once a valid scope is encountered, a *ScopeObj* is created on the heap and added to the parse tree. A pointer to the *ScopeObj* is pushed on to the hierarchy stack. This maintains the information about the current scope that is being parsed. Whenever a valid closing brace, '{', is encountered, it is treated as the end of scope. The pointer to the *ScopeObj* that was stored on top of the hierarchy stack is popped.

All the scopes in the function that is parsed are stored temporarily in the hierarchy stack and are processed. At the same time, the parse tree is built by synchronizing the push operations on to hierarchy stack and building the parse tree. The parse tree is completed by processing all the functions in the source code. Once we have completed parsing all the functions, we have all relevant information in understanding the structure of the source code. The parse tree is built on the heap and is persistent through out the execution of the program.

### **3.1.4 Parsing Data Declarations and Saving Data Spans**

Referring back to Section 3.1.2, we have discussed about types of parsers used in detecting scopes and data. The process of detecting scopes and data is done simultaneously. The determination of usage and spans of data is done during function parsing process, with the help of hierarchy stack, discussed in section 3.1.1. Every User-Defined type, function or

scope has a collection of *DataObjects*, that are declared in their scope. The reference point of data spans and usage is its line number. Every *DataObject* has in its property a collection of line numbers where it is used. In this section, we discuss how this information is got and stored in the respective *DataObject*.

The hierarchy stack represents the current scope being parsed. For example, if we are parsing a method of a class, the contents of the stack include the root, the class and the current method. All the entries in the stack are objects and maintain their collection of *DataObjects*. When ever, a data declaration of any type is encountered – basic or user-defined type, a new *dataObject* is created on the heap and added to the current scope's collection of *dataObjects*. However, the source code line contains usage of a data variable; the variable is searched in the current scope's collection of *DataObjects*. If the variable is found, its usage is updated in the *dataObject*'s properties. If the variable is not found, then a copy of the hierarchy stack is created and the first element; which represents the current scope, is popped out.

The copy of the stack now contains the enclosing scopes of the current scope. The variable is searched in every scope, by searching and popping the elements from the hierarchy stack's copy. Once the variable is searched, the *dataObject*'s properties are updated. By this, we determine all the sections of the source code where a data variable is used.

As mentioned earlier in the earlier sections, every Node in the parse tree maintains a collection of *DataObjects* that are declared in their scope. All properties of all data variables declared in a scope are maintained by *DataObjects*. This enables us to maintain all information of all data variables. Information about *DataObjects* in other scopes can be

obtained by message passing mechanism, in object oriented terms. All *DataObjects* in a Node of the parse tree are data variables local to that scope. That is the data for which there is access in that node scope. For example, all *DataObjects* in a Function Node are local data in that function and can be accessed anywhere in the function. Below is a figure depicting the relation between any node type and

DataObjects –

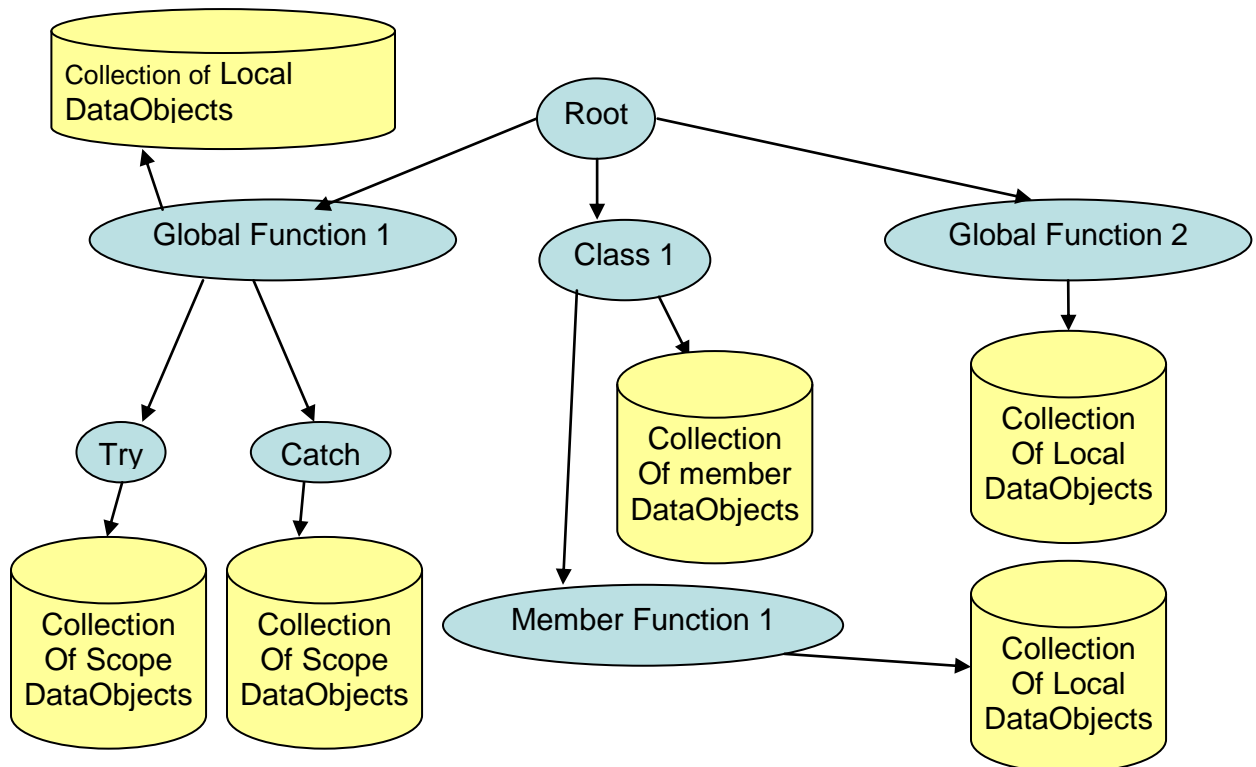


Figure 3.5 – Top Level Containment diagram of Parse Tree

As we can see from the above figure, every node type maintains a repository of *DataObjects* that are declared in their scope. All the properties associated with data variables are stored in *DataObjects*. This information is used when identifying feasible regions for extracting functions, which gives us the idea about how many data variables have to be passed when we are pulling functions. Optimization techniques can be applied based on the line count of

feasible regions and the number of *DataObjects*. More information about feasible region selection and function extraction is discussed in the Section 3.2.3.

### 3.1.5 Selecting Feasible Regions

In this section, we discuss the process of identifying feasible regions, the criteria for determining feasible regions, the types of algorithms and using feasible regions.

Identifying feasible regions is carried out after the parse tree is created and by scanning the properties of nodes. The properties that are considered for each node is span of the control type, which is the number of lines of a particular scope or function node spans. This is the first step for short listing the candidate source code regions as ‘feasible’ regions. The maximum number of lines that a feasible region can span is decided by the user and is passed as a command line argument. Below is a figure depicting the identification of candidate ‘feasible regions’ –

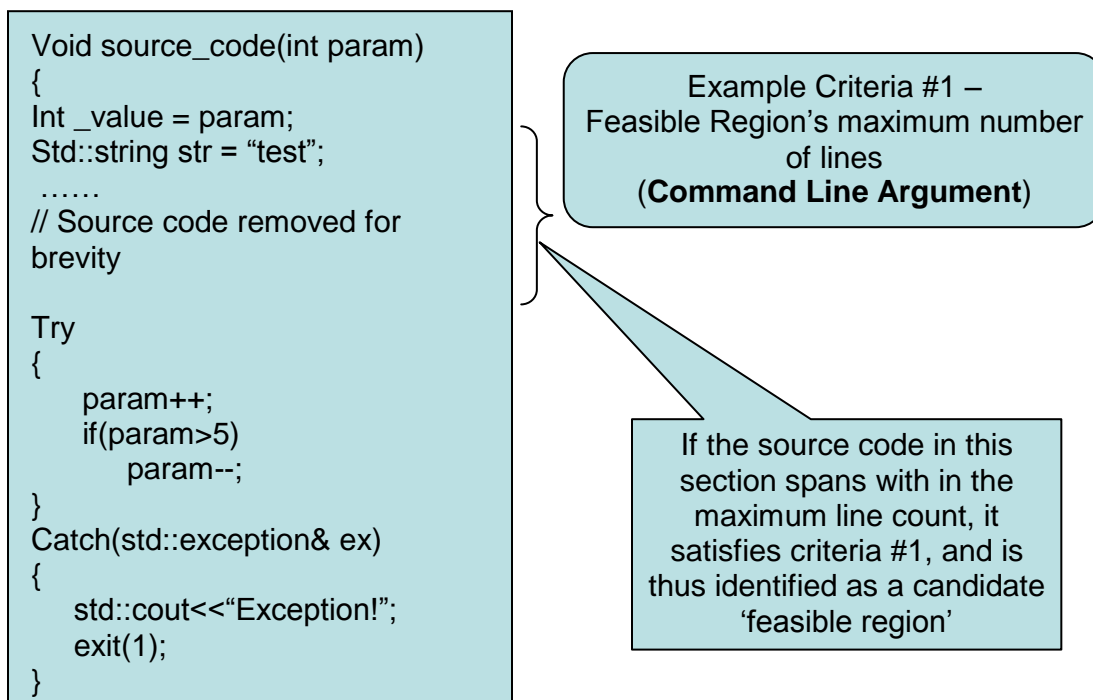


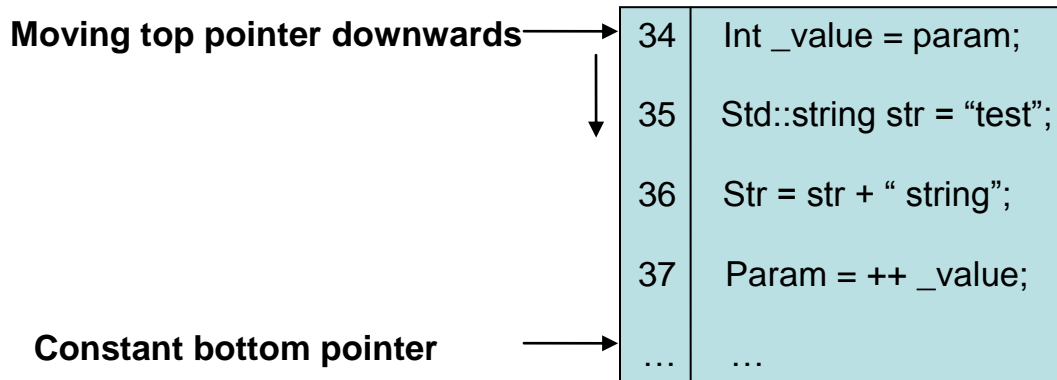
Figure 3.6 – Line number criteria for Feasible Regions

Once a linear section of source code is identified, which is with-in the maximum number of lines a feasible region can span, it satisfies the first criteria of maximum line count. The next step is to determine the number of arguments to be passed, if a new function is extracted from the identified feasible region. In the first step of identifying feasible region, we short listed a section of source code from which a feasible region may be extracted. In this step, we determine whether the short listed section of source code, meets the second criteria of number of parameters to be passed, if a new function is extracted from that section of source code. There are three methods of identifying a feasible region from a candidate section of source code – Top Down, Bottom up and two-way approach. In all three approaches, the maximum number of parameters that may be passed to an extracted function is passed as a command line argument. Below is a discussion of all three approaches.

### **3.1.5.1 Top down Approach**

In this approach, the identified candidate feasible region is analyzed for the number of parameters to be passed if a section of this source code is extracted as new function, in a top down fashion. From the identified feasible region, two pointers are used to analyze source code – top and bottom pointer. The top pointer points to the first line in the code fragment where as the bottom pointer points to the last line in the code fragment. The bottom pointer is kept constant pointing to the last line in the code fragment. The top pointer is moved downwards, one line at a time and in each step, the number of parameters to be passed from that line to the end of code fragment is determined. The first criteria is to satisfy the condition that the number of parameters at any point is less than the maximum number of parameter count, which is passed as a command line argument. The section of code with least number

of parameters to be passed or that which satisfies the maximum parameter limit is identified as a feasible region. Below is a figure depicting this approach –



**Figure 3.7 – Top down approach for determining parameters**

As we can see from the above figure, the bottom pointer is kept constant and the top pointer is moved downwards, one line at a time and determining the number of parameters to be passed in each step.

### **3.1.5.2 Bottom up Approach**

In this approach, the top pointer, pointing to the first line of the code fragment is kept constant and the bottom pointer is moved up, one line at a time. In each step, the number of parameters to be passed, if a function is extracted, is determined. The number of parameters to be passed should not exceed the maximum limit that is passed as a command line argument. Below is a figure depicting this approach –



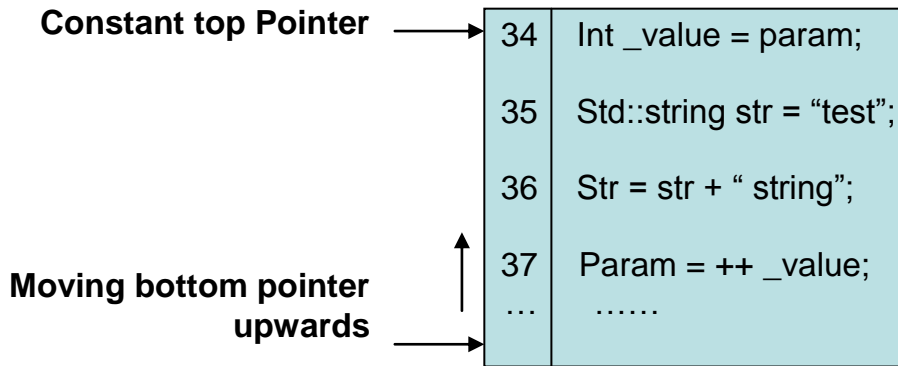


Figure 3.8 – Bottom up approach for determining parameters

As we can see from the above figure, the bottom pointer is moved up, one line at a time and the top pointer is constantly pointing to the first line in the feasible area.

### 3.1.5.3 Two-way Approach

In this approach, both the top and bottom pointers are moved one line at a time, downwards and upwards respectively. The top pointer is moved downwards one line at a time, at the same time moving the bottom pointer upwards one line at a time; and determining the number of parameters to be passed if a function is extracted from that location in each step. Below is a figure depicting this approach –

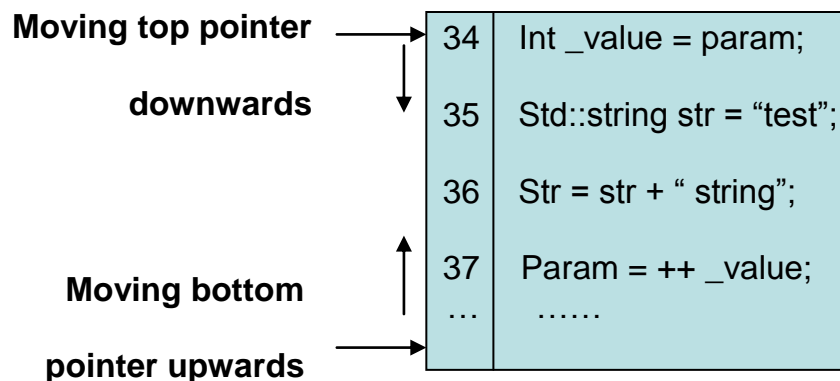


Figure 3.9 – Two-Way approach for determining parameters

As we can see from the above figure, both the top and bottom pointers are moved towards each other, one line at a time. At each step, the number of parameters to be passed if a new function is extracted at that location is determined.

All the three approaches discussed above can be used for analyzing a feasible region.

However, for our implementation we use top down approach. The identified feasible region should satisfy the maximum line number and maximum parameter count which is entered by the user, through command line arguments. Once feasible regions are identified, all information of the feasible region is stored in a data structure which is used when restructured code is generated. More information about the data structures used are discussed in Section 3.2.3

## ***3.2 Implementation***

In the previous sections, we have discussed the topics involved in parsing source code and saving source code information in a format used during code restructuring. In this section, we present the implementation details of all stages and data structures used in our approach.

### **3.2.1 Implementing Grammar Detectors**

Grammar detectors are used in Code Restructuring to identify source code elements that represent the source code structure and, which are used to build the parse tree. The building of parse tree is discussed in detail in the following sections. In this section, we discuss the details about parsers that are used and how they work in determining the parse tree elements.

There are two types of Parsers that are used in grammar detection and building of parse tree – *TypeParser* and *FunctionParser*. There is also a *Utility* class that is used by both the parsers in type checking and determining the keywords and types. The parse tree is built in two phases – First, the first three levels of the tree, the root, Global functions and data, and user-defined types with member functions and data are built. Next, the functions are parsed to complete the rest of the tree.

*TypeParser* is used in the first step in parsing to build the first three levels of the parse tree. *TypeParser* class makes use of *Utility* Class for type checking and building of parse tree. *FunctionParser* also makes use of *Utility* Class for type checking, determining the scope objects and building the parse tree. The *dataObjects* are built dynamically in every stage and the creation of *dataObjects* and setting their properties is done by both the parsers with the services from *Utility* Class. Below is a class Diagram depicting the relations that exist between *TypeParser*, *FunctionParser* and *Utility* Class –

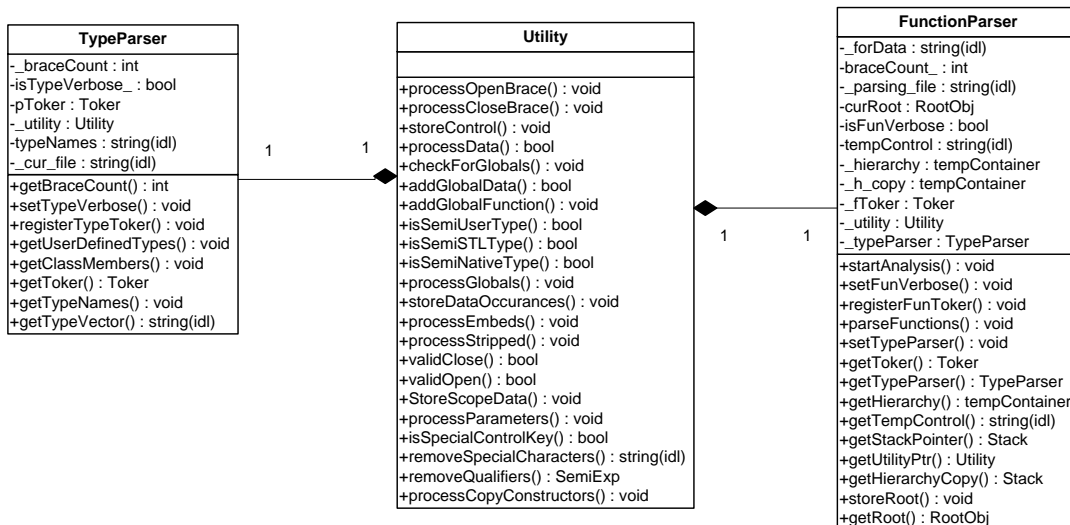


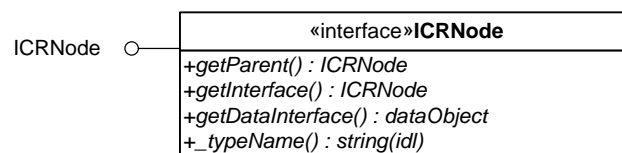
Figure 3.10 – Class Diagram of Parsers using Utility Class

As we can see from the above figure, both the parsers make use of Utility Class for type checking and identifying language keywords. Once the type checking of Semi-Expressions is done, appropriate object types are created on the heap and added to the parse tree. The identification of feasible regions is done partially during type analysis and the FunctionParser class contains pointers to identified feasible regions. The complete description and building of parse tree are briefed in the following sections.

### 3.2.2 Implementing Parse Tree

We have seen in Section 3.1.3, that the parse tree is a collection of connected nodes. Every node has its own unique properties, based on the type of the node. In this section, we discuss the implementation details of all the node types that form the parse tree.

Every Node is represented as an object that is persistent through out the execution of the program. There are different classes representing different Node types. Hence, we have a Class Node, Function Node and a Scope Node. Each of the nodes, have ‘Data Objects’ that are contained in their class. All node classes are derived from the Interface ‘ICRNode’. ICRNode is an interface with method declarations that are common to all node types and the implementation to which have to be provided by respective classes that are derived from ‘ICRNode’. Below is a class diagram of the Interface ‘ICRNode’ –

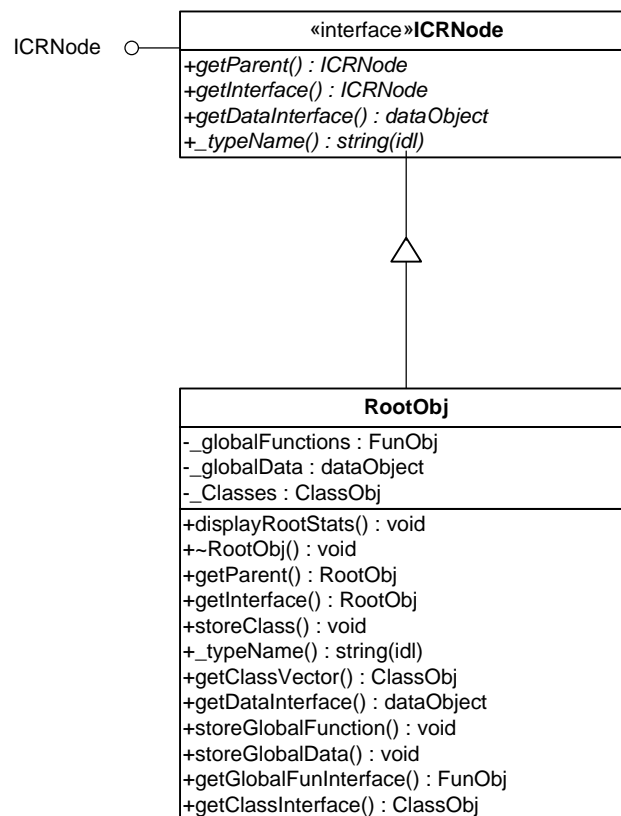


**Figure 3.11 – Class Diagram of ICRNode Interface**

ICRNode is the interface for any type of Node. The Root Node which is the parent node for all other nodes is derived from the interface ICRNode. All other types of nodes are derived

from the Root Node. And hence, all the nodes that are derived from Root Node have to implement the pure virtual functions declared in the interface, including the Root Node. Each of the node types, have their own version of implementations for the pure virtual functions. In addition to implementing the pure virtual functions, there are additional functions for each of the node classes that are specific to that particular node type. In the following section, we detail the class implementation of each of the node types. We start with Root Node, which is derived from all of the other node types. In our implementation of Root Node, we term the class representing the root node as 'RootObj'.

Below is the class diagram of 'RootObj' –



**Figure 3.12 - Class Diagram of RootObj**

As we can see from the above figure, the RootObj class is derived from the interface 'ICRNode' and implements all the pure virtual functions that are declared in the interface.

The RootObj Node Class has global data and global functions in addition to the implemented functions of ICRNode. The behavior of RootObj class is similar to a base class for other Node types. All other Node types are derived from RootObj Node. Below we will start with a top level Class Relationship diagram depicting the relationships that bind RootObj, ClassObj, FunObj, ScopeObj, and the DataObject Classes. In the following sections we brief all the classes in detail.

Top Level Class Relationship Diagram of All Objects –

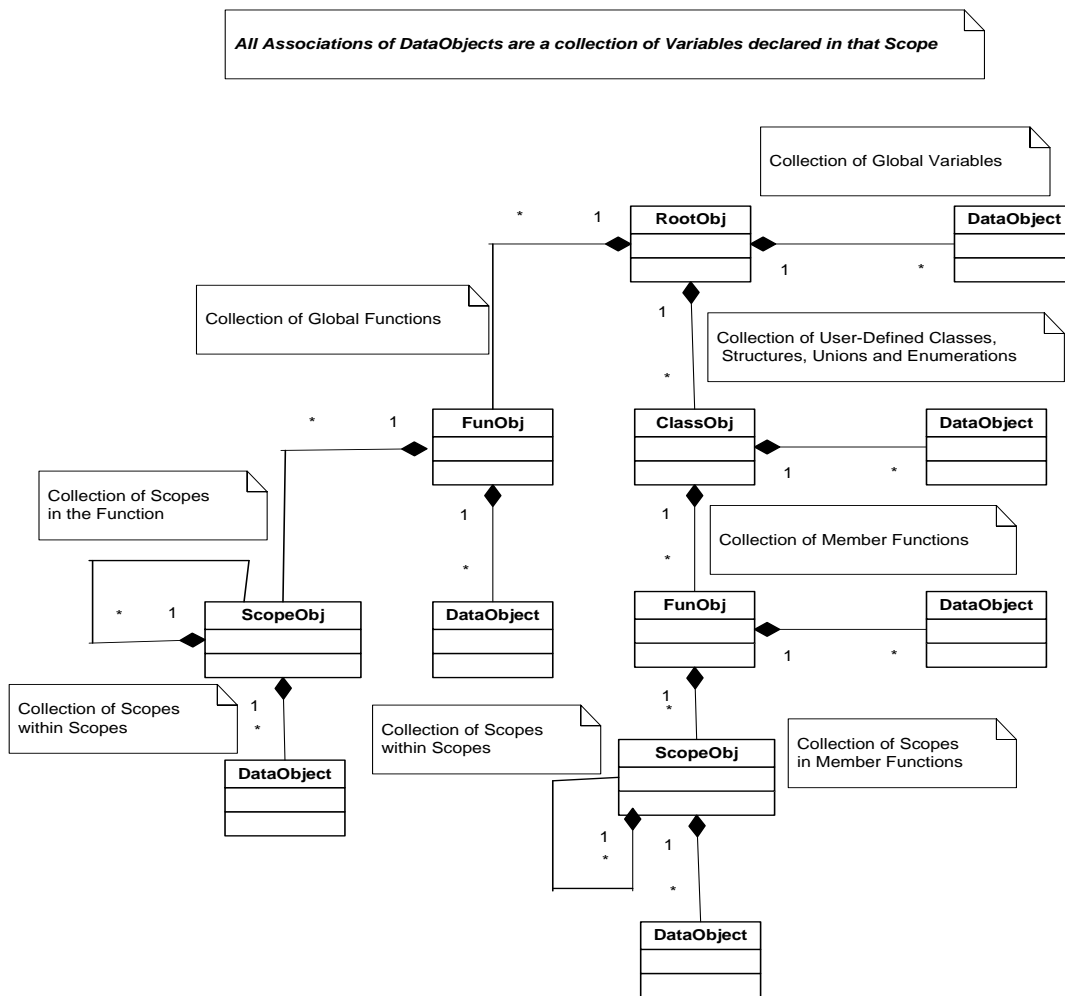


Figure3.13 – Class Relationship diagram of Parse tree Objects

Below is a figure representing ClassObj, FunObj and ScopeObj Classes –

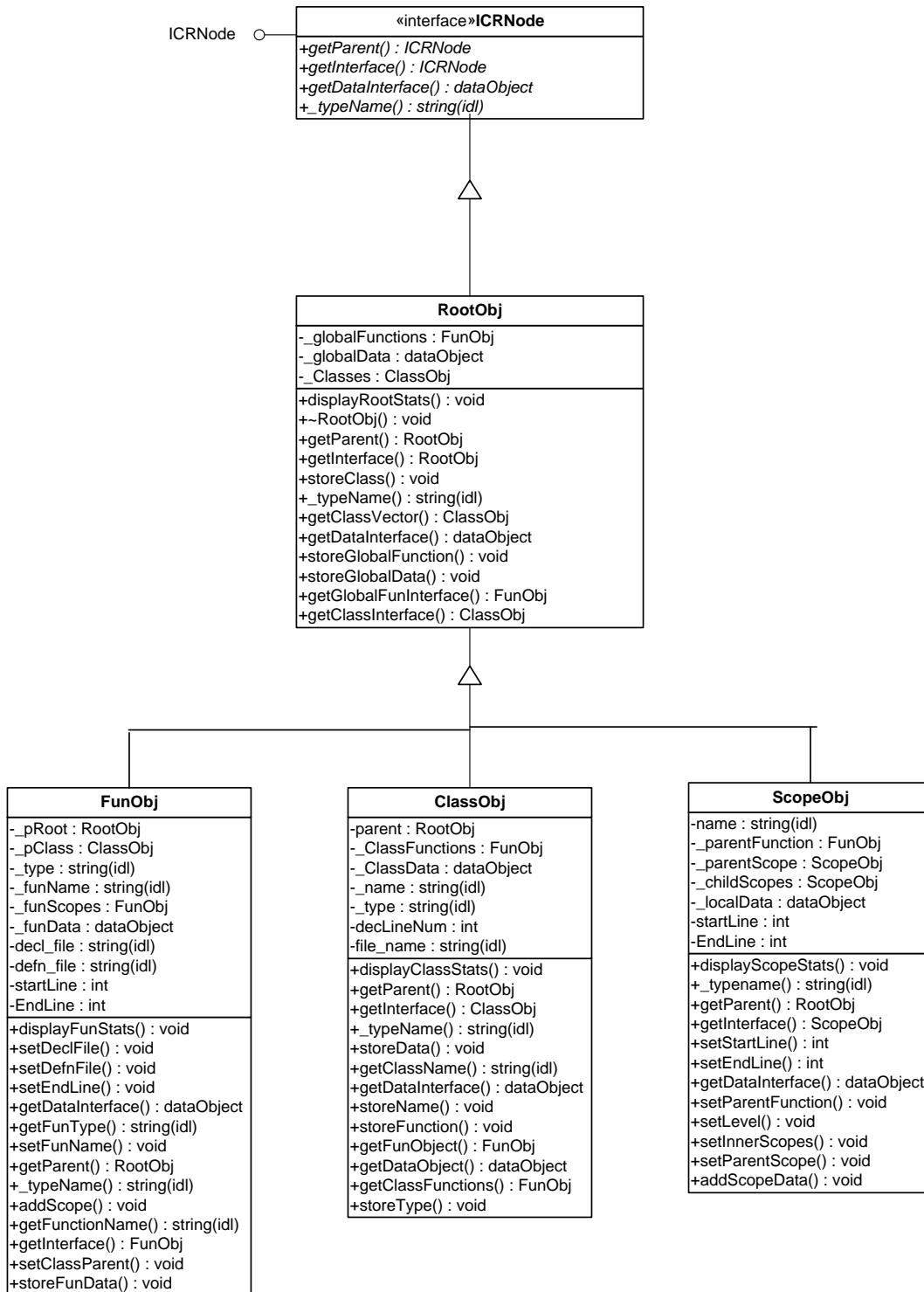


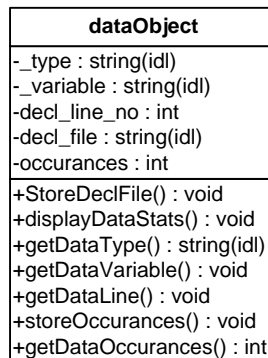
Figure 3.14 – Class Diagram of Different Node Types

The above figure is a top level Class Diagram of different Node Types. The different Node types are Root Node, Class Node, Function Node and Scope Node. Each of the Node types are represented as RootObj, ClassObj, FunObj and ScopeObj respectively. RootObj Class acts as a base class for all other node types. All other node types are derived from RootObj and implement the interface ICRNode. Each of the node types have their own implementations of the function declarations in the ICRNode interface. In addition to that, each of the node types, have additional properties associated with their node type, with functions to access these properties. All of these node types are represented as Objects and are created and added to the parse tree when encountered during parsing of the source code. The process of parsing the source code and building the parse tree is discussed later in this chapter.

All of the node types have a collection of DataObjects. DataObjects represent the data is declared in respective node's scope. By Scope we mean a boundary in the source code where a variable can be accessed. DataObjects are also represented as classes with all the required properties. There are a collection of properties that are associated with a DataObject. DataObjects are used to determine the data that is bound to a section of source code. In the process of Code Restructuring, decisions are made to determine feasible regions which are the candidates for extracting functions. Since each of the node types are used as a reference to determine feasible regions, DataObjects are used to shortlist feasible regions from candidate feasible regions, based on the data is bound to a node. The discussion of identifying feasible regions and extracting functions are detailed in next Section.



Below is the class diagram of DataObject class –

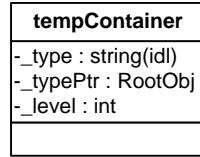


**Figure 3.15 – Class Diagram of DataObject**

The above figure shows the data members and member functions of dataObject class. Every object of dataObject class represents a data variable that is declared in the source code. A dataObject is created by the parser when a variable declaration is encountered. The variable can be of any type – a basic type or a user-defined type. Based on the type of the variable, a dataObject of that particular type is created. DataObject objects have supporting properties to determine the type of the variable, variable name, the file it is declared and all the places it is used in the source file. The reference for usage is line numbers in source file.

We have discussed about hierarchy stack in Section 3.1.1. Every entry into hierarchy stack is an object, which contains a pointer to the scope object and contains information about the type of pointer. This object type is called ‘tempContainer’. The temporary Container is a structure to hold node type and pointer information that is maintained in the stack.

Below is the class of diagram of our temporary Container –

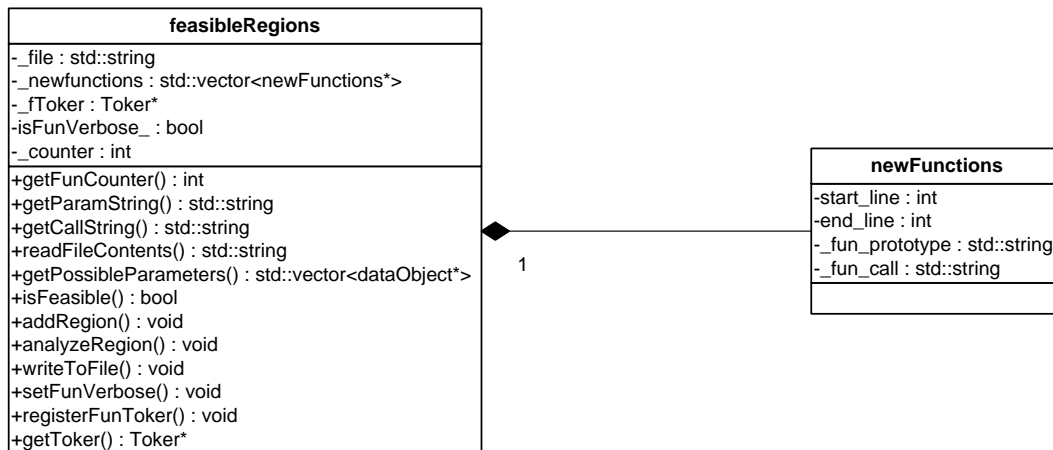


**Figure 3.16 – Class Diagram of TempContainer**

The variable ‘\_typePtr’ in the definition of the structure is a pointer to the node object and the variable ‘\_type’ determines the type of node. When objects are added and removed from stack, all push and pop items are of type ‘TempContainer’. The variable ‘\_type’ in *tempContainer* determines the type of node at runtime and suitable dynamic cast operation is performed to type-cast the object pointer to its respective type.

### 3.2.3 Implementing Code Restructuring

Feasible regions that are identified to be extracted as new functions, are stored on the heap for look up when new restructured files are written to the disk. For the purpose of keeping track of all the necessary information about feasible regions, the data structures – *feasibleRegions* and *newFunctions* are used. All properties required while generating restructured code are stored in these data structures and is retrieved when new files are created. Below is a class diagram depicting the components and relationship between these classes.



**Figure 3.17 – Class Relationship diagram of feasibleRegions and newFunctions**

As we can see from the above figure, *feasibleRegions* class contains a collection of identified feasible regions. This collection is built on the heap and is maintained till the complete parsing of the source file is done. Once the complete parsing of source file is done, this collection is used to extract new functions and write new restructured files. *FeasibleRegions* class also contains methods that are used in determining whether the criteria of number of lines and parameters meet the requirement. Once a region satisfies these criteria, a new object of type *newFunctions* is created on the heap and stored till the complete parsing of source code is complete. In the next section, we discuss how this collection of *newFunctions* objects is used.

### 3.2.3.1 Writing new restructured files

In this section we discuss the procedure of extracting feasible regions from source code as new functions, and writing new restructured files. Once the parsing of source file is complete, a collection of feasible regions are identified which form newly extracted functions. This is the last step in code restructuring process. Parsing of source code functions

is carried out by *FunctionParser* class, about which we have discussed in the earlier sections. For the purpose of reading source files and create new restructured files, a new class *fileManager* is used, which works with *feasibleRegions*. Below is a class relationship diagram of *FunctionParser*, *feasibleRegions*, *newFunctions* and *fileManager* –

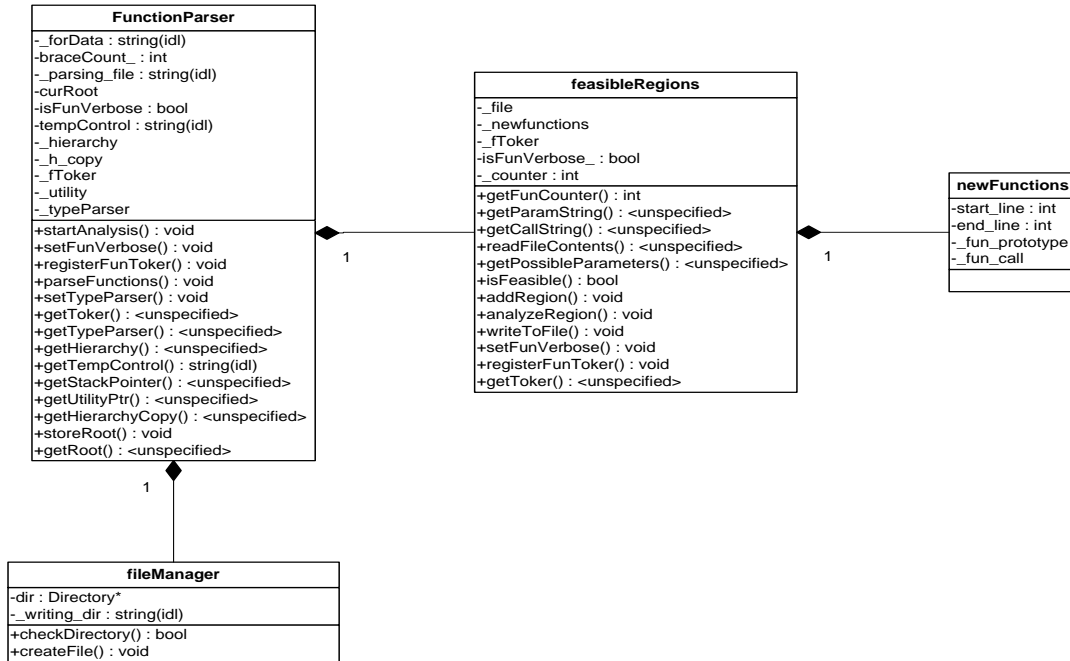


Figure 3.18 – Class Relationship diagram of *FunctionParser* and *fileManager*

As we can see from the above figure, the *FunctionParser* class contains a pointer to *feasibleRegions* object, which in-tern contains a collection of newly extracted functions. Once the parsing of source code is complete, and feasible regions are identified, the last step is to create new restructured files. The newly extracted functions are named by source file name followed by an incrementing integer. For example, if the source file is ‘test.cpp’, then the extracted functions would be named ‘test\_1’, ‘test\_2’ etc. After parsing of source file is complete, the *FunctionParser* parser makes use of *feasibleRegions* and *fileManager* object

to read source file again and write new restructured files, using the information of feasible regions' line numbers stored in *newFunctions* class.

The *feasibleRegions* class is used by *FunctionParser* to read the contents of the source files in a restructured fashion, by getting the properties of feasible regions from *newFunctions* class. *FeasibleRegions* class reads the contents of the source file by writing newly extracted functions and making appropriate function calls to new functions in place where those feasible regions existed in the source code. Once the restructured source file contents are created by reading through the source code and making appropriate changes, *fileManager* class is used to create a directory in the current directory of the source file with the name ‘\_Restructured’. Once the directory is created, the restructured file is created with the name of source file followed by ‘\_restructured’. For example, if the source file name is ‘test.cpp’, then the restructured file name would be ‘test\_restructured.cpp’. The contents of restructured file, which is created by *feasibleRegions* and *newFunctions* objects, are passed to *fileManager* object which writes the contents to disk, completing the code restructuring process.

## Chapter 4 Semi-Automated Restructuring Results

Code Restructuring proceeds by analyzing source code and finding suitable large functions from within which to create new functions, and maintains the original external behavior of the source code. Our implementation performs code restructuring for global functions and member functions of user-defined types such as Classes and Structures. As mentioned before, we follow the naming convention calling a global function as ‘function’ and a member function as ‘method’. New functions and methods are extracted by identifying feasible regions in the source code, which we discussed in Chapter 3. In this section we present a few examples of code restructuring performed on real source code functions and methods, by our prototype.

### ***4.1 Restructuring Functions***

Global Functions are user-defined functions declared and defined in the global scope and can be accessed by any other function or object that has access to the function declaration and definition. Code Restructuring of Global Functions follows the same conventions and nearly the same procedures as member functions. The newly extracted functions from Global functions are also in global scope and are, themselves, global functions. They can be accessed by any other function or object with access to the original. The newly extracted functions are written in the same file where the parent global function is defined. In the place where the extracted code fragment was found, a function call to the newly extracted function is made. All parameters that have to be passed to the newly extracted function are passed by reference to maintain the function’s behavior. The restructured file is written to

‘\_Restructured’ directory and is named as the original file name followed by ‘\_restructured’, and with the same file extension.

Considering a global function, let’s see some results of how restructuring is performed and what parameters have to be considered during restructuring. We present both the cases where in, no arguments are passed to the newly extracted function and arguments are passed to the new function.

### 4.1.1 Extracting functions with no parameters

Below is the source code of a global function – setRootValues (), taken from the source code of our framework:

```
void setRootValues()
{
    try
    {
        std::string inFile = getInputFile();
        Directory dir;
        Scanner scanr;
        scanr.doRecursiveScan(inFile);
        dir.RestoreFirstDirectory();
        if(dir.dirContainIncludes())
            scanr.setFileIncludes();
        std::vector<std::string> _files = getCompleteFiles();
        if(_files.size() > 0)
        {
            RootObj* root = new RootObj();
            std::string _type = root->_typename();
            if(_type == "")
                _type = "pRoot";
            root->displayRootStats();
        }
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() <<std::endl;
    }
}
```

Performing code restructuring on the above function, using our prototype, we extracted a new function, `setRootValues_1()`. Below is the source code of the extracted function and changes that appear in the parent function, `setRootValues`.

```
void setRootValues_1()
{
    std::string inFile = getInputFile();
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    if(dir.dirContainIncludes())
        scanr.setFileIncludes();
}

void setRootValues()
{
    try
    {
        setRootValues_1();
        std::vector<std::string> _files = getCompleteFiles();
        if(_files.size() > 0)
        {
            RootObj* root = new RootObj();
            std::string _type = root->_typename();
            if(_type == "")
                _type = "pRoot";
            root->displayRootStats();
        }
    }
    catch(std::exception& ex)
    {
        std::cout<< ex.what() <<std::endl;
    }
}
```

From the above source code, we notice that a new function ‘`setRootValues_1`’ is extracted from the parent function ‘`setRootValues`’. In place where the body of extracted function was present in ‘`setRootValues`’, before restructuring, a function call has been made to the new function. We also notice that no arguments have been passed to the new function.



## 4.1.2 Extracting functions with one parameter

When the new function, that is extracted, requires arguments to be passed, there are 3 scenarios that exist, that have to be considered for proper restructuring. In order for the restructured code to compile and maintain the original behavior, it's important to consider these scenarios which we'll discuss below while passing arguments to new functions.

### 4.1.2.1 Scenario 1 – Argument declared before Function Call

In this section, we present an example where the extracted function requires one parameter to be passed as an argument. Below is the source of a function –

```
std::string& removeSpecialCharacters(std::string& _temp)
{
    try
    {
        if(_temp.find("*")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'*');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find("&")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'&');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find("<")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'<');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find(">")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'>');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find(":")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),':');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find("~")!=-1)
```

```

    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '~');
        _temp.erase(junk2, _temp.end());
    }
    if(_temp.find("(")!=-1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '(');
        _temp.erase(junk2, _temp.end());
    }
    if(_temp.find(")")!=-1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), ')');
        _temp.erase(junk2, _temp.end());
    }
    if(_temp.find(";")!=-1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), ';');
        _temp.erase(junk2, _temp.end());
    }
}
catch(std::exception& ex)
{
    std::cout << ex.what() << std::endl;
}
return _temp;
}

```

Below is the restructured code for the example presented above.

```

void removeSpecialCharacters_1(std::string& temp)
{
    if(_temp.find "*" != -1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '*');
        temp.erase(junk2, _temp.end());
    }
    if(_temp.find "&" != -1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '&');
        temp.erase(junk2, _temp.end());
    }
    if(_temp.find "<" != -1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '<');
        _temp.erase(junk2, _temp.end());
    }
    if(_temp.find ">" != -1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), '>');
        _temp.erase(junk2, _temp.end());
    }
    if(_temp.find ":" != -1)
    {
        std::string::iterator junk2 = remove(_temp.begin(), _temp.end(), ':');
        _temp.erase(junk2, _temp.end());
    }
}

```

```

}
if(_temp.find("~")!=-1)
{
    std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'~');
    _temp.erase(junk2,_temp.end());
}
if(_temp.find("(")!=-1)
{
    std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'(');
    _temp.erase(junk2,_temp.end());
}
}

```

The changes that apply in the calling function are –

```

std::string& removeSpecialCharacters(std::string& _temp)
{
    try
    {
        removeSpecialCharacters_1(_temp);
        if(_temp.find("&")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(),'&');
            _temp.erase(junk2,_temp.end());
        }
        if(_temp.find(";")!=-1)
        {
            std::string::iterator junk2 = remove(_temp.begin(),_temp.end(';'));
            _temp.erase(junk2,_temp.end());
        }
    }
    catch(std::exception& ex)
    {
        std::cout << ex.what() << std::endl;
    }
    return _temp;
}

```

In this scenario, we notice that an argument ‘\_temp’ is passed to the newly extracted function. This variable is passed to the parent function ‘removeSpecialCharacters’ as an argument. We also notice that the variable is passed by reference, so that the changes made in the new function would reflect in the calling function.

### 4.1.2.2 Scenario 2 – Local Variable passed to new function

Considering a scenario where a local variable declared inside a function has to be passed to the newly extracted function. This scenario is similar to the one discussed earlier, but the variable being passed is a local variable declared in the parent function and used in the new function. Below is the source code of a function:

```
std::string& processParams(typeParser* _parser, SemiExp& se, FunObj* _fun)
{
    std::string tStr = "";
    utility* util = new utility();
    se = util->removeQualifiers(se);
    std::string _return = "";
    se = util->removeQualifiers(se);
    tStr = se.showSemiExp();
    std::string _par(tStr.substr(tStr.find("(")+1));
    tStr = tStr.substr(tStr.find("(")+1, _par.find(")")-1);
    Toker* pToker = new Toker(tStr, false);
    SemiExp _semix(pToker);
    if(pToker == 0)
    {
        delete pToker;
    }
    while(_semix.getSemiExp())
    {
        if(CheckParaData(_semix))
        {
            int num_params = 1;
            int num_commas = getOccurances(",");
            num_params = (num_commas==0)?1:num_commas+1;
            char* params = NULL;
            params = new char[num_params];
            if(num_params > 0)
                _return.append(getParamsList());
            else
                _return.append(" ");
        }
    }
    return _return;
}
```

The corresponding restructured code is:

```
void processParams_1(std::string& _return)
{
    int num_params = 1;
    int num_commas = getOccurances(",");
    num_params = (num_commas==0)?1:num_commas+1;
```

```

char* params = NULL;
params = new char[num_params];
if(num_params > 0)
    _return.append(getParamsList());
else
    _return.append(" ");
}

std::string& processParams(typeParser* _parser, SemiExp& se, FunObj* _fun)
{
    std::string tStr = "";
    utility* util = new utility();
    se = util->removeQualifiers(se);
    std::string _return = "";
    se = util->removeQualifiers(se);
    tStr = se.showSemiExp();
    std::string _par(tStr.substr(tStr.find("(")+1));
    tStr = tStr.substr(tStr.find("(")+1, _par.find(")")-1);
    Token* pToker = new Token(tStr, false);
    SemiExp _semix(pToker);
    if(pToker == 0)
    {
        delete pToker;
    }
    while(_semix.getSemiExp())
    {
        if(CheckParaData(_semix))
        {
            processParams_1(_return);
        }
    }
    return _return;
}

```

The variable ‘\_return’ is a local variable declared in the function ‘processParams’.

The extracted function requires using this variable, and is passed by reference to the new function.

### 4.1.2.3 Scenario 3 – Use of extracted variable

In this section, we discuss the scenario where a new function cannot be extracted, as a variable that is declared in the extracted function needs to be used in the calling function. In such a case, the analyzing region would fail to be determined as feasible. For this, let’s consider the source code in the previous section and try to restructure it again. Below is a

section of source code, from function ‘processParams’ that the parser determines to be a feasible region, based on the number of lines and the arguments that have to be passed, if a new function is extracted.

```
std::string tStr = "";
utility* util = new utility();
se = util->removeQualifiers(se);
std::string _return = "";
se = util->removeQualifiers(se);
tStr = se.showSemiExp();
std::string _par(tStr.substr(tStr.find("(")+1));
tStr= tStr.substr(tStr.find("(")+1,_par.find(")")-1);
Toker* pToker = new Toker(tStr,false);
SemiExp _semix(pToker);
```

If we see the source code in the previous section, Section 4.1.2.2, we see that the variables ‘pToker’, ‘\_semix’ and ‘\_return’ are used after the considered feasible region, shown above. We can’t consider this as a feasible region, as making this as a new function will make the source fail to compile. This is because the declaration of variables will be in the newly extracted function, which would be used in the calling function. Because of this, we consider this as a non feasible region, although other criteria for determining a feasible region are satisfied.

### 4.1.3 Extracting functions with many parameters

We have seen new functions being extracted with one argument being passed to the new function from the calling function. When working with source code, it is sometimes likely for newly extracted functions to have more than one argument. In this section, we

discuss passing of many parameters to the newly extracted function. The rules and scenarios are same as ones discussed in the previous section. Below is the source code of a function from our prototype source code:

```

bool NativeTypeCheck(typeParser* _parser, SemiExp& se, FunObj* _fun)
{
    utility* util = new utility();
    se = util->removeQualifiers(se);
    const int NUMKEYS = 8;
    static std::string KeyTable[NUMKEYS] = { "int", "float", "double", "char",
                                             "string", "short", "long",
                                             "bool"};

    std::string _temp="";
    if(se.find("=")!=se.length())
    {
        for(int i = 0; i < se.find("="); i++)
            _temp.append(se[i] + " ");
    }
    else
        _temp = se.showSemiExp();
    Toker* pTok = new Toker(_temp, false);
    SemiExp _semi(pTok);
    if(pTok == 0)
    {
        delete pTok;
        return false;
    }
    while(_semi.getSemiExp())
    {
        for(int i=0; i<NUMKEYS; i++)
        {
            std::string _key = KeyTable[i];
            int _ind3 = _semi.find(_key);
            if(_ind3!=_semi.length() && !(util->isSTLTypeCheck(_semi)) &&
                _semi[_semi.find(_key)+1]!="(" &&
                _semi.find("iterator")==_semi.length())
            {
                bool isPtr = false;
                if(_semi[_semi.find(_key)+1] == "**")
                    isPtr = true;
                std::string variable = util->getVariableName(_key, _semi);
                dataObject* dObj = new dataObject
                    (_key, isPtr, se[se.find(_key)], variable,
                    _parser->getToker()->lines()+1);
                dObj->StoreDeclFile(_parser->_cur_file);
                _fun->storeFunData(dObj);
            }
        }
    }
    delete pTok;
    return false;
}

```

Below is the restructured code –

```
void NativeTypeCheck_1(FunObj& _fun, SemiExp& se, std::string& _key,
SemiExp&
    _semi,utility& util, typeParser& _parser)
{
    int _ind3 = _semi.find(_key);
    if(_ind3!=_semi.length() && !(util.isSTLTypeCheck(_semi)) &&
        _semi[_semi.find(_key)+1]!="(" &&
        _semi.find("iterator")==_semi.length())
    {
        bool isPtr = false;
        if(_semi[_semi.find(_key)+1] == "**")
            isPtr = true;
        std::string variable = util.getVariableName(_key,_semi);
        dataObject* dObj = new dataObject(_key,isPtr,se[se.find(_key)],
            variable,_parser.getToker()->lines() + 1);
        dObj->StoreDeclFile(_parser._cur_file);
        _fun.storeFunData(dObj);
    }
}

bool NativeTypeCheck(typeParser* _parser, SemiExp& se, FunObj* _fun)
{
    utility* util = new utility();
    se = util->removeQualifiers(se);
    const int NUMKEYS = 8;
    static std::string KeyTable[NUMKEYS] = {"int", "float", "double",
        "char", "string",
        "short","long", "bool"};

    std::string _temp="";
    if(se.find("=")!=se.length())
    {
        for(int i = 0;i < se.find("="); i++)
            _temp.append(se[i] + " ");
    }
    else
        _temp = se.showSemiExp();
    Toker* pTok = new Toker(_temp,false);
    SemiExp _semi(pTok);
    if(pTok == 0)
    {
        delete pTok;
        return false;
    }
    while(_semi.getSemiExp())
    {
        for(int i=0; i<NUMKEYS; i++)
        {
            std::string _key = KeyTable[i];
            NativeTypeCheck_1(*fun, se, _key,_semi, *util, *parser);
        }
    }
    delete pTok;
    return false;
}
```



We see from the above restructured code that multiple arguments are passed to the new function. The number of arguments for the newly extracted function depends on the structure of source code, size of the feasible region and user defined constraints. We also notice that all parameters are passed by reference, irrespective of the parameters being native type or pointer type. The same rules discussed in the previous section apply when passing multiple arguments.

## **4.2 Restructuring Methods**

We have seen the process of restructuring functions in Section 4.1. In this section, we discuss restructuring methods of user defined types. As mentioned earlier, we use the convention of calling a member function of a user-defined type such as class or structure, as a method. Parsing methods and identifying feasible regions are same as that of a function. However, restructuring of methods require changes in the method declared and its files. Methods may or may not be declared and defined in the same file. Hence, restructuring methods require two changes – Making changes to the method declared file and making changes to the method defined file, if different. The newly extracted function is defined in the same file as that of the method.

Below is the source code of a class definition, which includes method declarations.

File: **FileManager.h**

```
class ManageFile
{
    private:
        Directory* dir;
        std::string _writing_dir;
        int line;
        int end_line;
        std::string decl;
        Toker* _fToker;
        bool isFunVerbose_;
        int _counter;
```

```

public:
    ManageFile()
    {
        dir = new Directory();
    }
    int& getFunCounter()
    {
        return _counter;
    }
    int& getEndLine()
    {
        return end_line;
    }
    std::string& getExtractedContents();
    void setFunVerbose(bool isFunVerbose);
    void registerFunToker(Toker* pToker);
    Toker* getToker()
    {
        return _fToker;
    }
    void setLine(int _line)
    {
        line = _line;
    }
    void setDecl(std::string _decl)
    {
        decl = _decl;
    }
    int getLine()
    {
        return line;
    }
    std::string& getDecl()
    {
        return decl;
    }
    bool checkDirectory(std::string _dir);
    void createFile(std::string contents, std::string _file);
    std::string& readFileContents(FunctionParser* _fParser, std::string
                                file);
};

```

Below is the source code of method defined file –

The method we are considering for this example is ‘readFileContents’.

### File: **FileManager.cpp**

```

std::string& ManageFile::readFileContents(FunctionParser* _fParser,
std::string
                                file)
{
    std::string _contents = "";
    Toker _toker(file);

```

```

setFunVerbose(true);
registerFunToker(&_fToker);
if(_fToker == 0)
    throw std::exception("no registered toker");
SemiExp se(_fToker);
se.makeCommentSemiExp();
try
{
    int line_count=0;
    int line_end = 0;
    int _index = ++getFunCounter();
    std::ostringstream os;
    os << _index;
    std::string _extracted_fun = "void " + _fParser->_fNameIndex + "_" +
                                os.str().c_str() + "()\n";
    std::fstream file_op(&file[0],ios::in);
    char str[2000];
    while(!file_op.eof())
    {
        file_op.getline(str,2000);
        std::string _str = str;
        line_count++;
        if(line_count == getLine())
        {
            _contents.append(str);
            _contents.append("\n");
            _contents.append(decl);
            _contents.append("\n");
        }
        else if(line_count == getEndLine())
        {
            _contents.append(getExtractedContents() + "_" + _counter++);
            _contents.append("\n");
        }
        else
        {
            _contents.append(str);
            _contents.append("\n");
        }
    }
    file_op.close();
    return _contents;
}
catch(std::exception& ex)
{
    std::cout<<"Exception reading feasible regions";
    exit(1);
}
}

```

Below we present the changes that are made to the declared and defined file after code restructuring.

## 4.2.1 Changes to Method Declared File

Please note the new file name will be appended with ‘\_restructured’.

File: **FileManager\_restructured.h**

```
class ManageFile
{
    private:
        Directory* dir;
        std::string _writing_dir;
        int line;
        std::string decl;
        Toker* _fToker;
        bool isFunVerbose_;
        int _counter;
        int end_line;
    public:
        ManageFile()
        {
            dir = new Directory();
        }
        int& getFunCounter()
        {
            return _counter;
        }
        int& getEndLine()
        {
            return end_line;
        }
        std::string& getExtractedContents();
        void setFunVerbose(bool isFunVerbose);
        void registerFunToker(Toker* pToker);
        Toker* getToker(){return _fToker;}
        void setLine(int _line) { line = _line; }
        void setDecl(std::string _decl) {_decl = _decl; }
        int getLine()
        {
            return line;
        }
        std::string& getDecl()
        {
            return decl;
        }
        bool checkDirectory(std::string _dir);
        void createFile(std::string contents, std::string _file);
        std::string& readFileContents(FunctionParser* _fParser, std::string
            file);
        void readFileContents_1(int& line_count, std::string& str,
            std::string& _contents);
};
```

The changes to the method declared file include adding the newly extracted method's prototype to the class definition. From the above code, we see that the prototype of new function 'readFileContents\_1' is added to *ManageFile*'s definition.

## 4.2.2 Changes to Method Defined File

Renaming of restructured file is also done for method defined files.

File: **FileManager\_restructured.cpp**

```
void ManageFile::readFileContents_1(int& line_count, std::string& str,
                                   std::string& _contents)
{
    line_count++;
    if(line_count == getLine())
    {
        _contents.append(str);
        _contents.append("\n");
        _contents.append(decl);
        _contents.append("\n");
    }
    else if(line_count == getEndLine())
    {
        _contents.append(getExtractedContents() + "_" + _counter++);
        _contents.append("\n");
    }
    else
    {
        _contents.append(str);
        _contents.append("\n");
    }
}

std::string& ManageFile::readFileContents(FunctionParser* _fParser,
                                         std::string file)
{
    std::string _contents = "";
    Token _toker(file);
    setFunVerbose(true);
    registerFunToker(&_toker);
    if(_fToker == 0)
        throw std::exception("no registered toker");
    SemiExp se(_fToker);
    se.makeCommentSemiExp();
    try
    {
        int line_count=0;
        int line_end = 0;
    }
}
```

```

int _index = ++getFunCounter();
std::ostringstream os;
os << _index;
std::string _extracted_fun = "void " + _fParser->_fNameIndex + "_" +
                             os.str().c_str() + "()\n";

std::fstream file_op(&file[0],ios::in);
char str[2000];
while(!file_op.eof())
{
    file_op.getline(str,2000);
    std::string _str = str;
    readFileContents_1(line_count, _str, _contents);
}
file_op.close();
return _contents;
}
catch(std::exception& ex)
{
    std::cout<<"Exception reading feasible regions";
    exit(1);
}
}

```

In the above restructured method, we see that same rules apply for passing of parameters and identifying feasible regions. One interesting thing to notice is, the variable ‘\_counter’ is a data member of the class and is not passed to the newly extracted function. This is done because both the data member and method belong to the same class.

### 4.2.3 Transforming local data into member data

For the above source code, we have shown the restructured code in the previous section. Suppose we consider the following segment of code, in a method, to be a feasible region:

**Please note:** The below section of code is an extraction from a method and complete function definition is omitted for brevity.

```

std::fstream file_op(&file[0],ios::in);
char str[2000];
while(!file_op.eof())
{
    file_op.getline(str,2000);
    std::string _str = str;
    line_count++;
    if(line_count == getLine())

```

```

    {
        _contents.append(str);
        _contents.append("\n");
        _contents.append(decl);
        _contents.append("\n");
    }
    else if(line_count == getEndLine())
    {
        _contents.append(getExtractedContents() + "_" + _counter++);
        _contents.append("\n");
    }
    else
    {
        _contents.append(str);
        _contents.append("\n");
    }
}

```

The variable 'file\_op' is declared in the feasible region, which would be extracted as a new function. However, it will be used in the calling function, after a call to the extracted function. If the feasible region is extracted, the restructured code won't compile as the declaration and definition would be two separate functions. In such a case, the variable can be added as a data member of the class. This idea is presented in Chapter 5 for future work.

### ***4.3 Restructuring functions in multiple passes***

We have seen in earlier sections, restructuring of functions and methods, with variants of number of parameters being passed to the extracted function. It would be interesting to know the affect of restructuring source code multiple times. By this, we mean, restructuring source code which is already restructured, in multiple passes. For this, let's consider a large source code method presented in [Appendix A.1](#)

By performing restructuring of the method shown above, in the first pass, we get the following restructured code.

```

void fileAnalyzer::testFun_1(std::string& _file, ExecAnalyzer& analExec,
                             TypeParser& _typeParser)
{
    std::cout << "\n Processing file " << _file <<std::endl;
    try

```

```

    {
        int flag = analExec.checkInput(_file);
        Tokener newTokener(_file);
        _typeParser.setTypeVerbose(true);
        _typeParser.registerTypeTokener(&newTokener);
        _typeParser.getUserDefinedTypes(root);
        _typeParser.getTypeNames();
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}

```

The changes made to the calling function, is presented in [Appendix A.1.1](#)

As we can see from the above code, a new function ‘testFun\_1’ is extracted from the method ‘testFun’. We also notice the parameters ‘\_file’, and pointer variables ‘\_typeParser’ and ‘analExec’ is passed by reference. The variable ‘root’, that is present in the extracted function is not passed as a parameter. This is because, ‘root’ is a data member and can be accessed from the new member function ‘testFun\_1’.

Performing code restructuring on the restructured code, shown above, we get the following results –

```

void fileAnalyzer::testFun_2(std::string& _file, TypeParser& _typeParser)
{
    try
    {
        std::cout<<"Pre-Processing File - "<< _file <<std::endl;
        Tokener newTokener(_file);
        _typeParser._cur_file = _file;
        _typeParser.setTypeVerbose(true);
        _typeParser.registerTypeTokener(&newTokener);
        _typeParser.getUserDefinedTypes(root);
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}

void fileAnalyzer::testFun_1(std::string& _file, ExecAnalyzer& analExec,
                             TypeParser& _typeParser)
{

```



```

std::cout << "\n Processing file " << _file <<std::endl;
try
{
    int flag = analExec.checkInput(_file);
    Toker newToker(_file);
    _typeParser.setTypeVerbose(true);
    _typeParser.registerTypeToker(&newToker);
    _typeParser.getUserDefinedTypes(root);
    _typeParser.getTypeNames();
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}

```

The changes made to the calling function is presented in [Appendix A.1.2](#)

We notice from the above results, that a new method ‘testFun\_2’ is extracted from the method ‘testFun’. By this we conclude that, code restructuring can be performed in multiple passes to make smaller components from a large method or function. By repeating this, we can further restructure to extract more methods from ‘testFun’ and reduce the size of ‘testFun’. In this process of restructuring, it is also possible that feasible regions may be identified in extracted functions, and new functions may be extracted from previously extracted functions.

Below is a table of results got by restructuring source code in multiple passes –

	Maximum number of parameters	Maximum number of lines	Number of lines in host method: <code>testFun</code>
Pass 1	3	20	105
Pass 2	3	20	97
Pass 3	3	20	92
Pass 4	3	20	79

**Figure 4.1 – Restructuring source code in multiple passes**

We notice from the above results that, by repeated restructuring source code, we can derive smaller sized functions. The size of the functions may keep decreasing in every pass, depending on the source code and the user entered constraints. It is also possible that the function that was extracted in the previous pass, can itself be analyzed as a feasible region and a new function may be extracted from the new function.

## **Chapter 5 Contributions, Conclusions, and Future Work**

In this chapter, we summarize the goals of Software Code Restructuring, review our research statement, our contributions to Software Code analysis and Restructuring, and discuss future work.

### ***5.1 Reviewing Research Statement***

As discussed in sections 1.1, working with large software source code is time consuming and prone to errors. We have seen the importance of code structure and how it can impact the overall complexity and design of software products. As discussed in section 1.2, code structure also plays an important role in determining the quality of software. It's time consuming and error prone to work with lengthy source code as we have seen from sections 1.3. Maintaining and validating source code for its functionality is also affected by the complexity and size of the source code.

Our goal in this study was to devise a method, and prototype, for semi-automatic restructuring that would reduce the size of large source code functions by extracting parts of the code into called functions. This makes it easier for users to work with lengthy source code. To achieve this, our study was concentrated on code structure and analyzing the relationships that exist in source code elements such as scopes, data accessibility and class structure. Our work begins by developing a source code parser which scans source code files, converting them to tokens and semi-expressions. We identify program elements that create scopes and data by analyzing each semi-expression.

We made use of a parse tree data structure, which was created on the heap, to hold values and attributes of the structure of the source code. We have seen the makeup of the

parse tree in Chapter 3, which includes various node types, data elements and an intermediate stack, which are represented as objects.

By analyzing the contents of a built parse tree, we can identify sections of source code which are candidates to be extracted as new functions, based on user entered constraints, discussed in section 4.1 and 4.2. These identified sections of source code, termed ‘Feasible regions’ are extracted and written as new functions, in a restructured file. The external behavior of source code is kept invariant by calling the newly extracted functions at suitable locations. The data that needs to be passed to newly extracted functions are determined and all data variables are passed by reference, which maintains in the calling code the same data values as before restructuring.

The restructured files are written to a new file, which gives the user a chance to compare and analyze the restructured files.

## **5.2 Contributions**

Our contributions in this thesis focus on understanding source code structure and develop techniques for determining sections of source code which can be extracted as new functions. Below is a discussion of our contributions to software code restructuring

- We devised a framework to work with source code elements by breaking down input file streams into a set of identifiable tokens and semi-expressions. We developed a type analysis parser that would identify host language’s source elements. Our parser was devised to work with native C/C++ languages, with object oriented constructs.
- We proposed a new approach to determine the source code structure by representing all the elements of source code in a traceable tree structure called the parse tree. The parse tree was composed of different types of nodes and data objects, with each data

object representing a data variable of the source code. We created an intermediate data structure called the ‘hierarchy stack’ to maintain the current source code position while parsing through the source code .Hierarchy stack was also used to analyze a section of source code and short list feasible regions. The novelty of this is the use of very simple grammar constructs that are nicely matched to the analysis task.

- We developed a mechanism for determining feasible regions. This was based on user entered constraints and implementing an optimization algorithm for determining feasible regions which extracts the largest number of lines that meet a constraint on the number of parameters passed. The algorithm has three possible methods of top-down or bottoms-up extraction and two look ups, of which, we have implemented top-down approach, as discussed in Chapter 3.
- The entire process of code restructuring is semi-automated. The user has control over the restructuring properties as discussed in section 2.6. After entering the values required to start the restructuring process, our implementation doesn’t require any user guidance, thus making it semi-automatic. The user doesn’t need to worry about selecting feasible regions or writing the extracted new functions into a new file, with all the required changes.

### **5.2.1 Accomplished Work**

In this section, we survey the work that is accomplished in our study and the results we have achieved.

In Chapter 4 we have provided and interpreted results obtained from code restructuring. We have demonstrated the methods of parsing and identifying source code elements from which semantic analysis can be performed in Chapter 3. The scope of our

study is targeted towards native languages such as C and C++. The parsing mechanism and analysis is entirely based on the syntactic analysis of code structure that exists in source code files. The analyzed information which is retrieved from the parse tree data is used, together with other parser information, to identify feasible regions and extract new functions. New files are written to the disk, keeping the source code as it is, to review results and as a back up mechanism.

We have also demonstrated the semi-automated nature of our implementation where code restructuring is performed automatically with no user intervention after the code restructuring process is started. The user has some control over code restructuring results by providing information of the source code file, the maximum number of parameters that can be passed to a newly extracted function and the maximum number of lines in a feasible region. The automation part of our implementation also includes creating result directory and files automatically, following the naming conventions discussed in earlier sections.

However, the process of extraction of new functions is based on programmatic grammar analysis and constraint checking. When the source code is visually examined, we may find areas that are, in some sense, better choices than the ones identified by our implementation. This is a limitation of our automation, as compared to human intelligence and reasoning. We identify various leads and interesting observations, by which our study can be expanded in the future to improve the restructuring process. Our next section discusses those areas that were identified as interesting leads for future development of source code analysis and restructuring.

### **5.3 Future Work**

During our study of code restructuring, many ideas, limitations, and other directions were discussed that may be topics for future research:

- **Relationship Analysis –**

Our study of Code restructuring can be extended to perform syntax-based relationship analysis of the source code. By relationship analysis, we mean performing type information analysis of object containment as in association, composition and using relationships that exist between user-defined types. Software code restructuring could be made more effective if relationship information can be used to factor common code from derived classes into a base, move processing centered on a composed type into the definition of that type, and repackage class definitions to bring dependent types into the same file, where size permits.

- **Extracting Derived types –**

User defined types such as classes, structures, and enumerations could be extracted in the same way as new functions are extracted. Recognizing a data structure and methods that make transformations on its contents may be the basis for automation of the definition of new classes from within existing code.

- **Semantic Cues –**

Our implementation of identifying feasible regions and extracting new functions are constrained to user entered values and implemented algorithm. If a section of source code satisfies these criteria, that section is extracted as a new function. Semantic cues that indicate certain types of processing may help make this process more rational. For example, recognizing that certain sections of code deal exclusively with input and

output, or threads, or handling storage, could be used more cleverly to extract meaningful functions and objects.

- **Language Specific Constructs –**

We have identified some language specific constructs that are not recognized by our parsing mechanism and are left for future implementations to address. Some of the C/C++ language constructs that could be included in our parsing mechanisms, but currently are not, are template classes and template functions, multi-level pointers, function pointers and nested classes. Addressing these was judged complex enough to be put aside temporarily while we built a functioning prototype.



## Appendix

In this section, we present some of the experimental data we have used and the results that are generated executing the prototype.

### A.1. Large Method

```
void fileAnalyzer::testFun(char* inFile)
{
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    fileInfo f;
    std::string _home = f.getPath();
    Analyzer* anal = new Analyzer();
    char* _argv[] = {&_amp;_home[0], "*.h", "*.cpp", "*.c"};
    std::vector<std::string> files = anal->getDirFiles(4, _argv);
    try
    {
        typeParser* _typeParser = new typeParser();
        ExecAnalyzer* analExec = new ExecAnalyzer();
        for(int i=0; i<files.size(); ++i)
        {
            std::string _file = files[i];
            std::cout << "\n Processing file " << _file <<std::endl;
            try
            {
                int flag = analExec->checkInput(_file);
                Toker newToker(_file);
                _typeParser->setTypeVerbose(true);
                _typeParser->registerTypeToker(&newToker);
                _typeParser->getUserDefinedTypes(root);
                _typeParser->getTypeNames();
            }
        }
    }
}
```

```

    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}
customPrepro* cust = new customPrepro();
std::vector<std::string>::iterator iter;
for(iter=cust->myIncludes.begin();iter!=cust->myIncludes.end();iter++)
{
    std::string _temp=*iter;
    if(_temp.find("\\")!=-1 || _temp.find("/")!=-1)
    {
        Toker newToker(_temp);
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
        _typeParser->getTypeNames();
    }
}
std::vector<std::string> files_2 = scanr.getCompleteFiles();
std::vector<std::string>::iterator iter2;
for(iter2 = files_2.begin();iter2!=files_2.end();iter2++)
{
    try
    {
        std::string _temp=*iter2;
        Toker newToker(_temp);
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
        _typeParser->getTypeNames();
    }
    catch(std::exception& ex)
    {
        std::cout<<ex.what()<<std::endl;
        exit(1);
    }
}

```

```

    }
}

for(int i=0; i<files.size(); ++i)
{
    std::string _file = files[i];
    try
    {
        std::cout<<"Pre-Processing File - "<< _file <<std::endl;
        Toker newToker(files[i]);
        _typeParser->_cur_file = files[i];
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}

for(int i=0; i<files_2.size(); ++i)
{
    std::string _file = files_2[i];
    try
    {
        std::cout<<"Pre-Processing File - "<< _file <<std::endl;
        Toker newToker(_file);
        _typeParser->_cur_file = _file;
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}

```

```

std::string _file = inFile;
std::cout << "\n Processing file " << _file <<std::endl;
try
{
    traceFunctions(_typeParser,&_file[0]);
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}

```

### ***A.1.1. Restructured Method in First Pass***

```

void fileAnalyzer::testFun(char* inFile)
{
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    fileInfo f;
    std::string _home = f.getPath();
    Analyzer* anal = new Analyzer();
    char* _argv[] = {&_home[0], "*.h", "*.cpp", "*.c"};
    std::vector<std::string> files = anal->getDirFiles(4,_argv);
    try
    {
        typeParser* _typeParser = new typeParser();
        ExecAnalyzer* analExec = new ExecAnalyzer();
        for(int i=0; i<files.size(); ++i)
        {

```

```

        std::string _file = files[i];
        testFun_1(_file, *analExec, *_typeParser);
    }
    customPrepro* cust = new customPrepro();
    std::vector<std::string>::iterator iter;
    for(iter=cust->myIncludes.begin();iter!=cust->myIncludes.end();iter++)
    {
        std::string _temp=*iter;
        if(_temp.find("\\")!=-1 || _temp.find("/")!=-1)
        {
            Toker newToker(_temp);
            _typeParser->setTypeVerbose(true);
            _typeParser->registerTypeToker(&newToker);
            _typeParser->getUserDefinedTypes(root);
            _typeParser->getTypeNames();
        }
    }
    std::vector<std::string> files_2 = scanr.getCompleteFiles();
    std::vector<std::string>::iterator iter2;
    for(iter2 = files_2.begin();iter2!=files_2.end();iter2++)
    {
        try
        {
            std::string _temp=*iter2;
            Toker newToker(_temp);
            _typeParser->setTypeVerbose(true);
            _typeParser->registerTypeToker(&newToker);
            _typeParser->getUserDefinedTypes(root);
            _typeParser->getTypeNames();
        }
        catch(std::exception& ex)
        {
            std::cout<<ex.what()<<std::endl;
            exit(1);
        }
    }
    for(int i=0; i<files.size(); ++i)
    {

```

```

std::string _file = files[i];
try
{
    std::cout<<"Pre-Processing File - "<< _file <<std::endl;
    Toker newToker(files[i]);
    _typeParser->_cur_file = files[i];
    _typeParser->setTypeVerbose(true);
    _typeParser->registerTypeToker(&newToker);
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}
for(int i=0; i<files_2.size(); ++i)
{
    std::string _file = files_2[i];
    try
    {
        std::cout<<"Pre-Processing File - "<< _file <<std::endl;
        Toker newToker(files_2[i]);
        _typeParser->_cur_file = files_2[i];
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}
std::string _file = inFile;
std::cout << "\n Processing file " << _file <<std::endl;
try
{
    traceFunctions(_typeParser,&_file[0]);
}

```

```

    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}

```

## ***A.1.2 Restructured Method in Second Pass***

```

void fileAnalyzer::testFun(char* inFile)
{
    Directory dir;
    Scanner scanr;
    scanr.doRecursiveScan(inFile);
    dir.RestoreFirstDirectory();
    fileInfo f;
    std::string _home = f.getPath();
    Analyzer* anal = new Analyzer();
    char* _argv[] = {&_amp;_home[0], "*.h", "*.cpp", "*.c"};
    std::vector<std::string> files = anal->getDirFiles(4, _argv);
    try
    {
        typeParser* _typeParser = new typeParser();
        ExecAnalyzer* analExec = new ExecAnalyzer();
        for(int i=0; i<files.size(); ++i)
        {
            std::string _file = files[i];
            testFun_1(_file, *analExec, *_typeParser);
        }
        customPrepro* cust = new customPrepro();
    }
}

```

```

std::vector<std::string>::iterator iter;
for(iter=cust->myIncludes.begin();iter!=cust->myIncludes.end();iter++)
{
    std::string _temp=*iter;
    if(_temp.find("\\")!=-1 || _temp.find("/")!=-1)
    {
        Toker newToker(_temp);
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
        _typeParser->getTypeNames();
    }
}
std::vector<std::string> files_2 = scanr.getCompleteFiles();
std::vector<std::string>::iterator iter2;
for(iter2 = files_2.begin();iter2!=files_2.end();iter2++)
{
    try
    {
        std::string _temp=*iter2;
        Toker newToker(_temp);
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
        _typeParser->getUserDefinedTypes(root);
        _typeParser->getTypeNames();
    }
    catch(std::exception& ex)
    {
        std::cout<<ex.what()<<std::endl;
        exit(1);
    }
}
for(int i=0; i<files.size(); ++i)
{
    std::string _file = files[i];
    try
    {
        std::cout<<"Pre-Processing File - "<< _file <<std::endl;
    }
}

```



```

        Toker newToker(files[i]);
        _typeParser->_cur_file = files[i];
        _typeParser->setTypeVerbose(true);
        _typeParser->registerTypeToker(&newToker);
    }
    catch(std::exception& ex)
    {
        std::cout<<std::endl<<ex.what()<<std::endl;
        exit(1);
    }
}
for(int i=0; i<files_2.size(); ++i)
{
    std::string _file = files_2[i];
    testFun_2(_file, *typeParser);
}
std::string _file = inFile;
std::cout << "\n Processing file " << _file <<std::endl;
try
{
    traceFunctions(_typeParser,&_file[0]);
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}
catch(std::exception& ex)
{
    std::cout<<std::endl<<ex.what()<<std::endl;
    exit(1);
}
}
}

```

## Bibliography

1. Structured Models for Large Software Systems by Murat Kahraman Gungor  
Ph. D Dissertation, July 2006
2. Syracuse Medical Imaging Research Group  
<http://smirg.syr.edu>
3. Upstate Medical University  
<http://upstate.edu>
4. Compiler-Directed Code Restructuring for Improving Performance of MPSoCs  
By Guilin Chen, Mahmut Kandemir  
<http://doi.ieeecomputersociety.org/10.1109/TPDS.2007.70760>
5. Compiler-directed code restructuring for reducing data TLB energy  
By M. Kandemir, I. Kadayif, G. Chen  
<http://doi.acm.org/10.1145/1016720.1016747>
6. Compiler-directed code restructuring for Improving I/O Performance  
By Mahmut Kandemir, Seung Woo Son  
[http://usenix.org/event/fast08/tech/full\\_papers/kandemir/kandemir\\_html/index.html](http://usenix.org/event/fast08/tech/full_papers/kandemir/kandemir_html/index.html)
7. Restructuring in FORTRAN  
<http://citeseer.comp.nus.edu.sg/2512.html>
8. Restructuring in Embedded DSPs  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.1402>
9. Code Restructuring in Virtual Hardware Systems  
<http://www.patentstorm.us/patents/7356456/fulltext.html>
10. Code Restructuring in Source Code Transformation

<http://www.patentstorm.us/patents/6934940.html>

11. Code Restructuring in Emulator Soft wares

<http://www.patentstorm.us/patents/6718485/claims.html>

12. Microsoft Visual Studio IDE

<http://msdn.microsoft.com/en-us/vstudio/default.aspx>

13. Eclipse IDE

<http://www.eclipse.org/>