International Journal of Computer Architecture and Mobility
(ISSN 2319-9229) Volume 1-Issue 10, August 2013

Efficient Grammar Recovery and Refactoring Through Parsing Technique

Sachin Singh[1] *SRM University* singhsachin2407@gmail.com, A. Kulothungan [2] *SRM University* kulosoft@gmail.com

***Abstract*— Many software development tools that assist with tasks such as testing and maintenance are specific to a particular development language. Parser is required to generate a grammar but a grammar is not always available for a language.**

By the grammar which is recovered, we can generate test case application. Testing is also performed. The grammars are engineered from scratch, reverse engineered from the tools that contain them implicitly, extracted from available sources. The list of possible artefacts bearing grammar knowledge includes language processor source code, language documentation, and codebase. The extraction process often comprises more than simple mapping activities.

The main characteristic of the approach is that the grammars are not constructed from scratch but they are rather recovered by extracting them from language references, compilers, and other artefacts. We provide a structured process to recover grammars including the adaptation of raw extracted grammars and the derivation of parsers. The process is applicable to possibly all existing languages for which business critical applications exist.

We implement the refactoring algorithm to remove the iterations from the grammar. On this iterative grammar we perform grammar refactoring. By refactoring, the internal structure of grammar is modified. Therefore, we get an unambiguous grammar which is more efficient for use. As a result a recursive grammar is generated. This recursive grammar can be used effectively for test case generation.

Keywords*— Grammar, grammar recovery, grammar refactoring, parse tree, recursive grammar.

I. INTRODUCTION

A language is formally defined as a countable set of finite sequences of symbols from a given alphabet. Conversely, anything that can be expressed or perceived as a set of symbol sequences can be considered a language.

A language is commonly defined by a grammar. Most of language processing tools and methodologies rely on the parsing process, which analyses the source code according to the rules of the grammar. This places the grammar at the foundation of almost any language processing infrastructure. The software which input can be described by a grammar is called grammar-based software or grammarware.

A grammar does not exist for every language. When a grammar for a language is not available, acquiring a correct and complete grammar for that language is the most difficult, costly, and time-consuming phase of constructing a tool for use with the language. To address the problem of grammar acquisition, researchers have directed significant attention to the problem of grammar recovery, which comprises the procedures involved in the derivation of a grammar for a language from the available resources.

Recovering a grammar from only code samples is called grammar inference. Gold’s theorem states that it is impossible to infer the grammar of an arbitrary unknown language from only positive (syntactically correct) code samples. Recovering a grammar from a language reference manual necessarily involves much manual effort, and reference manuals often contain errors. La¨mmel and Verhoef use a language reference manual (including code samples from that reference manual) to semi-automatically recover a grammar.

Sellink and Verhoef automatically recover a grammar from BNF found in the source code of a compiler. Finally, recovering a grammar from a hard-coded parser requires manual inspection of source code. Duffy and Malloy describe a related, but distinct, approach to recovering a grammar from a hard-coded parser. They instrument the source code of a hard-coded parser to generate parse trees; using these parse trees, they automatically recover a grammar.

A significant issue associated with the recovery of a grammar from a hard-coded parser is that parser generation algorithms place restrictions on the form of a grammar. These restrictions make the grammar difficult to comprehend, meaning that the grammar might not be useful in its recovered form. For example, left recursion often is used to introduce repetition in a grammar. However, a recursive descent parser cannot recognize a grammar in which a nonterminal is expressed using left recursion. In some cases, such a nonterminals can be rewritten with right recursion; however, in other cases, such a nonterminals cannot be easily rewritten with right recursion.

We define a recursive grammar as one that contains a nonterminal expressed using left or right recursion and an iterative grammar as one that contains a nonterminal expressed using iteration. If a grammar contains both kinds of nonterminal, we refer to it as iterative. Unlike a recursive grammar, an iterative grammar is illegible to the average software developer further; an iterative grammar is more verbose than the corresponding recursive grammar. Finally, multiple implementations of an iterative grammar can result in distinct versions of the grammar because iteration must be bounded and the bound is an implementation-defined value.

The grammars are engineered from scratch, reverse engineered from the tools that contain them implicitly, extracted from available sources. The list of possible artefacts bearing grammar knowledge includes language processor source code, language documentation and codebase. The extraction process often comprises more than simple mapping activities. In the case of one primary grammar and a set of secondary grammars derived from it, the former is usually called a base-line grammar.

The problem with language dialects is that there has been little research, to date, addressing the problem of reverse engineering a grammar or language specification for a language dialect from existing language artefacts. L¨ammel and Verhoef have developed a technique that uses a language reference manual and test cases to recover a grammar for a language or dialect. However, their technique requires user intervention along most of the stages of recovery and some of the recovery process is manual. Bouwers et al. present a methodology for recovering precedence rules from grammars. However, there is no existing technique to enable a developer to automatically reverse engineer a grammar for an existing language or language dialect.

La¨mmel and Verhoef in [3] describe a sequence of cases that cover virtually all of the approaches for recovering a grammar. They refer to the grammar recovery process as grammar stealing because the language already exists and the goal of grammar recovery is to leverage existing language artifacts to "steal" the grammar. The sequence of cases that they enumerate is distinguished by the language artefacts that are available for the recovery process. The first case in the sequence distinguishes those artifacts that include compiler sources from those that include only a language reference manual.

Sellink and Verhoef present a completely automated approach to grammar recovery using the source code of a compiler. Their approach leverages a parser for which the grammar is encoded in a dialect of BNF. They translated the extracted production rules into the modular syntax definition formalism (SDF) and used the recovered grammar in the development of a Software Renovation Factory.

There are several advantages to the approach of Sellink and Verhoef. First, their approach is applicable when the language reference manual is unusable. Second, they require neither code samples nor parse trees in their grammar recovery. A disadvantage of their approach is that they do require that the BNF of the grammar be included in the compiler source code.

Duffy and Malloy describe an approach to grammar recovery for a dialect of the C++ language; the current paper is an extension of this previous research. An advantage of their approach is that the grammar need not be hard-coded in the parser, and given parse trees for the language, their approach is fully automated. However, their approach does require an existing parser, and to generate the parse trees, they may have to modify the parser by inserting probes into the semantic actions of the parser. Furthermore, their approach to introduction of left recursion in place of iteration requires a priori knowledge of the grammar.

Grammar Recovery from a Language Reference Manual and Code Samples deals with grammar recovery from a language reference manual, and possibly, code samples. La¨mmel and Verhoef observe that the manual can be either a compiler vendor manual or an official language standard. Moreover, the language is explicated either through code samples, through general rules, or through a combination of both samples and rules.

They present a semi-automated approach to grammar recovery that uses a language manual and a test suite. They use the manual to construct syntax diagrams for the language, correct the diagrams, write transformations to correct connectivity errors, and then, use the test cases to further correct the generated grammar.

There are several advantages to their approach. The first advantage is that the grammar can be recovered quickly; for example, recovery of a COBOL grammar required only two weeks effort. A second advantage of their approach is that their grammar recovery technique is not connected to a specific parser implementation. A disadvantage of their approach is that many phases of the recovery process are manual.

Using both positive and negative code samples, they systematically search for valid parse trees, stopping when either the parse tree produces a context-free grammar that accepts all positive code samples and rejects all negative code samples, or when the search is exhausted.

In Semi-automated grammar recovery [2] an approach to the construction of grammars for *existing* languages was proposed. This approach was simple. For, the grammars are already written, they only had to extract them and transform them into the correct form. More precisely, the following steps were taken:

- raw grammar extraction from a language reference, a compiler or another artefact;
- resolution of static errors such as unconnected nonterminals, also called sort names, if the grammar is extracted from a non-executable source;
- extraction or definition of lexical syntax;
- test-driven correction and completion of the raw grammar if necessary;
- beautification;
- modularization;
- disambiguation if necessary;
- generation of a browsable version of the grammar if needed;
- Adaptation of the grammar for the intended purpose (e.g., renovation).

An object-oriented focused refactoring tool which when given a set of objects is able to produce an equivalent set, without duplication of methods or certain expressions by replacing inheritance hierarchies and factoring out expressions [7]. The inherent benefit of fully auto-mated refactoring is that it reduces the amount of user interaction required for larger system programs and may result in a more comprehensive transformative process.

The Grammar Recovery Kit illustrates options for automation and corresponding tool support in the context of developing quality language references that readily cater for the derivation of parsers [4].

GRK provides the proof-of-concept for two notions: (i) semi-automatic grammar recovery; (ii) language-reference re-engineering. GRK's support for semi-automatic grammar recovery means that GRK can be used to obtain a relatively correct and complete as well as implementable grammar from a language reference. GRK's support for language-reference re-engineering means that GRK can be used to update the original language reference such that it reflects the completed and corrected grammar knowledge.

In this project we present the design and implementation of a technique for reverse engineering, or recovering, a grammar from existing language artefacts. Throughout this project we use the term grammar recovery to refer to the extraction, assessment and testing of a grammar from existing language artefacts.

Grammarware comprises grammars and all grammar-dependent software. The term grammar is meant here in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, and XML schemas as well as some forms of tree and graph grammars. The term grammar-dependent software refers to all software that involves grammar knowledge in an essential manner. Archetypal examples of grammar-dependent software are parsers, program converters, and XML document processors.

The term grammar is used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, and XML schemas as well as some forms of tree and graph grammars. Grammars are used for numerous purposes, for example, for the definition of concrete or abstract programming language syntax, and for the definition of exchange formats in component-based software applications.

An important aspect of any language is its grammar. A grammar is the formal specification of the syntactic structure of a language. Such specifications are indispensable inputs to parser generators, like Lex and Yacc, or other generic tools, such as programming environment generators. Grammars are omnipresent in software engineering. Not solely in the language technology field, but also in other areas.

Once we obtain the parse trees, grammar recovery from those parse trees is automated. Because our technique for grammar recovery is based on parse trees, which are frequently utilized for testing and debugging of language-dependent software, we also believe that the technique can be applied successfully elsewhere. In addition to grammar recovery, we address refactoring of a recovered grammar. Our work on refactoring a recovered grammar is motivated by problems that result from recovering a grammar from a compiler source, such as a hard-coded parser. In particular, we present a metrics-guided approach to refactoring an iterative grammar to obtain a recursive grammar. In this approach, we leverage the grammar metrics presented by Power and Malloy. While our approach does require human input to identify candidate nonterminals for refactoring, computation of the metrics and refactoring of the identified candidate nonterminals are both fully automated.

II. PROPOSED METHODOLOGY

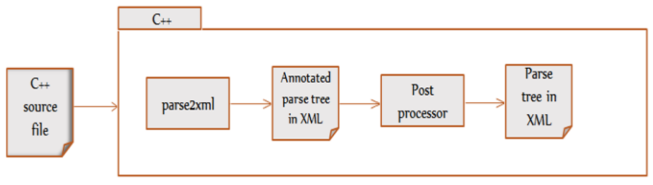
An overview of our system, which consists of two major subsystems: the parse tree recovery subsystem and the grammar recovery and refactoring subsystem. The parse tree recovery subsystem is shown in Fig. 1.1(a), and the grammar recovery and refactoring subsystem, grecovery, is shown in Fig. 1.1(b). It takes as input one or more C++ source files and produces as output the corresponding parse tree(s) encoded in XML.

To create parse2xml, we instrumented the source tree. The source code in cp/parser.c implements a hand-written, backtracking recursive descent parser. The postprocessor, in the middle of fig. 1.1, is a 233 line C++ program that converts an annotated parse tree to the actual parse tree. We implemented postprocessor to perform this conversion in two steps.

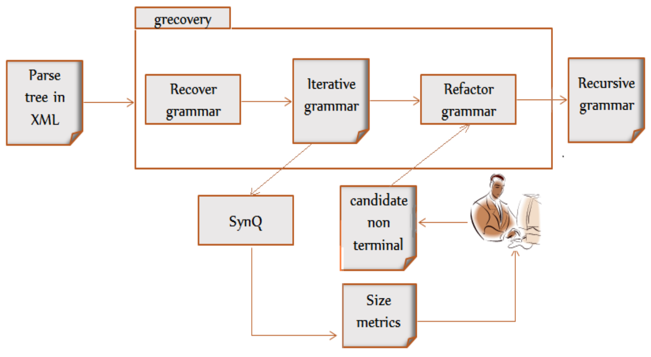
In the first step, we backpatch delayed parse subtrees. Parse2xml emits member function bodies and their default parameter lists after the class that contains these constructs, so we must backpatch the member function bodies and default parameter lists into the appropriate class to obtain a structurally correct parse tree. The second step performed by postprocessor is to commit or rollback the subtrees that result from tentative parsing.

Each annotated parse tree indicates which tentatively parsed subtrees were accepted (and thus, should be committed) as well as which tentatively parsed subtrees were rejected (and thus, should be rolled back, or eliminated). In this second step, we remove these annotations and possibly the annotated subtrees. The output of postprocessor is a structurally correct, XML-encoded parse tree.

We provide the XML-encoded parse trees produced and given as input to the grecovery system. In the Recover Grammar class, we implement the grammar recovery algorithm. The output of this class is an Iterative Grammar, which we provide both to SynQ and the Refactor Grammar class. SynQ is a metric computation system for grammars we use SynQ to compute the size metrics exploited by our methodology. The human shown in the lower right must review the size metrics computed by SynQ to identify Candidate Nonterminals. Once the candidates are identified, they are provided, along with the Iterative Grammar, to the Refactor Grammar class. This class implements the grammar refactoring algorithm and produces as output a Recursive Grammar.



(a)



(b)

Fig. 1: System Overview: fig. 1 (a) is the steps for Parse Tree Recovery and (b) tells the steps for Grammar Recovery and Refactoring.

A. Methodology for grammar recovery and Refactoring:

We describe a technique for automatic grammar recovery from parse trees and a metrics-guided approach to semi-automatic grammar refactoring. Our approach to grammar refactoring comprises three steps:

i. Computation of grammar metrics,
ii. Analysis of the metrics to identify candidate nonterminals to be transformed by replacing iteration with left recursion, and
iii. Transformation of the candidate nonterminals.

The first and third steps are fully automated; the second step is manual.

A parse tree captures the derivation of a sentence from a language, that is, a parse tree encodes the productions of a grammar that are exercised by the sentence in the language from which the parse tree is derived. Thus, a parse tree encodes an instance of the grammar, a partial grammar, for the language, and we can recover a partial grammar for a language from a parse tree. By taking the union of two partial grammars, that is, the union of the productions in the two partial grammars, we can recover a grammar that captures the productions encoded in each of the corresponding parse trees.

Depending on the parsing technology used by the parser that generates the parse trees, we might recover an iterative grammar rather than a recursive grammar. Because all possible production right-hand sides are expressed explicitly for a non-terminal written using iteration, a recovered iterative grammar will generate only a subset of the intended language unless an exhaustive test suite is used in conjunction with the grammar recovery technique.

B. Grammar Recovery from Parse Trees:

Our methodology for automatic grammar recovery minimally requires as input a single parse tree but can accept multiple parse trees with no modification. For simplicity, in this section, we describe the recovery of a (partial) grammar from a single parse tree.

The Grammar recovery algorithm is as follows

```
1.   grammar = { }
2.   recover_production(node)
3.           production = [node]
4.           foreach n in node.children
5.                   production.append(n)
6.           return production
7.
8.   recover_grammar(root)
9.           nodes = {root}
10.          foreach n in nodes
11.                  If n is interior node
12.
             grammar.add(recover_production(n))
13.                          nodes.add(n.children)
14.                          nodes.remove(n)
```

Line 1 of the algorithm lists the declaration for the global set grammar, which holds the recovered grammar. The grammar we recover using our algorithm is represented as a set of lists, where each list represents a production.

In line 9 of the algorithm, a set node is initialized to include only the root node. The loop that begins in line 10 adds the children of the current node n to nodes each time an interior node of the parse tree is encountered. Further, as an interior node corresponds to the left-hand side of a production, each such node is passed to the recover_production subroutine in line 12 of algorithm. The result of the subroutine call, a production in list form, is added to grammar. Upon termination of the recover_grammar subroutine, the grammar encoded by the parse tree is stored in grammar.

III. Result

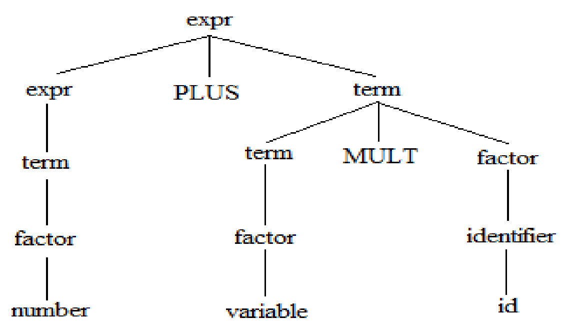
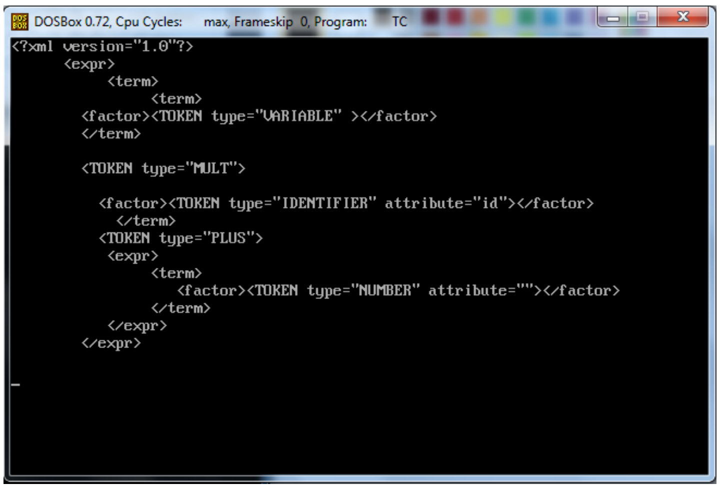


Fig. 2: Parse Tree

The above figure shows the parse tree for which the grammar is to be recovering. This parse is entered into the program as input.



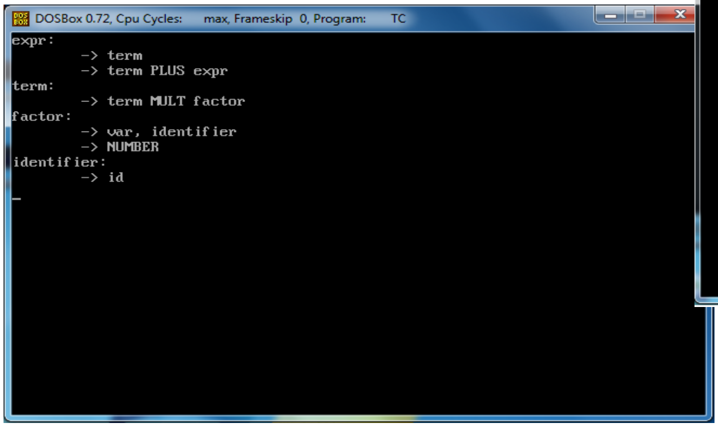
```
<?xml version="1.0"?>
      <expr>
           <term>
                <term>
        <factor><TOKEN type="VARIABLE" ></factor>
        </term>

        <TOKEN type="MULT">

          <factor><TOKEN type="IDENTIFIER" attribute="id"></factor>
            </term>
          <TOKEN type="PLUS">
           <expr>
                <term>
                   <factor><TOKEN type="NUMBER" attribute=""></factor>
                </term>
           </expr>
        </expr>
```

Fig. 3: Generated XML

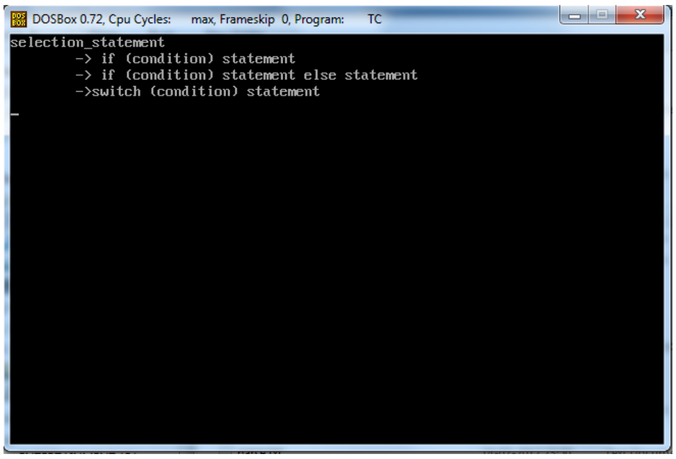
The fig. 3 shows the generated xml as an output. The parse tree (in fig. 2) is inputted to the program and we get xml as output from that program.



```
expr:
        -> term
        -> term PLUS expr
term:
        -> term MULT factor
factor:
        -> var, identifier
        -> NUMBER
identifier:
        -> id
_
```

Fig. 4: Grammar Recovery

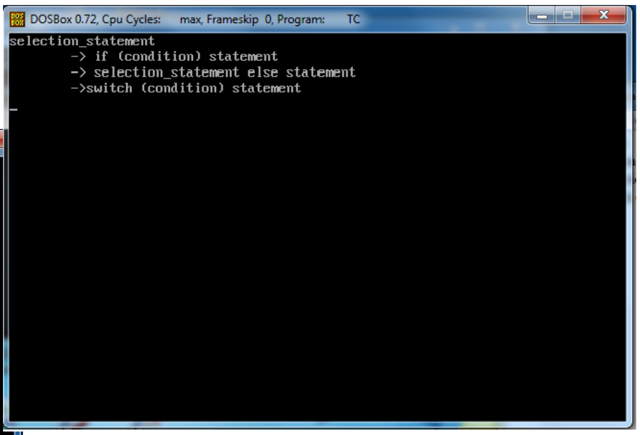
The fig. 4 shows the grammar that is recovered for the parse tree (in fig. 2). XML is entered to the program as input and grammar is generated as output after processing.



```
selection_statement
        -> if (condition) statement
        -> if (condition) statement else statement
        ->switch (condition) statement
_
```

Fig. 5: Iterative Grammar

The fig. 5 shows the iterative grammar before refactoring. An iterative grammar is one that which contains a nonterminal expressed using iteration or nonterminal expressed using left or right recursion. If a grammar contains both kinds of nonterminal, we refer to it as iterative. An iterative grammar is illegible to the average software developer; further, an iterative grammar is more verbose than the corresponding recursive grammar. Finally, multiple implementations of an iterative grammar can result in distinct versions of the grammar because iteration must be bounded and the bound is an implementation-defined value.



```
selection_statement
        -> if (condition) statement
        -> selection_statement else statement
        ->switch (condition) statement
_
```

Fig. 6: Recursive Grammar

The fig. 6 shows the recursive grammar after refactoring.

As iterative grammar is not useful for any work therefore we apply refactoring on it and generate a recursive grammar. We define a recursive grammar as one that contains a nonterminal expressed using left or right recursion.

IV. CONCLUSION

The methodology is comprised of manual instrumentation of the parser, a technique for automatic grammar recovery from parse trees, and a semi-automatic metrics-guided approach to refactoring an iterative grammar to obtain a recursive grammar. We presented algorithms for recovering a grammar from a parse tree and for rewriting nonterminals expressed using iteration with left recursion.

We also investigated the use of grammar size metrics for identifying candidate nonterminals for refactoring and found that, by refactoring the identified nonterminals.

In particular, no previously published research describes a methodology for grammar recovery from a hard-coded parser. Moreover, there is no published research that describes an approach for refactoring an iterative grammar to obtain a recursive grammar.

REFERENCES

[1] A. Sellink and C. Verhoef, "*Generation of Software Renovation Factories from Compilers*" Proc. 15th IEEE Int'l Conf. Software Maintenance, pp. 245-255, 1999.

[2] R. La¨mmel and C. Verhoef, "*Semi-Automatic Grammar Recovery*," Software: Practice and Experience, vol. 31, no. 15, pp. 1395-1438, Oct. 2001.

[3] R. Lammel and C. Verhoef, "*Cracking the 500-Language Problem*," IEEE Software, vol. 18, no. 6, pp. 78-88, Nov./Dec. 2001.

[4] Ralf L¨ammel "*The Amsterdam toolkit for language archaeology*", 2004.

[5] P. Klint, R. La¨mmel, and C. Verhoef, "*Toward an Engineering Discipline for Grammarware*," ACM Trans. Software Eng. And Methodology, vol. 14, no. 3, pp. 331-380, 2005.

[6] E.B. Duffy and B.A. Malloy, "*An Automated Approach to Grammar Recovery for a Dialect of the C++ Language*," Proc. 14th Working Conf. Reverse Eng., pp. 11-20, 2007.

[7] E. Bouwers, M. Bravenboer, and E. Visser, "*Grammar Engineering Support for Precedence Rule Recovery and Compatibility*," Proc. Seventh Workshop Language Descriptions, Tools, and Languages, pp. 82-96, Mar. 2007.

[8] Jingfeng Peng, "*Semi Automated Refactoring Applied to the C Programming Language*," 2008.

Sachin Singh received B.Tech degree in Computer Science & Engineering from Uttar Pradesh Technical University in 2010 and M.Tech degree in Computer Science & Engineering from SRM University, NCR Campus, in 2012.

A.Kulothungan received M.E degree in Computer Science & Engineering. He is currently working as an Assistant Professor in the Department of Computer Science & Engineering at SRM University, NCR Campus, Modinagar.