SUPPORTING SOFTWARE EXPLORATION WITH A SYNTACTIC AWARE SOURCE CODE QUERY LANGUAGE

A dissertation submitted to Kent State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

by

Brian Bartman

July 20, 2017
Dissertation written by

Brian Bartman

M.S., Kent State University, USA, 2012

B.S., Kent State University, USA 2010

__________________________  _________________________
Dr. Jonathan I. Maletic       Chair, Doctoral Dissertation Committee

__________________________  _________________________
Dr. Michael L. Collard        Members, Doctoral Dissertation Committee

__________________________
Dr. Ruoming Jin

__________________________
Dr. Gokarna Sharma

__________________________
Dr. Xinyue Ye

Accepted by

__________________________  _________________________
Dr. Javed I. Khan           Chair, Department of Computer Science

__________________________
Dr. James L. Blank           Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. VI

LIST OF TABLES ....................................................................................................................... XVI

DEDICATION ............................................................................................................................... XVII

ACKNOWLEDGEMENTS ............................................................................................................ XVIII

CHAPTER 1 INTRODUCTION ..................................................................................................... 1

1.1 Research Contributions ..................................................................................................... 3

1.2 Organization of Dissertation ............................................................................................ 4

CHAPTER 2 BACKGROUND AND RELATED WORK ............................................................... 5

2.1 Other Source Code Query Languages ............................................................................... 5

2.2 Discussion ......................................................................................................................... 10

CHAPTER 3 PRELIMINARY WORK .......................................................................................... 12

3.1 srcML .............................................................................................................................. 12

3.2 XPath ............................................................................................................................. 13

CHAPTER 4 SRCQL: SOURCE CODE QUERY LANGUAGE ..................................................... 18

4.1 srcPat .............................................................................................................................. 19

4.2 srcQL Language .............................................................................................................. 29

   4.2.1 Implicit srcPat ........................................................................................................ 30

   4.2.2 FIND Context .......................................................................................................... 30

   4.2.3 srcQL Operators .................................................................................................... 32

   4.2.4 CONTAINS ............................................................................................................ 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Language to Language Comparison</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>srcQL</td>
<td>76</td>
</tr>
<tr>
<td>6.4</td>
<td>RASCAL</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Coccinelle</td>
<td>84</td>
</tr>
<tr>
<td>6.6</td>
<td>JQuery</td>
<td>89</td>
</tr>
<tr>
<td>6.7</td>
<td>PMD</td>
<td>90</td>
</tr>
<tr>
<td>6.8</td>
<td>SCRUPLE</td>
<td>93</td>
</tr>
<tr>
<td>6.9</td>
<td>CodeQuest</td>
<td>98</td>
</tr>
<tr>
<td>6.10</td>
<td>Aria</td>
<td>99</td>
</tr>
<tr>
<td>6.11</td>
<td>srcML and XPath</td>
<td>101</td>
</tr>
<tr>
<td>6.12</td>
<td>Findings and Discussion</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>107</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>108</td>
</tr>
<tr>
<td>A</td>
<td>APPENDIX A SRCQL LANGUAGE GRAMMAR</td>
<td>111</td>
</tr>
<tr>
<td>B</td>
<td>APPENDIX B A SRCQL LANGUAGE SPECIFICATION</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>175</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 3.1 A simple rotation example where three values are swapped between one another. ................................................................. 13

Figure 3.2 The result of running srcML on source code from the rotation example in Figure 3.1 ................................................................. 13

Figure 3.3 The XPath axes relationships between the current self node A all other nodes relative to a given axis. ................................................................. 14

Figure 3.4 Example of selecting all child elements with the name E. ......................... 15

Figure 3.5 Selecting the second child element with the name E. ................................. 15

Figure 3.6 Selecting all child elements with the name E and a child element with the name D......................................................................................... 15

Figure 3.7 Selecting a child element C with parent element B....................................... 15

Figure 3.8 Selecting all elements with the name C which are a descendant of the current element......................................................................................... 16

Figure 3.9 Selecting all preceding elements within the document with the name Z. ...... 16

Figure 4.1 srcPat matching a specific call to new int. .................................................. 20

Figure 4.2 match all calls to operator new........................................................................ 20

Figure 4.3 Unification of syntactic variables $T to be the same type............................. 20

Figure 4.4 Searching for all function declarations that have a void return type............ 21

Figure 4.5 The result of running new int; through srcML ........................................... 21

Figure 4.6 Logic used to determine the search context for srcPat.................................. 23
Figure 4.7 Control flow generated for matching a srcPat pattern................................. 23
Figure 4.8 srcPat query generation algorithm.......................................................... 28
Figure 4.9 srcQL basic query structure................................................................. 29
Figure 4.10 Implicit srcPat, srcQL query. This provides a means for easily searching for simple patterns rather than having to write FIND before all queries. ....................... 30
Figure 4.11 The transformed query from Figure 4.10 ............................................ 30
Figure 4.12 Implicit FIND XPath........................................................................ 31
Figure 4.13 Implicit FIND source......................................................................... 31
Figure 4.14 FIND guess context for contains......................................................... 31
Figure 4.15 FIND the XPath A * B. ................................................................... 31
Figure 4.16 FIND the srcPat pattern A* B of a pointer declaration ....................... 31
Figure 4.17 Locate the name of all types used with operator new. ....................... 32
Figure 4.18 Find all functions that contain a certain usage of the new operator. ....... 32
Figure 4.19 Find all functions which allocate a variable using new and delete the same variable. .................................................................................. 33
Figure 4.20 Find all of the declaration statements that contain an XML
<operator>new</operator> in srcML ........................................................................ 33
Figure 4.21 Searching for the same thing twice with the contains has no effect on cardinality. ............................................................................................... 34
Figure 4.22 Functionally equivalent to the query in Figure 4.21............................. 34
Figure 4.23 Find all functions that open and close a C IO file stream. ................... 35
Figure 4.24 Find all functions that contain the opening of a file stream, printing to that stream, and closing the same stream. ................................................................. 35
Figure 4.25 Find all function definitions within a class..................................... 35
Figure 4.26 Example source containing a C-style file access pattern.................. 36
Figure 4.27 Find functions that contain a basic source code usage pattern for C file I/O. 36
Figure 4.28 Find all classes whose name matches the given regular expression. .... 37
Figure 4.29 Find all uses of operator new where the type is not an array type. .... 37
Figure 4.30 Find all classes that inherit from a typedef.................................. 38
Figure 4.31 Find the name of all variables matched as part of the nested query .... 39
Figure 4.32 Find all instances of new $T where the $T type is a class. ............. 39
Figure 4.33 Use pattern for binary set operators of srcQL................................ 40
Figure 4.34 Unioning the names of all structs and classes within a system.......... 40
Figure 4.35 Find all class definitions and sort them by name........................... 40
Figure 4.36 Find all classes and group them by name and order those groups by the number of classes per group from largest to smallest. ................................. 41
Figure 4.37 Find all function definitions and group them into overloads............... 41
Figure 4.38 Find all function definitions and group them by name, then sort those group by number of items in each group from greatest to smallest........................................ 41
Figure 4.39 Sorting all classes by their syntactic category............................... 42
Figure 5.1 An example of locating a function which news and deletes a variable within the same function in that order. ................................................................. 46
Figure 5.2 Steps within an XPath query. ....................................................... 48
Figure 5.3 srcQL Computational Stack .............................................................................. 57
Figure 5.4 The XPath representation used for the srcPat pattern new int......................... 58
Figure 5.5 Expression predicate sub-query optimization. This shows how the simple
srcPat new int is reduced in computational complexity...................................................... 60
Figure 5.6 The srcQL IR chain representation. This depicts the traversal taken by srcQL
during the evaluation of the srcPat expression. The Follow-sib is short for following-sibling which maps directly to the following sibling axis of XPath.......................... 61
Figure 5.7 Source code to search for the pattern new int within........................................ 62
Figure 5.8 Source code example after being run through srcML......................................... 63
Figure 5.9 Initial starting state for computational traversal.................................................. 64
Figure 5.10 The first located src:expr element within the srcML document..................... 64
Figure 5.11 Computational stack after locating the first <expr> element within the
document......................................................................................................................... 65
Figure 5.12 The stack after adding the child traversal.......................................................... 65
Figure 5.13 The stack state after the operator element is located......................................... 65
Figure 5.14 The reference structure between the XML tree representation and the current
state of the computational stack of srcQL. The line colors only serves to visually
differentiate lines between the stack and tree................................................................. 66
Figure 5.15 The state of the computational stack after evaluating the first sub query
within sub query expression predicate which yielded a match of the name element. 67
Figure 5.16 Collecting the matched context from on the computational stack.................. 68
Figure 5.17 The traversal algorithm used for implementing the descendant axis traversal.

Figure 6.1 srcQL: Locate all of the expression statements. This query simply searches on an XPath for all AST nodes with the default srcML namespace given as src and the element name expr_stmt.

Figure 6.2 srcQL: Locating the expression A + B. This is done within srcQL by using the srcPat only query. srcPat knows that if there is no trailing ; on an expression or declaration to treat the expression as a partial expression rather than a statement.

Figure 6.3 srcQL: Locate the all for loop which are within another for loop. search for all for loops using an XPath notation with an ancestor relationship with another for loop given by the WITHIN operator, which is also done using XPath notation.

Figure 6.4 srcQL: Locate all calls to foo. One would think this query may be matched as an XPath expression because it meets the XPath grammar notation for function calls, but this works because XPath doesn't define a function foo so this is treated as srcPat instad. Due to the closed world principal that srcQL and srcPat operate under, this actually matches foo with any number of parameters.

Figure 6.5 srcQL: Find all of the variable declarations from within the set of for loops.

The FIND operator provides a query which matches any variable declaration with the name i and searches all of the for loops for the variable declaration pattern which declares i.

Figure 6.6 srcQL: Find all instances of the expression new $T. This searches for all calls to operator new.
Figure 6.7 srcQL: Find all functions which contain calls to both foo and bar, and have the
same variable name for their first parameter. .......................................................... 77

Figure 6.8 srcQL: Locate all classes with a name that starts with a capital letter Q. srcQL
handles this by allowing for the application of a regular expression to the normalized
string values of syntactic variables. ........................................................................ 78

Figure 6.9 srcQL: Locating all classes with any name which contain the function
declarations of foo, bar, foobar, and fizz in any order............................................. 78

Figure 6.10 srcQL: Locate a call node with the ancestor that is condition which has parent
that is an if stmt. ........................................................................................................ 78

Figure 6.11 srcQL: Locate all functions which contain calls to fopen followed by fclose.
The srcQL followed by operator provides a relative association between two
expressions while still being within the context of a function. ................................. 78

Figure 6.12 srcQL: Find all instances of the variable $I being assigned to and being
deleted, in relative order within the context of a single function. ............................ 79

Figure 6.13 srcQL: Select the syntactic variable $V from within a sub-query given using
the FROM operator, where the variable $V exists as parameter to a function as well
as being used within the condition of an if statement. Due to the way that unification
works within srcQL this may also match expressions that occur in both locations as
well as variables......................................................................................................... 79

Figure 6.14 RASCAL: The Java grammar defined by RASCAL does not have a an
explicit concept of an expression statement............................................................ 79

Figure 6.15 RASCAL: Finding the expression A + B.................................................... 79
Figure 6.16 RASCAL: Locating a for loop within another for loop. ........................................... 80

Figure 6.17 RASCAL: Finding all expression statements to foo. This does not locate foo in all possible locations only when it is used as part of an expression statement. .... 80

Figure 6.18 RASCAL: Finding all of the for loops which contain a variable with the name i. ......................................................................................................................... 80

Figure 6.19 RASCAL: .................................................................................................................. 81

Figure 6.20 RASCAL: Finding all calls to both foo and bar with the same variable. ...... 82

Figure 6.21 RASCAL: Find all classes that start with a capital letter Q. ..................... 82

Figure 6.22 RASCAL: Find all classes which implement foo, bar,foobar, and fizz. ...... 82

Figure 6.23 RASCAL: Find all if statements with a call within the condition.................. 83

Figure 6.24 RASCAL: Find all functions with calls to fopen followed by fclose. ....... 83

Figure 6.25 RASCAL: Finding all of the variables that are either set using new or declared and assigned using new and set to null. ................................................................. 83

Figure 6.26 RASCAL: Finding all variables used within the condition of an if statement and within the block of the if statement. ................................................................. 84

Figure 6.27 Coccinelle: Locate all expressions e that are part of a statement. .............. 84

Figure 6.28 Coccinelle: Locate the expression statement A + B. There are caveats with this query due to the limitation of the language. ......................................................... 85

Figure 6.29 Coccinelle: Matching nested for loops. The ... within the pattern indicates that this pattern which matches everything. ................................................................. 85

Figure 6.30 Coccinelle: Match all calls to foo as an expression statement. ............... 85
Figure 6.31 Coccinelle: Locating for loops with the variable named i within them. This is done using multiple different patterns because there is not means of specifying relationships between two statements other than parent child relationships and in a for loop there is multiple locations in which that pattern could occur. .................... 86

Figure 6.32 Coccinelle: Locate all functions containing calls to both foo and bar and both with the same variable. ........................................................................................................ 87

Figure 6.33 Coccinelle: Locate all function calls within an if statement. ......................... 88

Figure 6.34 Coccinelle: Locate all functions with a call to both fopen and close in that order ........................................................................................................................................ 88

Figure 6.35 Coccinelle: Find functions that contain calls to both malloc and free of the same variable. ...................................................................................................................................... 88

Figure 6.36 Coccinelle: Locating all if statements which contain a variable and contain a call containing that variable within the matched if statement. ........................................ 89

Figure 6.37 JQuery: Find all types with a name that matches the regular expression requiring that the name start with a capital letter Q. .................................................... 90

Figure 6.38 JQuery: Find types which have methods with the name foo, bar, foobar, and fizz. ................................................................................................................................................ 90

Figure 6.39 PMD: Find all expression statements. ............................................................... 91

Figure 6.40 PMD: Find the expression A + B. ................................................................. 91

Figure 6.41 PMD: Find all for statements which are within other for statements. .......... 91

Figure 6.42 PMD: Find all calls to function foo. ............................................................. 91
Figure 6.43 PMD: Find all variable declarations of a variable with the name which occur within a for loop. ................................................................. 92

Figure 6.44 PMD: Find all expressions which contain the use of the AllocationPrefix of operator new. ................................................................................. 92

Figure 6.45 PMD: Find all of the function calls within the condition of an if statement. 92

Figure 6.46 SRCUPLE: Match the assignment statement C = A + B;. SCRUPLE does not provide a means of matching partial expressions only statements............. 94

Figure 6.47 SCRUPLE: Match all for statements within other for statements............ 94

Figure 6.48 SCRUPLE: Match all calls to foo, with no parameters. This only matches at the statement level and not the expression level......................................................... 94

Figure 6.49 SCRUPLE: Find functions containing a call to both foo and bar in any order. SCRUPLE has a strict notion of ordering so in order to do this one must query for both possible orderings................................................................. 96

Figure 6.50 SCRUPLE: find all function calls within the condition of an if statement. SCRUPLE does not provide a means for narrowing specific selections but this pattern will locate all if statements with a single function call within the condition. 96

Figure 6.51 SCRUPLE: Locating all functions that contain a call to both fopen and fclose on the same variable within the same function......................................................... 97

Figure 6.52 SCRUPLE: Find the allocation and free of the same variable within a function................................................................. 97
Figure 6.53 SCRUPLE: Due to the limitations of the SCRUPLE language one cannot directly query for variable names, the only thing that can be done is to match the location and allow the user to visually select the desired variables instead. .......... 98
Figure 6.54 CodeQuest: Locating a type with a name starting with a capital Q. .......... 98
Figure 6.55 CodeQuest: Find types which have methods with the name foo, bar, foobar, and fizz. .................................................................................................................. 99
Figure 6.56 Aria: Find all expression statements and print them. ........................................ 99
Figure 6.57 Aria: Find all nested for loops. ........................................................................ 100
Figure 6.58 Aria: Locate all calls to the function foo. .......................................................... 100
Figure 6.59 srcML: Find all expression statements. ............................................................. 101
Figure 6.60 srcML: Find an expression containing A + B. There is a limitation to expressing this within XPath and that is that this is actually an expression that contains the variable A, an operator +, and the variable B, in that respective order. ................................................................................................................................. 101
Figure 6.61 srcML: Find all for loops with an ancestor that is also a for loop. ................. 102
Figure 6.62 srcML: Find all calls to the function with name foo. ........................................ 102
Figure 6.63 srcML: Find all declarations of a variable named I which occur within a for loop. ........................................................................................................................................ 102
Figure 6.64 srcML: Find all expressions containing a use of operator new. ................... 103
Figure 6.65 srcML: Find all functions that contain function declarations of foo, bar, foobar, and fizz. ........................................................................................................................................ 103
Figure 6.66 srcML: Find all function calls within the condition of an if statement. ...... 103
LIST OF TABLES

Table 3.1 A list of all XPath Axes ................................................................. 17
Table 4.1 List of possible syntactic categories of a class and a description of how that
could occur within source code ..................................................................... 43
Table 5.1 Expression evaluation timings. Timings given using wall time rather than user
time because user time registers as 0.0s. ........................................................... 52
Table 5.2 Timings run on the first 1,000,000 lines of GCC run through srcML. Timings
given in the table below are given using user time rather than wall time because user
time is more accurate ......................................................................................... 53
Table 5.3 srcQL Query evaluation timings. All timing results are given in seconds. ..... 55
Table 6.1 List of test queries to implement in srcQL and other query language in order to
compare and ......................................................................................................... 75
Table 6.2 Queries by language and a check mark indicates that the language was able to
create a query which satisfied the query ............................................................. 104
DEDICATION

I'd like to dedicate this work to my friends and family who supported me through the last decade I spent in college. A special thanks goes out to my dog Gus who passed away in April, he was always there and always a good boy.
ACKNOWLEDGEMENTS

I'd like to thank my advisor Dr. Jonathan Maletic for giving the opportunity to complete this work. I wouldn't be here without his guidance, patience, and direction. Dr. Michael Collard for holding my work to a high standard and directing maintenance efforts on srcML. Every that has ever been in our research group, with special thanks to Dr. Michael Decker, and Dr. Christian Newman.

Brian M. Bartman

July 13, 2017, Kent, Ohio
CHAPTER 1

Introduction

Currently, search tools in IDEs (Integrated Development Environment) typically use regular-expression matching. The main drawback with regular expression or text-based search tools is that too many, non-relevant, parts of source code are matched. This leads to developers manually filtering the results (sometimes 1000’s of matches). While these matching techniques are very powerful, they are intended to be applied across any text-based document.

The research presented here takes a different viewpoint. Since developers are searching and exploring source code, we feel that the search and matching tools will benefit greatly from being inherently aware of the underlying syntax of the programming language. We present our implementation of a syntax-aware query language that is built on top of a syntax aware pattern-matching language. The query language is called srcQL (source code Query Language) and the pattern matching language is called srcPat (source code Pattern matching).

The objective is to give developers the ability to easily construct queries that take into account the syntactic features in the programming language. For example, a developer may want to find all functions in a system that have a call to both open and close within the body. Furthermore, the ordering may be of interest; open before close. This can be further refined to locating all calls to open within a guard. That is, find all
calls to open within a body of an if-statement. These types of queries are very difficult, if not impossible, to describe as regular expressions.

Sometimes a developer must search through thousands, tens-of-thousands, even tens-of-millions, of lines of source code or more in order to find what they are looking for. There are many tools that are capable of searching source code but many of them require the source code to be compilable, like .QL [Moor et al. 2008], or the developer to have intimate knowledge of the source code's AST structure, such as TXL [Cordy 2006]. While being compilable is not an issue for everyone, often times a developer simply wants to locate the language element without having to construct a database to query against. Having to understand a how a language AST is composed is often best left to compiler and language experts. For example, C++ is a complicated language to implement, just look at tools like clang and GCC, both are non-trivial pieces of software that both use different but similar AST structures and learning the in's and out's of the C++ AST structure a non-trivial task. While this level of detail is necessary within a compiler implementation, it should not be required when simply searching source code.

Source code querying and pattern matching is used in almost every aspect of every tool that operates on source code in some way. This includes everything from a simple GREP, to other more complicated tasks such as fact extraction and source code re-documentation [Dragan, Collard, Maletic 2006], to heavy weight analysis of the meta information of a program using a tool like RASCAL [Klint, van der Storm, Vinju 2009]. There are still open issues within the source code querying area of study. This is because source code querying language do not use source code examples as input for their
queries. Using source code examples without a given context can lead to ambiguities for languages like C/C++ this approach would still see more through a wider range of the community because it's does not require an expert to construct a complex query.

Since developers are searching and exploring source code, we feel that the search and matching tools will benefit greatly from being inherently aware of the underlying syntax of the programming language. We feel that extending the language to incorporate scope awareness and logical ordering of statements and patterns.

1.1 Research Contributions

The research contributions are as follows:

- Development of the srcQL and srcPat languages
- Development of a language a syntax aware query language
- Development of a language that uses source code examples as queries
- Present a comparison between srcQL and other query languages.

The standard for the srcQL programming language is provided in Appended B. The language standard provides a more formal definition of the srcQL and srcPat languages than are presented in the main body of this paper. A language to language comparison of existing source code querying technologies is presented in the section titled Language to Language Comparison. This excrement shows that srcQL, which does not perform compilation or use type constructors in order to construct matches to queries within the AST, is as capable as those languages which do full compilation, use type constructors, or require you to give a full grammar specification in order to compose any queries.
What makes srcQL novel is that it uses a query by example functionality in order to compose matches to other parts of the source code while respecting the syntax of the languages, including but not limited to, in complete or partial expressions. No other language allows for query by source code example and handles partial expressions at the same time. Adding to that the ability to add variable parts to those patterns makes srcQL a very expressive and simple language. While srcQL doesn't provide all of the general computing power available in some languages it makes up for in pattern expressability.

1.2 Organization of Dissertation

The paper is organized in the following manner. The background and related work are presented. Preliminary work is which lead up to the creation of the srcQL language is presented in CHAPTER 3. This is followed by an in depth overview of the srcQL language in CHAPTER 4, and the optimization and implementation strategies employed in CHAPTER 5. A language to language comparison between srcQL and other languages is presented in CHAPTER 6. Finally, the conclusions and future work of srcQL is presented in CHAPTER 7.
CHAPTER 2

Background and Related Work

There has been a large amount of work done on querying software. While many tools are meant for source code transformation, TXL [Cordy 2006], RASCAL [Klint, van der Storm, Vinju 2009], DMS [Baxter, Pidgeon, Mehlich 2004], ELAN [Borovanský et al. 1996], Stratego [Visser 2001], and Coccinelle [Padioleau, Lawall, Hansen, Muller 2008], none of them are specifically designed for use in querying software. We compare mainly against languages used for pattern matching as part of term rewriting because srcQL is designed to locate transformation site patterns as well as be a general purpose query language. While there are other languages designed to query source code they are typically only limited to a single task such as locating sets of statements, answering questions about the type system, or other sorts of fact extraction.

2.1 Other Source Code Query Languages

Few languages have been created for the purpose of source code querying. Those that have been created normally require the user to have intimate knowledge of some form of internal tree structure, such as an AST, or the language grammar. If they implement a language explicitly for querying, the user must also learn a new language that, based on studies of previously created tools, do not follow typical query-language structure and take time to understand. They also require compilation of the source code.

We have studied tools that are used for Program Transformation. This is primarily because Program Transformation requires some sort of matching to specify the location...
for transformation. For the program transformation languages, we assume that minimally the technique used to locate transformation sites could also be used as a query language. Here are some of the languages we show: JQuery[De Volder 2006], PMD[Dangel], SCRUPLE[Paul 1992], TAWK [Atkinson, Griswold 2006], CodeQuest [Hajiyev, Verbaere, de Moor, De Volder 2005], Aria [Borovanský et al. 1996], A*[Ladd, Ramming 1995], MAWK[Griswold, Atkinson, McCurdy 1996], JRelCal [Rademaker 2008], LINQ, XPath, XQuery, and srcML.

TXL[Cordy 2006], RASCAL[Klint, van der Storm, Vinju 2009], DMS[Baxter, Pidgeon, Mehlich 2004], ELAN[Borovanský et al. 1996], Stratego[Visser 2001], and Coccinelle[Padioleau, Lawall, Hansen, Muller 2008] are all languages that perform Program Transformations. With the exception of Coccinelle and TXL, they require full compilation and have the capability to, minimally, match specific sections of code specified by the user. This can be considered a form of querying even if, with these tools, the goal is the transformation, not the query. Coccinelle uses variables to substitute isomorphisms in source code. To find the location of these isomorphisms, it uses a model checking algorithm called CTL. Its application and srcPat only overlap in terms of applicability to the issue of program transformation. RASCAL defines an imperative programming syntax to specify queries and transformation. Its language works on an AST. DMS, STRATEGO, and ELAN are all similar in that they define matching and transformation in terms of an AST. TXL defines a functional syntax for the same reason as RASCAL and works at the level of language grammar. The above tools, while able to perform some querying, are difficult to use for the purpose of querying. The functional,
imperative, and rule-based languages they employ are hard to use for describing queries. srcQL uses the syntax of the language it’s working on to describe queries, which allows the programmer to work with readily available knowledge instead of having to understand low-level concepts like language grammars.

SOUL [Roover, Noguera, Kellens, Jonckers 2011], and RScript [Klint 2003] provide functional languages in order to do matching and transformation. Browse-By-Query [MacDonald 2011] (BBQ) provides a language for querying source code that’s easy to learn. However, it does not provide an easy means of specifying patterns because one has to have initial knowledge of the source code in order to perform queries upon it and there is lack of documentation about the language. SOUL, RScript, SemmleCode [Verbaere, Hajiyev, Moor 2007], .QL [Moor et al. 2008], Java Tools Language (JTL) [Cohen, Gil, Maman 2006], JRelCal [Rademaker 2008], and CrocoPat [Beyer, Lewerentz 2003] all require full compilation in order to execute a query. JTL, however, does not require explicit knowledge of the internal AST. In order to do more complex queries additional functional programming is used.

JTransformer [Kniesel, Bardey 2006] performs querying in order to provide aspect weaving into a Java program, and as such requires the AST in order perform queries.

Functional languages are not a natural way to specify source code queries. The tools mentioned above are not all explicitly for source code querying but their users do have to accomplish some sort of matching. srcQL can remove the need to force users to learn complicated grammar or expression matching techniques and, instead, use the easy-to-learn format of srcQL expressions. This is what makes srcQL a great tool for querying.
XPath, LINQ, and XQuery are the most widely used tools when it comes to querying the XML structure of srcML, all of which were studied thoroughly. XPath is the most widely used XML querying tool. XPath provides a simple language to query XML, however, the downside to it is that it can require experts in XPath to write more complex queries. All but trivial queries on the srcML XML structure require the expert level knowledge and as such most developers shy away from using XPath. What XPath lacks in readability, it makes up for in speed. Execution of an XPath query is extremely fast compared to other querying tools we have studied.

LINQ is extremely easy to write and allows for the maximum amount of expressiveness and readability when querying the srcML XML structure. However, what the LINQ language gains in readability it loses in speed. This is because LINQ is executed from the language it was written in and, as such, is unable to make more complex optimizations that are available to tools such as XPath. While LINQ provides the necessary extensions and syntax for compiling and executing queries on source code the degradation in speed makes it too slow for the implementation of srcQL and srcPat.

The main down side to all three of the languages, XPath, LINQ and XQuery is that they only allow the user to operate at the XML level of the source code. This prevents the user from simply writing queries in terms of source code which is what they would usually do.

Another tool used for ad-hoc queries on source code is grep. Grep and tools like it are great for searching for regular expression patterns within arbitrary text documents; however, they have their limitations. Using a regular expression search only considers
the text of a document and not the internal structure of source code. This makes it
impossible to locate some segments of source code without manual inspection of results.

Another type of query language is one that simply extends an existing language to
incorporate or extend another language such as AWK, in this case, to allow for some
source code like relationships to be specified as part of the language. Two languages that
extend AWK to better work when querying source code are TAWK[Atkinson, Griswold
2006] and MAWK[Griswold, Atkinson, McCurdy 1996]. These types of languages
suffer from taking a textual matching language and attempting to make it work on trees
which it wasn't originally designed to do. Both TAWK and MAWK are similar in that
they are syntax only languages for which you supply a pseudo-grammar representation to
match specific patterns within the syntax of a programming language. In the end both
languages up similar to tools like YACC[Johnson 1975], in that both languages look very
similar to a rule based grammar. In the case of both MAWK and TAWK partial
compilation is preformed to build up a symbol table and some type information.

There is another vein of work that's related to srcQL and that is fact extraction.
There are a great deal of fact extracting libraries which allow users to query databases or
source code in order to gain a better understanding. We mainly focus on those fact
extracting tools which implement query languages, because srcQL can be used to extract
facts from source code in the form of patterns. For example, the LSME[Murphy, Notkin
1995] tool is a fact extractor that operates on source code tokens at a language level by
defining a grammar to match these tokens similar to TXL. LSME leverages heuristics in
order to better prune it's results which isn't something that we are interested in within
srcQL. While there are many languages that exist some have gone as far as implementing pattern matching directly within the compiler itself. This compiler extension allows for the language grammar to be matched as a pattern within the compilers AST. While this is very useful for the compiler writers this type of pattern matching isn’t something that a normal programmer would need or want to learn. This is due to the fact that the abstract syntax tree of a compiler is a representation of a program from a compilers prospective, and not from a developer’s prospective. For example, GENOA [Devanbu 1992] is one such language which is implemented as part of a compiler and allows specific matches within the AST to be made a pattern specification language. Another example of this is clang’s pattern matching library libASTMatchers which allows patterns within the compiler to be matched and interacted with. While not to diminish the value of tools that match patterns within a compiler, which can be extremely useful for things like optimization, or locating bugs, compilers are not designed to handle things like comments or keep whitespace from programmers, so going backwards from a program into a file can be confusing and provide an unusual prospective for the programmer.

2.2 Discussion

There is still room for improvement over previous techniques. While there are languages for interacting with source code as an AST none of the provided languages use source code examples as their input and the majority of them require either full or partial compilation or require that the query be specified as a set of rules to match when querying the AST. There is still room for improvement of source code querying languages. While a large number of these technologies are grounded in mathematics this
can make writing them difficult and confusing for developers not familiar with that style of programming and the math that it's related to.
CHAPTER 3

Preliminary Work

In order to understand the contributions of this work one must first understand the underlying infrastructure and the steps which lead to developing this language. We now present srcML, XPath, and the development phases that srcQL when through.

3.1 srcML

srcML is an XML format for source code that embeds AST information into the source-code text. This AST information is in the form of XML elements. srcML preserves all original source-code text, and can be used to write identity transformations from code to code. In addition, it can represent code before preprocessing; meaning srcML does not remove include statements or include guards. This further preserves the developer’s view of the source code. The srcML infrastructure provides tools to convert source code to the srcML format and back. The tools and the format are lightweight, highly scalable, and robust to programming language variations and errors.

srcML has been used for a variety of tasks including transformation[Collard, Decker, Maletic 2011], slicing[Newman et al. 2016], and fact extraction[Newman, Maletic, Collard 2016]. To get an idea of what srcML looks like, I now give an example. In Figure 3.1, there is an example piece of code with a single function in it. In Figure 3.2, we show the output of srcML applied to the example in. The primary characteristic of the output that I want to highlight is the tree structure. srcML generates an XML tree that wraps code in AST information. Hence, srcML’s tree is an AST. The way srcTL works
is by applying changes to this AST and then using srcML’s toolset to remove the srcML and produce the transformed code.

```cpp
#include "rotate.h"

// rotate three values
void rotate(int& n1, int& n2, int& n3)
{
    // copy original values
    int tn1 = n1, tn2 = n2, tn3 = n3;
    // move
    n1 = tn3;
    n2 = tn1;
    n3 = tn2;
}
```

Figure 3.1 A simple rotation example where three values are swapped between one another.

```xml
<!-- rotate three values--> <function><type>void</type> <name>rotate</name> <formal-params>(<param><type>int&amp;</type> <name>n1</name></param>, <param><type>int&amp;</type> <name>n2</name></param>, <param><type>int&amp;</type> <name>n3</name></param>)</formal-params> <block>{
    <comment type="line">// copy original values</comment>
    <decl-stmt>decl<int> <name>tn1</name> = <name>n1</name>, <name>tn2</name> = <name>n2</name>, <name>tn3</name> = <name>n3</name></decl-stmt>
    <comment type="line">// move</comment>
    <expr-stmt>expr<n1</name> = <name>tn3</name></expr-stmt>
    <expr-stmt>expr<n2</name> = <name>tn1</name></expr-stmt>
    <expr-stmt>expr<n3</name> = <name>tn2</name></expr-stmt>
}</block></function>
```

Figure 3.2 The result of running srcML on source code from the rotation example in Figure 3.1.

3.2 XPath

Currently, XPath is the primary way for uses to interact with srcML. I will provide enough details in this section for the reader to understand basic XPath with the goal of explaining why XPath is an insufficient query language for source code. XPath is
a query language for XML documents. It provides a way for its users to specify and retrieve nodes within an XML document. XPath scripts look a lot like file paths in that, given a node, to address this node’s child, you add a ‘/’ and then the name of the child. Hence, a pattern like: parent/child/grandchild is the general form of an XPath expression from a parent to its grandchild. Let’s take a few examples.

Figure 3.3 The XPath axes relationships between the current self node A all other nodes relative to a given axis.
In Figure 3.3, let us say we are currently at node $A$, which can also be referred to as self. If I want to access the leftmost node labeled $E$ that is a child of $A$, then I do the following:

\[
\text{self::*\[/E]
\]

**Figure 3.4 Example of selecting all child elements with the name E.**

This will select the leftmost $E$, since it is first in the child list if we scan from left to right. What if we wanted the second $E$ instead? We would do the following:

\[
\text{self::*\[/E[2]}
\]

**Figure 3.5 Selecting the second child element with the name E.**

In XPath, square brackets (i.e., []) are used to specify predicates. Predicates constrain which nodes may be selected by asserting that selected nodes satisfy a specific condition. In this case, the condition is that the node must be second in the child list. The following would also work:

\[
\text{self::*\[/E[D]}
\]

**Figure 3.6 Selecting all child elements with the name E and a child element with the name D.**

Where we constrain that the $E$ node we want must have a child node labeled $D$ in order to be selected by the query. This causes XPath to skip the first $E$ since it has no child. Next, let’s use XPath to select $C$, which is a grandchild of $A$.

\[
\text{self::*\[/B/C}
\]

**Figure 3.7 Selecting a child element $C$ with parent element $B$.**
This is the pattern I gave earlier: parent/child/grandchild. With the basics down, let us now look at XPath Axes. An XPath Axis indicates a navigation direction when looking for a node in a document. The Axes are given in Table 3.1.

Notice that the axes listed above correspond to the axis labels in Figure 3.3. The figure shows exactly what set of nodes each axis corresponds to relative to the self axis, which corresponds to the node labeled A. To use an axis, one merely needs to use the syntax axis::node. I will give a few examples.

To select A’s grandchild, C, I use the descendant axis:

```
self::*//C
```

**Figure 3.8 Selecting all elements with the name C which are a descendant of the current element.**

This lets me grab C without having to state each child node along the path from A to C. Let us take a look at another example. If we want to access node Z, which is A’s parent’s sibling, we can use the preceding axis. The XPath is as follows:

```
self::*//Z
```

**Figure 3.9 Selecting all preceding elements within the document with the name Z.**

srcQL leverages the libxml2 document object model in order to evaluate XPaths upon. While libxml2 does provide an implementation of XPath, srcQL also implements the XPath 1.0 standard in order to fully compile and extend the srcQL language. XPath is used within srcQL in order to specify basic context for which patterns can occur, parent child relationships between patterns and their ancestor AST nodes, and to provide a means of interacting with additional markup within a srcML document. The addition of the full XPath 1.0 standard allows for any extension to srcQL to be done by simply
annotating the srcML with new information and using XPath to interact with the new information, without having to alter srcQL.

While it is possible to use XPath to query AST relationships between nodes in an AST. XPath does not have the capability understand variable scoping, back references, or unification of logical programming elements, nor should it. XPath is a fine tree querying language, but it wasn't designed to interact with source code directly. What XPath lacks is the ability to scope axes to a limited section of the tree, reference back to elements that were the result of a previous step and to collect special selections of elements within the tree and save them for later use.

Table 3.1 A list of all XPath Axes.

<table>
<thead>
<tr>
<th>Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
</tr>
<tr>
<td>ancestor-or-self</td>
</tr>
<tr>
<td>attribute</td>
</tr>
<tr>
<td>child</td>
</tr>
<tr>
<td>descendant</td>
</tr>
<tr>
<td>descendant-or-self</td>
</tr>
<tr>
<td>following</td>
</tr>
<tr>
<td>following-sibling</td>
</tr>
<tr>
<td>namespace</td>
</tr>
<tr>
<td>parent</td>
</tr>
<tr>
<td>preceding</td>
</tr>
<tr>
<td>preceding-sibling</td>
</tr>
<tr>
<td>self</td>
</tr>
</tbody>
</table>
CHAPTER 4

srcQL: Source code Query Language

We now understand how source code's syntax can be leveraged in order to create a query that matches syntactic patterns within source code. Additionally, we have explored other existing techniques and found a need for a query language the operates using language examples, does not require compilation to evaluate a query, provides access to source code related relationships, all while scaling to run on very large bodies of source code very fast. The language must accomplish all of this while still being easily understandable by a typical developer. While the requirement of being easily understandable is difficult to verify, we attempted to create a language in such a way that it's easy to read and does not require a large amount additional learning in order to use the query language effectively. This was done through the use of source code examples as syntactic queries and leveraging the look of SQL. The use of source code as input removes the need for the user to have in depth knowledge of the source code AST while still allowing that kind of access through XPath if necessary.

Typical query languages require sets of complex rules in order to compose a query[Collard, Decker, Maletic 2011; Cordy, Dean, Malton, Schneider 2001; Klint, van der Storm, Vinju 2009]. srcQL is designed to remove these types of complicated details from specifying a source code pattern by simply using source code itself as the input to a query. We call the pattern part of the language srcPat which stands for source code pattern. srcPat also includes the ability to use logical variables within a piece of source code to all of matching to be more generic. We than added source code relations as part
of the high level language in order to relate pieces of source code together we call this part of the language srcQL. Due to the complexities of possible relationships within the AST we further extended srcQL to include XPath. The XPath language allows srcQL to specify relationships between individual elements of the AST that are not easily expressible as a single piece of source code as well as provide additional generality to the matching process. srcQL is designed to work with srcML in order to provide fast light weight markup of the AST of a program.

srcQL is a query language specifically for source code. It uses srcPat and XPath as input patterns, while providing a way to specify complex relationships those patterns. The objective of this research is to provide a syntax aware query language that uses source code patterns as input, and hides the complexities of learning an entire AST, while still having the power necessary to perform program transformations.

The srcQL language leverages srcML as its underlying AST structure in order perform a query. This allows for other annotation to be added to an XML document and queried against. Meaning that any additional information added as an attribute or an XML node can easily be queried against in order to extend the srcQL language. For example, callee information could be added to individual function declarations as an attribute or as an element to a function which could then be accessed during query evaluation.

4.1 srcPat

The srcPat (source code pattern) language is used for specifying patterns as part of the source code. Basically, any expression, declaration, statement, function or any
other language element can be used as the input to a srcPat query. srcPat uses the input to convert the source code pattern into srcML and use that output to build a srcPat AST by traversing the returned srcML expression. The AST only consists of XML traversals and predicates necessary to match the provided pattern within a srcML archive.

srcPat also provides the means of replacing names and some language elements with syntactic variables. A syntactic variable allows for expressions to be generalized. For example, to locate all uses of new int the pattern would be,

```
new int
```

**Figure 4.1** srcPat matching a specific call to new int.

but to locate all calls to operator new, a syntactic variable is used instead.

```
new $T
```

**Figure 4.2** match all calls to operator new.

srcQL and srcPat also provide unification of variables through string comparison of syntactic variables. So if one wanted to locate all variables of a type that are initialized with new of the same type that can be specified as follows:

```
$T* $Var = new $T
```

**Figure 4.3** Unification of syntactic variables $T to be the same type

Queries generated with srcPat make a closed world assumption about their input. Meaning that they do not search or assert against elements which are not there or that they should occur only a single time. This is apparent when querying for a function with a given number of parameters. For example, searching for a function definition with a any name:
Figure 4.4 Searching for all function declarations that have a void return type.

One would expect the query to simply return only functions with no parameters that have no body, but because of the closed world assumption made by srcPat this instead return all functions which have a void return type, any number of parameters, and any contents of the body of the function. This allows for more general queries to be composed easily. While the closed world assumption may seem like a limitation for srcPat, but srcQL provides a means for specifying relationships between pattern, as well as involving XPath where needed.

The processes of creating a querible expression from srcPat expression is fairly straightforward. A srcPat expression is transformed by tokenizing the expression and identifying all syntactic variables within the expression so that they can be unified during query evaluation.

The query is than run through srcML to produce the AST for the given pattern. srcML returns an XML document representing the AST structure of the query. Consider the pattern `new int;` this patterns generates the following XML document:

```xml
<unit xmlns="http://www.srcML.org/srcML/src"
   xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++" filename="srcPat_1.cpp">
   <expr_stmt><expr><operator>new</operator><name>int</name></expr></expr_stmt>
</unit>
```

Figure 4.5 The result of running `new int;` through srcML.

The XML document is inspected to locate the first element of a pattern, which is in this case is `<expr_stmt>`. Determining the starting element of an expression requires
some logic specific to srcML in order to allow for matching of partial expressions and declarations, rather than simply matching them in their statement form. The starting element of a pattern or search context of a srcPat expression is determined using the logic in Figure 4.6.

The reprocessing of expressions and declarations without a semi-colon allows srcPat to determine when a given query is only a part of an expression or declaration instead of an expression or declaration statement. This allows for expressions such as $x = 5$ to be located inside of if statements condition or $++x$ inside of a for loop group. Being able to specify incomplete parts of expressions makes srcPat extremely powerful.
**srcPat Search Context Selection Logic**

processingDoc = run pattern through srcML.
If processingDoc's unit element has 2 or more children
   return first child element of unit
else if the unit has only a single child element
   if the first child element is an expr_stmt or decl_stmt
       if the expression didn't contain a ';'
           append a semi-colon to the expression
           processingDoc = run expression + ';' through srcML
       return the expr or decl element
   else
     return first child element of unit
else
   return first child element of unit

**Figure 4.6 Logic used to determine the search context for srcPat.**

srcPat patterns and srcQL are represented using a simple model consisting of a query and list of predicates that make up the traversal. An element is matched once all elements within the list of predicates evaluate to true. The list of predicates maps directly to a set of nested for loops, assertions predicates, and collection predicates. Each predicate results in either true or false which can be based on the result of evaluating the next predicate in the chain, or evaluating an assertion against the input context.

**Figure 4.7 Control flow generated for matching a srcPat pattern.**
Within srcPat there is a limited number of predicates, ancestor, descendant, child, following-sibling, reverse-following-siblings, select-all-children, select-all-following-siblings, select-children-up-to, select-siblings-up-to, set context, collect-context, and an expression predicate. All predicates that start with select are used for capturing syntactic variables within a srcPat expression. All syntactic variables are internally represented as a range of sibling elements where the end element is another element or null to indicate that all following siblings are to be used. These predicates make up the core nodes used for matching a pattern within srcPat and additional nodes are used to create a super set of XPath as well. An explanation of each predicate follows:

- **Movement Predicates**
  - Ancestor
    - Visit all ancestor elements of the input context.
  - Descendant
    - Visit all descendant nodes of the input context.
  - Child
    - Visit each child of the input context.
  - Following-Sibling
    - Visit siblings which occur after the input context in document order.
  - Reverse-Following-Sibling
- Visits all following siblings in reverse document order starting with the last child. This is used to aid in matching syntactic variables for which the last valid match is needed.
  - Set Context
    - Alters the input context to references the output context from a previous predicate. This is used because a predicate can really only output to one other predicate. The setting of a context allows for the output of a context from a previous predicate to be reused.

- Syntactic Variable Selecting Predicates
  - Select-All-Children
    - Selects all children of the input context as the value of a syntactic variable.
  - Select-All-Following-Siblings
    - Selects all following siblings after the input context to be a syntactic variable. This could be 0 or more elements.
  - Select-Children-Up-To
    - Selects all children up to the last occurrence of a pattern. This axes works in consort with the Reverse-Following-Sibling predicate to quickly locate the last element of the syntactic variable.
  - Select-Siblings-Up-To
    - Selects all siblings after the input context up to the last occurrence of a pattern. This axes works in consort with the Reverse-
Following-Sibling predicate to quickly locate the last element of the syntactic variable.

- Other Predicates
  - Expression Predicate
    - An expression predicate takes the input context and evaluates an expression against that context which results in either true or false. An example of this is checking the type of a node to make sure it's an XML element, checking the namespace of an element, or checking the name of an element.
  - collect-context
    - Takes the input context and stores it into a variable or collection. This is how the results are returned from a query.

After locating the search context the AST is traversed in document order starting with the search context and continuing onto all following siblings. Each element visited creates two things, the first is a traversal predicate which modifies the context resulting in a new context, and the second is a set of assertions against the modified context to determine if the traversed node is a match to the pattern.
bool isFirstElement = true;
map<xmlNodePtr, Predicate> nodeToPred;

bool startElement(xmlNodePtr node) {
    if(node is a syntactic variable) {
        if(isFirstElement) {
            // the first XML element can not be a
            // syntactic variable due to srcML
            // structure.
            exit(1);
        }
        xmlNodePtr prevElementSib = getPreviousElementSib(node);
        xmlNodePtr nextElementSib = getNextElementSib(node);
        if (nextElementSib && prevElementSib) {
            // insert set context to previous sibling.
            query->appendSetContext(nodeToPred[prevSib])
            query->appendSelectSiblingsUpTo();
        } else if (!nextElementSib && prevElementSib) {
            query->appendSetContext(nodeToPred[prevSib])
            query->appendSelectAllFollowingSiblings();
        } else if (nextElementSib && !prevElementSib) {
            query->appendSelectChildrenUpTo();
        } else if (!nextElementSib && !prevElementSib) {
            query->appendSelectAllChildren();
        }
        return false;
    } else {
        // The first element always needs to be the self axis because
        that's // how the queries actually work correctly with
        // other queries proceeding them.
        if (isFirstElement) {
            isFirstElement = false;

            create ancestor, or descendants based
            on where the srcPat is located
            within srcQL

            return true;
        }

    }
    xmlNodePtr previousSib = getPreviousElementSib(node);
    if (node->name == "name") {
        if (node->children == nullptr) {
            if (node->parent->name != "name") {
                if (previousSib) {
                    query->appendSetContext(previousSib);
                    nextPredicate = query->createFollowingSiblingPredicate(true);
                } else {
                    nextPredicate = query->createChildPredicate(true);
                }
            } else {
                nextPredicate = query->createChildPredicate(true);
            }
        } else {
    }
Figure 4.8 srcPat query generation algorithm.

A srcPat pattern is built using the following algorithm during traversal to construct a query for matching a given pattern. During the traversal of the XML
document all mixed content is not visited during the traversal. Each time an element is reached for the first time the startElement is called and every time a text node is visited text is called. A map of which element was used to create which predicate is kept, as well as a list of known syntactic variables from within an expression. The startElement function returns true to traverse the sub-tree of the current node and false it skips it and continue to the following sibling element. The resulting query can then be compiled as part of srcQL.

### 4.2 srcQL Language

srcQL is a syntax-aware query language that leverages srcPat, XPath, and srcML. srcQL acts as a bridge between multiple srcPat patterns by providing search context, relational operators, and inter-pattern unification. The basic syntax for srcQL is of the form:

```
FIND search-context CONTAINS pattern
```

**Figure 4.9 srcQL basic query structure.**

This is read as “Find all search contexts that contain the pattern”. The *search context* is the syntactic category to be searched upon as well as what is returned as a result of the query. The search context is any valid syntactic category within the language grammar as defined in srcML. The *pattern* describes the syntactic structure being searched for. The pattern can be any srcPat or XPath expression. In addition, srcQL supports operators including WITHIN, FOLLOWED BY, WHERE, GROUP BY, and ORDER BY. These provide a way to specify such things as partial orderings and
constraints on the search space. The context as well as each operator is now described in more detail.

### 4.2.1 Implicit srcPat

A srcQL query need not start with an operator. When a query does not start with `FIND` or the `SET` operator the input query is assumed to be a source code pattern instead. srcQL converts the input to simply be `FIND` the given pattern. For example,

```
new $T
```

**Figure 4.10 Implicit srcPat, srcQL query. This provides a means for easily searching for simple patterns rather than having to write FIND before all queries.**

Is transformed internally into,

```
FIND new $T
```

**Figure 4.11 The transformed query from Figure 4.10**

### 4.2.2 FIND Context

The context of a srcQL query is the syntactic category that will be searched for and matched using XPath or srcPat. The find context refers to the node (in the abstract syntax tree) being selected by an XPath or srcPat expression. The root node can either come from the user as part of an XPath or be inferred from a srcPat expression. For example, if a context is specified by the XPath `src:class`, then the only elements searched will be the children of `src:class` elements. The elements returned will be those `src:classes` that match criteria provided by other operators.

`FIND` is one of the operator uses the grammar from XPath to infer if the provided context is an XPath or if it should be treated as source code. For example, if the context provided can be parsed into an XPath than it is treated as an XPath:
If the provided context isn't parsed as an XPath it is treated as source code and processed using srcPat to infer the context based on the context:

```
FIND $R $F(){
```

**Figure 4.12 Implicit FIND XPath.**

A similar expression can be used which allows FIND to simply be given a * and then FIND uses the syntactic context from CONTAINS. For example the following produces the same result as the previous query:

```
FIND * CONTAINS $R $F(){
```

**Figure 4.14 FIND guess context for contains.**

In cases where the FIND context is ambiguous between XPath and source code, the keywords src and XPath can be used to prefix a pattern to explicitly set the type of expression to be processed as srcPat or XPath. For example, in the case of A * B, this is either the XPath of the query A multiplied by the result of query B, or a pointer declaration of a variable B of type A*.

```
FIND A * B
```

**Figure 4.15 FIND the XPath A * B.**

This could be disambiguated using the src keyword with the FIND operator.

```
FIND src A * B
```

**Figure 4.16 FIND the srcPat pattern A* B of a pointer declaration**

The FIND operator can also be used to select individual syntactic variables from a CONTAINS or FROM pattern. This allows for queries selecting all of the names used
inside as part of a pattern. For example, to select all of the variable names of all types used with a new from the entire program:

```
FIND $T CONTAINS new $T
```

Figure 4.17 Locate the name of all types used with operator new.

### 4.2.3 srcQL Operators

srcQL provides operators for structural relations within source code that are otherwise very difficult to describe using srcPat or XPath alone. The operator keywords are used to separate various srcPat expressions from one another.

The srcQL language consists of six operators. A description of each is given below along with an example of how it is used and an explanation to clarify its behavior. Some of the operators are drawn directly from SQL while others came about because we are querying a tree and not a table. The srcQL language provides an unordered relation, a partial ordering relationship between children, child to parent relationship, and basic predicates that can use regular expressions. We now describe each of these operators in detail.

#### 4.2.4 CONTAINS

CONTAINS is an operator used to locate multiple different expressions within the search context specified by FIND. CONTAINS provides an unordered relationship between child expressions within the search context.

We will now look at a few examples; the following query finds all functions that contain a certain usage of the new operator.

```
FIND src:function CONTAINS $var = new $T
```

Figure 4.18 Find all functions that contain a certain usage of the new operator.
The result of this query is a set of functions. We can further refine the search by looking for functions that contain both a new and delete. srcQL makes this simple by allowing for multiple `CONTAINS`, as shown below:

```plaintext
FIND src:function
CONTAINS $var = new $T
CONTAINS delete $var
```

**Figure 4.19** Find all functions which allocate a variable using new and delete the same variable.

The query finds all functions (FIND `src:function`) which contain an assignment to a variable with a call to the new operator (CONTAINS `$var = new $T`) and also has the variable matched by `$var` deleted later in the search context that, in this case, is what was matched by `src:function` (CONTAINS `delete $var`).

`CONTAINS`, in the above example, behaves such that it only matches functions that contain one or more ‘new’ and ‘delete’ keywords with matching `$var` in their expressions.

The following query is an example of using XPath instead of the default srcPat. It finds the use of operator new within a declaration statement:

```plaintext
FIND src:decl_stmt
CONTAINS XPath src:operator[.="new"]
```

**Figure 4.20** Find all of the declaration statements that contain an XML `<operator>new</operator>` in srcML.

Because `CONTAINS` supports an unordered relationship among expressions within the `FIND` context, multiple `CONTAINS` can be used while describing one syntactic context.
If two \texttt{CONTAINS} operators are used that both search for the same expression, the parts of the AST they match are not restricted to being different, meaning that they can both match the same nodes within the AST. For example

\begin{verbatim}
FIND src:function
CONTAINS write()
CONTAINS write()
\end{verbatim}

\textbf{Figure 4.21} Searching for the same thing twice with the \texttt{contains} has no effect on cardinality.

Will have the same result as,

\begin{verbatim}
FIND src:function
CONTAINS write()
\end{verbatim}

\textbf{Figure 4.22} Functionally equivalent to the query in Figure 4.21

This means that \texttt{CONTAINS} cannot be used to search for cardinality of an expression within a context. Providing two \texttt{CONTAINS} to the same search context with the same pattern only matters when you add additional descriptive operators to them such as \texttt{FOLLOWED BY} or \texttt{WITHIN} which are described later. One implication of this is that \texttt{CONTAINS} cannot be used to provide a count operation to a particular pattern within a search context. The count operation can be obtained through the use of the \texttt{ORDER BY} and \texttt{GROUP BY} operators.

\subsection*{4.2.5 \texttt{FOLLOWED BY}}

\texttt{FOLLOWED BY} is used to specify ordering of statements in a query. An expression matched with \texttt{FOLLOWED BY} is limited to occur within the sub-tree matched by the search context within document order but after the preceding \texttt{CONTAINS}
or FOLLOWED BY. For example, to find a function which contains an `fopen()` followed by an `fclose()` on the same variable is as follows:

```
FIND src:function
CONTAINS $X = fopen()
FOLLOWED BY fclose($X)
```

**Figure 4.23** Find all functions that open and close a C IO file stream.

The partial ordering can be chained multiple times to match sequences of more than two statements. For example, to locate a function which first called `fopen()`, followed by `fprintf()` then followed by `fclose()` is done with the following query:

```
FIND src:function CONTAINS $X = fopen()
FOLLOWED BY fprintf($X)
FOLLOWED BY fclose($X)
```

**Figure 4.24** Find all functions that contain the opening of a file stream, printing to that stream, and closing the same stream.

4.2.6 WITHIN

**WITHIN** is an operator that provides a relationship between a FIND, CONTAINS or FOLLOWED BY and the context in which it occurs. The ancestor can reach outside of the search context specified by FIND and does not change the search context’s scoping to operators like FOLLOWED BY.

Like FIND, WITHIN implicitly accepts XPath or source code by first checking of the pattern is an XPath and if not treating it like srcPat. So if one wanted to locate all function definitions within classes within the entire system it can be easily achieved:

```
FIND src:function WITHIN src:class
```

**Figure 4.25** Find all function definitions within a class.
The operator \texttt{WITHIN} can also be applied to more than one operator of a query to specify exactly where to locate specific parts of both partially ordered and unordered expressions. For example, if one were to search for functions that uses a basic C style I/O pattern of \texttt{fopen}, check to make sure stream is open, \texttt{fprintf} and \texttt{fclose}, as seen in the following example,

```c
void outputData() {
    FILE* outputFile = fopen('w');
    if(outputFile) {
        printf(outputFile, "data");
        fclose(outputFile);
    }else{
        return;
    }
}
```

\textbf{Figure 4.26 Example source containing a C-style file access pattern.}

the srcQL query that usage pattern can be located:

\begin{verbatim}
FIND src:function CONTAINS $Z = fopen()
    FOLLOWED BY fprintf()
    WITHIN srcPat if($Z) { }
    FOLLOWED BY fclose($Z)
\end{verbatim}

\textbf{Figure 4.27 Find functions that contain a basic source code usage pattern for C file I/O.}

\texttt{WITHIN} does not change the location of the next \texttt{FOLLOWED BY}, so the call to \texttt{fclose} could be within \texttt{if($X) { }}, after it, or within an else block. It is currently not possible to specify which block, either the \texttt{if} or the \texttt{else} but, the \texttt{fprintf} will occur within at least one of them. In the future using additional operators it will be possible to do so.
4.2.7 WHERE

WHERE is an operator that provides a way to augment the unification process with predicates and functions that operate on the variables of srcPat. The provided predicates include regular expressions, equality, and inequality comparisons, all of which can be applied to variables during unification. This implies that if the user wants to locate a class with a specific naming convention they could use WHERE to do just that by leveraging the match:

```java
FIND class $T { };
WHERE match("^foo[0-9]", $T)
```

**Figure 4.28** Find all classes whose name matches the given regular expression.

This will search for all classes that have a name beginning with the word “foo” followed by a number. The match function provides regular expression syntax to the syntactic variable’s values to allow the programmer to refine their search further. The matching is done on a pretty printed string; this means that if there was an export macro it would be included in the name of $T and would also have to be matched, but it would always be separated by a single space.

WHERE also excepts XPath expressions to be evaluated with the context to which it's attached, either FIND, FROM, CONTAINS, or FOLLOWED BY. This allows for more specific queries about individual sections of the AST to be asserted against when needed. As shown in the following example,

```java
FIND new $T
WHERE not(src:operator[.="new"]/following-sibling::node()[1][.//src:index])
```

**Figure 4.29** Find all uses of operator new where the type is not an array type.
4.2.8 FROM

The FROM operator is designed to allow for nesting of queries. FROM provides a means of using FIND to select complex syntactic variables from a complex query. Allowing for nesting of queries provides a way for filtering operations to be performed on the results of another query or to change the resulting search context to select a sub expression. FROM accepts either a srcPat expression, a srcQL query, or a variable set using the SET operator as it's input.

For example if the user wanted to locate all classes that inherit from a typedef. This can be done using the FROM operator to first select all of the typedefs that occur within the name of the typedef with the inheritance list. FROM allows for variable unification between nested queries using the WHERE operator and nested variable name = Outer query variable name to indicate that a syntactic variable between two queries must be unified.

```plaintext
FIND class $T : $InheritsFrom { };
FROM
  FIND $T CONTAINS typedef $ClsNm $T;
END FROM
WHERE $T = $InheritsFrom
```

Figure 4.30 Find all classes that inherit from a typedef.

Selecting more complicated syntactic variables is impossible without using the FROM operator. FIND allows for syntactic variable selection using FROM, instead of CONTAINS. If we wanted to select all variables names that were matched using the pattern from Figure 4.27 this can be done using FROM as shown in the following example.
FIND $Z
FROM
  FIND src:function
  CONTAINS $Z = fopen()
  FOLLOWED BY fprintf()
  WITHIN srcPat if($Z) { }
  FOLLOWED BY fclose($Z)
END FROM

Figure 4.31 Find the name of all variables matched as part of the nested query.

4.2.9 SET

SET provides a way for queries to save temporary results and use them as part of subsequent queries. SET operations are not returned to the user and cannot be sorted. SET accepts srcQL or srcPat input and stores the result into a given variable. Variables created by SET have their type inferred through use in other parts of the language. For example, if the result of a query is used as part of an IN within the WHERE operator, to check if a syntactic variable exists within the set variable the type of the set variable will be kept internally as an unordered set of strings to allow for faster look up. An example of using SET to in conjunction to check if a syntactic variable's string version exists within a previous queries result follows.

SET $ClassNames FIND $ClsName CONTAINS class $ClsName { }; FIND * CONTAINS new $T WHERE $T IN $ClassNames

Figure 4.32 Find all instances of new $T where the $T type is a class.

4.2.10 UNION, INTERSECTION, MINUS

UNION, INTERSECTION, and MINUS are binary operators whose operands are srcQL queries. Union combines two sets of results, INTERSECTION locates all elements which the queries have in common, and MINUS removes all elements from set
B that exist in set A. All of the set operations follow the same syntactic use pattern as show in the following figure.

```
QueryResultOrvariable SET-OPERATION QueryResultOrvariable
```

**Figure 4.33 Use pattern for binary set operators of srcQL.**

Locating the names of all class and structs is given in the following example.

```
FIND $StrctName CONTAINS struct $ StrctName
UNION
FIND $ClsName CONTAINS class $ClsName
```

**Figure 4.34 Unioning the names of all structs and classes within a system.**

### 4.2.11 Sorting And Grouping

Sorting and grouping of queries is only valid on queries that are returned to the user. This allows for optimizations on other queries that are used as part of other queries but are not directly seen by the user.

### 4.2.12 ORDER BY

**ORDER BY** is an ordering clause that allows for alphanumeric type sorting of the results of the query. It accepts a syntactic variable for which provides a total ordering of all search results returned to the user or a user provided function to provide a total ordering of groups. A query to get all classes in the archive and order them by name is used as an example.

```
FIND class $T { }; ORDER BY $T
```

**Figure 4.35 Find all class definitions and sort them by name.**

The **ORDER BY** can occur in one of two places. The first is to provide a total ordering and the second is after a **GROUP BY** in order to provide a group ordering in the
event that **ORDER BY** is used to provide an ordering for groups with a **Count()** function or a user provided extension function. For example, to order the groups of a **GROUP BY** the largest to smallest:

```sql
FIND class $T { };
GROUP BY $T
ORDER BY Count()
```

**Figure 4.36** Find all classes and group them by name and order those groups by the number of classes per group from largest to smallest.

If a total ordering is not given then the default is to use document ordering. In the event that a group ordering isn’t given then item groups are organized by which occurred first within the document.

### 4.2.13 GROUP BY

**GROUP BY** provides a simple way to gather and sort results from srcQL. By default, srcQL outputs results in document order, i.e., the order they occur in the source code. **GROUP BY** accepts a syntactic variable used in a previous **FIND**. For example, to sort by function name, the query is:

```sql
FIND $R $F() { }; GROUP BY $F
```

**Figure 4.37** Find all function definitions and group them into overloads.

All of the functions are grouped by function name. To change the sort order of the groups, an **ORDER BY** can be used:

```sql
FIND $R $F() { }; GROUP BY $F ORDER BY Count()
```

**Figure 4.38** Find all function definitions and group them by name, then sort those group by number of items in each group from greatest to smallest.

The groups within the above example refer to all of the unique values of $F. This will sort the groups from largest to smallest before returning them.
In addition to taking a syntactic variable name, the clause also allows for a grouping function, or the keyword `syntacticCategories`. Grouping matches into syntactic categories is used for locating all locations that an expression, statement, or declaration occurs within. A syntactic category for match is defined within srcQL differently for statements, expressions, and type or function declarations. For statements, the category is defined as the search context to the nearest ancestor function, class, struct, union, or translation unit. For expression this is defined as the search context to the nearest ancestor statement, such as if, for, expression statement, declaration, to name a few. For types and function declarations the category is defined as the search context to the nearest ancestor translation unit.

When “GROUP BY syntacticCategories” is used, no other ordering can be given with it. All syntactic categories are ordered by the `Count()` function. The following is an example of how to use the syntactic category ordering of results:

```sql
FIND class $T { };
GROUP BY syntacticCategories
```

Figure 4.39 Sorting all classes by their syntactic category.

The following list is some example syntactic categories that could be returned for class $T { }:
Table 4.1 List of possible syntactic categories of a class and a description of how that could occur within source code

<table>
<thead>
<tr>
<th>Syntactic Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace/block/class</td>
<td>Class defined within a namespace</td>
</tr>
<tr>
<td>namespace/block/namespace/block/class</td>
<td>Class defined within a nested namespace</td>
</tr>
<tr>
<td>struct/block/class</td>
<td>Class defined within the body of a struct</td>
</tr>
<tr>
<td>function/block/class</td>
<td>Class defined within the body of a function</td>
</tr>
</tbody>
</table>
CHAPTER 5

Realization and Implementation

I now present the different implementation strategies and work that's been done thus far in developing srcQL. There were many different strategies used to implement the various parts of srcQL. The first phase was to develop a language called srcQuery, which would later become srcPat.

5.1 srcQuery

srcQuery was the first attempt at a source code query language and it was implemented using C# and worked as part of a visual studio. srcQuery would only accept source code without syntactic variables, when it was first developed. srcQuery would take a piece of source code, use that source code to generate an XPath an evaluate the XPath on the current C++ project loaded in Visual Studio.

While this work well for locating simple or trivial patterns it still lacked the ability to generalize queries. To that end syntactic variables were added. Syntactic variables allowed the user to remove pieces of code and replace variable names or statements with a named wild card. The wild card would essentially match any elements that occurred as part of the tree within a query. This added tremendous power to the language, but we found that it was matching one crucial thing, unification. Unification in this sense would allow two syntactic variables within an expression to be compared and asserted against as a predicate within an XPath expression. While XPath does not allow for back references, using a few implementation tricks I was able to force XPath to record matching elements of syntactic variables and generate a unification function based on the
recorded syntactic variables. The unification function was added to XPath as an extension function which would be added to each pattern as part of a predicate so that during the evaluation of an XPath matches could be unified on the fly.

While srcQuery had the capability of handling source code patterns, it lacked the ability to specify relationships between patterns beyond a following-sibling level of detail. The next phase of development was creating the first draft of the srcQL language.

5.2 srcQL V1.0

The first language prototype for srcQL was developed using C# with the .Net framework, and using the expressions library to compile some part of the language into extension functions. This version of srcQL still converted the query into XPath internally in order to evaluate it. srcQL V1.0 didn't contain a full version of the XPath 1.0 standard but instead simply treated the given XPath expressions as text and allowed the .Net XPath engine to evaluate them. The initial version of srcQL contained all of the same operators as the current version. The where operator of srcQL V1.0 compiled some of the expression into extension functions which could be called by XPath.

There were several short comings of this version of srcQL. Version 1.0 implemented all of the source code relationships by using XPath extension functions. While the liberal use of extension functions was fine, getting XPath to be evaluated in the order needed was extremely difficult due how typical XPath implementations evaluate individual steps. Currently, most XPath implementations evaluate a single step at a time on the entire tree composing a breadth first traversal of a document, rather than evaluating the entire XPath at a single location and than moving on to the next possible
match. By structuring queries a particular way it is possible to force the complete evaluation of an XPath before continuing on to the next possible match for a single pattern. By forcing an XPath to evaluate depth first rather than breadth first I was able to develop a system for collecting queries and their related syntactic variables.

The second and biggest problem with version 1.0 of srcQL was that the unification algorithm for unifying variables between operators, for example between a \texttt{CONTAINS} and \texttt{FOLLOWED BY}, was extremely slow. Because XPath lacks the ability to compute back references to previous states of the evaluation this meant that once all possible matches were collected by srcQL they would then have to be unified and removed if the unification failed. Unification for this version of srcQL was done by building a table of all possible values for syntactic variable within the located scope. This was done because there could be multiple matches to a single \texttt{CONTAINS} but only within a single match. For example, consider the following query:

\begin{verbatim}
FIND src:function CONTAINS $X* $I = new $T FOLLOWED BY delete $I
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{An example of locating a function which news and deletes a variable within the same function in that order.}
\end{figure}

A single function could contain multiple instances that match the pattern \texttt{$X* \ $I = new \ $T} and multiple following patterns that matched \texttt{delete \ $I}. All possible combinations of those two items needed to be included in the unification process. Each variable would then be unified with variables of the same name and removed if they were not the same. If at the end of the process at least one matched remained then that match was added to the results to be returned to the user. While this method of unification
worked it was extremely computationally expensive and multiple occurrences of a
contains clause could cause a dramatic increase in evaluation time.

This version of srcQL also included the first results sorting and syntactic category
evaluation of results. This would allow users seeking to identify possible transformation
locations and context to better understand exactly how many of each special case of a
transformation would be needed in order to modify a particular expression or statement.

5.3 srcQL V2.0

The second version of srcQL was developed in C++ instead of C# and would
again hand off an XPath, this time to libxml2. The reimplementation of the srcQL
language in C++ allowed for a much faster query time because libxml2's XPath
implementation is faster than the one implemented by .Net. This implementation failed
to meet some of the more complicated requirements such as sorting and implementing
precompiled functions for evaluating WHERE operator expressions. This implementation
was abandoned and a new direction was taken to instead compile the entire srcQL query
into binary and run it using LLVM. XPathVM

I developed XPathVM as an XPath implementation to use for compiling an XPath
directly into a binary representation which could be executed over a libxml2 XML
document. XPathVM needed to evaluate XPaths faster than the libxml2 interpreted
version of XPath. XPathVM was also developed in order to provide srcQL with the
XPath grammar so that for operators such as FIND and WITHIN would know the
difference between XPath and a srcPat expressions.
One of the primary design requirements of this language is fast execution time over extremely large XML documents, we decided to look into other possible approaches for evaluating queries. For example, fully compiling the query prior to execution rather than interpreting the query as needed. We decided to use LLVM in order to compile an XPath query to binary. We implemented an XPath 1.0 compiled titled XPathVM which works with the libxml2 DOM (Document Object Model) when evaluating queries.

XPathVM implements most of the XPath1.0 standard, but takes liberties during AST optimizations due to the way that srcPat queries generate an XPath representation, and to verify that making these types of design decisions would lead to faster execution times of individual queries. Both .Net and libxml2 use a pipe and filter like implementation of XPath traversals where each time an step is used it generates a new set of nodes.

Figure 5.2 Steps within an XPath query.

This means that the above query would at minimum generate 4 node sets and at worst 5 node sets depending on the implementations reuse of previously allocated node sets. This is the typical implementation strategy taken by both .Net's and libxml2's XPath implementation. While this is a valid strategy for querying XML documents, we took a different approach and instead of performing a breadth first search with the query, we instead opted for a depth first approach. A depth first approach to XPath evaluation
provided us with greater opportunity to optimize queries based on the context and conversion in which a path within an XPath occurs.

### 5.4 XPathVM Optimizations

XPathVM implements several basic optimizations that allow XPath's to behave the same as other XPath implementations, but to only ever do the minimum amount of work required in order to properly evaluate a query. For example, XPath requires that all queries return node sets in document order. Most axes iterate over elements in document order. There is only a limited number of situations which could result in elements being visited out of document order. Therefore, XPathVM won't sort any of the elements being returned unless it detects a situation in which elements of a document could be visited out of order. The same is true for uniquing elements within a node set.

By iterating with a query in a depth first manner allows the creation of short circuiting paths within a query. A short circuiting path, is a path that only the first element of the result is needed. A path can be short circuited when it's converted to text, a number, a string, or a boolean value. All of these situations can be detected when compiling an XPath expression. This means that a path will stop at the first possible result or fail to locate any matching elements within the query. This saves a massive amount of time because traversals do not have to find every possible match within a node set prior to using a single value from it. A conversion of a path to a boolean is the most common conversion used by the XPath's generated by the initial versions of srcQL. This holds true for new implementations as well.
XPathVM also leverages some of the implementation details of libxml2 in order to avoid large numbers of string and object comparisons for attributes such as the name and namespace of an element. libxml2 optionally uses a string dictionary to avoid constantly reallocating the same element name for every node within a document. XPathVM extracts the pointer from the dictionary and directly in lines the character pointer address into the comparison instruction, thereby avoiding string comparisons for all element names. A similar thing is done for namespace objects. The namespace URI is resolved to the correct namespace object associated with the XML document being queried and a pointer to that object is in lined into a comparison instruction to avoid continuous comparison URI string comparison.

5.5 XPathVM Time Trial Results

In order to judge the speed of our XPath implementation we used a section of approximately 1 million lines of source code from GCC which we ran through srcML to create an XML document. We then compared the amount of time XPathVM took vs. the time libxml2's XPath implementation took on the same XPath in order to see which implementation was faster. After the execution of each test the results from both XPathVM and libxml2 were compared to verify that the correct results were returned and that the results from XPathVM were in the same order as those from libxml2. The tests shown are from the windows execution but the test's were run under both Windows and Linux and similar results were achieved under both environments. Table 5.1 shows the trivial XPath execution timings. The libxml2 XPath implementation provides an interface for compiling a query prior to execution. This interface gives libxml2 the
opportunity to set up a query prior to the execution of an XPath. We used the libxml2
interface for pre-compiling an XPath to determine the best execution time for a query
would be.

The Table 5.1 shows that under most circumstances for trivial documents the
compile time for compiling an XPath expression isn't typically worth doing. With the
exception of expressions involving a union operation, the execution time of all of queries
is faster using XPathVM than libxml2's XPath implementation. One thing that makes the
evaluation time of expressions so fast is the simple fact that LLVM's optimizers are able
to fold the evaluation of constants into a single expression resulting in the removal of
instructions and replacing them with their constant values.

In every case on the larger trees XPathVM is faster than libxml2's XPath
implementation. This evidence was used to proceed with the implementation strategy
that's currently being used to implement srcQL. srcQL is designed to operate on an entire
system rather than on small XML documents.
Table 5.1 Expression evaluation timings. Timings given using wall time rather than user time because user time registers as 0.0s.

<table>
<thead>
<tr>
<th>XPath</th>
<th>XPathVM</th>
<th></th>
<th></th>
<th></th>
<th>libxml2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile Time (s)</td>
<td>Execution Time (s)</td>
<td>Total Time (s)</td>
<td></td>
<td>Compile Time (s)</td>
<td>Execution Time (s)</td>
<td>Total Time (s)</td>
</tr>
<tr>
<td><strong>false() + false()</strong></td>
<td>0.006</td>
<td>0.000003</td>
<td>0.01</td>
<td></td>
<td>0.00006</td>
<td>0.00001</td>
<td>0.00007</td>
</tr>
<tr>
<td><strong>false() = false()</strong></td>
<td>0.005</td>
<td>0.000005</td>
<td>0.006</td>
<td></td>
<td>0.00004</td>
<td>0.00002</td>
<td>0.00006</td>
</tr>
<tr>
<td><strong>false() != false()</strong></td>
<td>0.005</td>
<td>0.000007</td>
<td>0.008</td>
<td></td>
<td>0.00004</td>
<td>0.000008</td>
<td>0.00005</td>
</tr>
<tr>
<td><strong>//src:name/text()</strong></td>
<td>0.089</td>
<td>0.00002</td>
<td>0.09</td>
<td></td>
<td>0.00003</td>
<td>0.00006</td>
<td>0.00009</td>
</tr>
<tr>
<td><strong>string(/t1/text())</strong></td>
<td>0.075</td>
<td>0.000006</td>
<td>0.09</td>
<td></td>
<td>0.00003</td>
<td>0.00003</td>
<td>0.00006</td>
</tr>
<tr>
<td><strong>(/tag)/tag1</strong></td>
<td>0.040</td>
<td>0.00002</td>
<td>0.04</td>
<td></td>
<td>0.00003</td>
<td>0.00003</td>
<td>0.00007</td>
</tr>
<tr>
<td>**//tag</td>
<td>//tag2**</td>
<td>0.020</td>
<td>0.00002</td>
<td>0.02</td>
<td></td>
<td>0.00007</td>
<td>0.00001</td>
</tr>
<tr>
<td>**string(/tag</td>
<td>//tag2)**</td>
<td>0.012</td>
<td>0.00003</td>
<td>0.02</td>
<td></td>
<td>0.00007</td>
<td>0.00003</td>
</tr>
</tbody>
</table>
Table 5.2 Timings run on the first 1,000,000 lines of GCC run through srcML. Timings given in the table below are given using user time rather than wall time because user time is more accurate.

<table>
<thead>
<tr>
<th>XPath</th>
<th>XPathVM</th>
<th></th>
<th>libxml2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile</td>
<td>Execution</td>
<td>Total</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Time (s)</td>
<td>Time (s)</td>
<td>Time (s)</td>
<td>Time (s)</td>
</tr>
<tr>
<td>//src:class</td>
<td>0.0156</td>
<td>0.1875</td>
<td>0.2031</td>
<td>0.2656</td>
</tr>
<tr>
<td>//src:expr_stmt</td>
<td>0.0781</td>
<td>0.1250</td>
<td>0.2031</td>
<td>1.3125</td>
</tr>
<tr>
<td>//src:expr_stmt/</td>
<td>0.0781</td>
<td>0.4375</td>
<td>0.5156</td>
<td>0.8594</td>
</tr>
<tr>
<td>src:expr/src:call</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>//src:expr_stmt/</td>
<td>0.1094</td>
<td>0.4219</td>
<td>0.5313</td>
<td>1.7813</td>
</tr>
<tr>
<td>src:expr[src:call]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>//src:call/</td>
<td>0.1250</td>
<td>0.4531</td>
<td>0.5781</td>
<td>1.4063</td>
</tr>
<tr>
<td>src:name[.=&quot;malloc&quot;]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>//src:expr_stmt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>//src:decl_stmt</td>
<td>0.0938</td>
<td>2.4219</td>
<td>2.5156</td>
<td>48.7031</td>
</tr>
<tr>
<td>//src:name/text()</td>
<td>0.0625</td>
<td>0.5469</td>
<td>0.6094</td>
<td>100.0000</td>
</tr>
<tr>
<td>//src:expr/src:name/</td>
<td>0.0781</td>
<td>0.4219</td>
<td>0.5000</td>
<td>1.5156</td>
</tr>
<tr>
<td>src:name[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((//src:name)/src:name)</td>
<td>0.0468</td>
<td>0.5000</td>
<td>0.5469</td>
<td>48.5000</td>
</tr>
</tbody>
</table>
5.6 srcQL V3.0

The final version of srcQL uses the grammar from XPathVM, and LLVM in order to full compile and evaluate the entire query. srcQL V3.0 alters the evaluation of a typical XPath to a depth first model instead of a breadth first model, thereby allowing back references and scoping to be implemented without the need for hacking it into an extension function, like the previous versions of srcQL. The final version of srcQL also uses a different model for unification, in that at the end of each possible match rather than at the end of the entire query. This allows the unification occur for each possible match without having to regenerate every possible match after the query is completed. If a particular piece of a variable fails to unify then srcQL continues on to the next possible match within the given search context. If a match is located than the algorithm short circuits, the syntactic variable and the search context are recorded and part of the algorithm short circuits skipping the remaining possible matches. We found that fully compiling a simple query allows for us to more easily evaluate queries far beyond what's allowed by XPath.

5.6.1 srcQLV3.0 Initial Time Trial Results

In order to assure that srcQL would evaluate queries fast enough we conducted a preliminary time trial on the first one thousand C header and source code files of GCC, which is just over 1.2 million lines of source code. While this isn't the largest portion of the project it serves as a good benchmark, and it fits easily within the memory of most modern day computers. We ran queries which were within the capabilities of the system at the time this was written the queries, the time it took to evaluate the queries is
presented. Each query execution consists of three phases, parsing, code generation, and evaluation. The parsing phase is the time it takes to turn a given query into a compilable AST. Code generation is the time it takes to generate and optimize LLVM byte code. Finally the evaluation is the time it takes to execute a query on the document. The total time and the number of results located are also presented.

Table 5.3 srcQL Query evaluation timings. All timing results are given in seconds.

<table>
<thead>
<tr>
<th>Query</th>
<th>Parsing (S)</th>
<th>Code Gen (S)</th>
<th>Eval (S)</th>
<th>Total (s)</th>
<th>Number of Results Located</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND src:expr_stmt</td>
<td>0.02</td>
<td>0.14</td>
<td>105.1</td>
<td>105.25</td>
<td>195,848</td>
</tr>
<tr>
<td>FIND src A + B</td>
<td>0.05</td>
<td>0.75</td>
<td>0.47</td>
<td>1.27</td>
<td>0</td>
</tr>
<tr>
<td>FIND src:for[ancestor::src:for]</td>
<td>0.02</td>
<td>0.20</td>
<td>0.84</td>
<td>1.06</td>
<td>1,828</td>
</tr>
<tr>
<td>FIND ST i FROM FIND src:for END FROM</td>
<td>0.03</td>
<td>0.50</td>
<td>6.98</td>
<td>7.52</td>
<td>1007</td>
</tr>
<tr>
<td>FIND new $T</td>
<td>0.02</td>
<td>0.22</td>
<td>0.39</td>
<td>0.63</td>
<td>9</td>
</tr>
<tr>
<td>FIND src:call[ancestor::src:condition ]/parent::src:if]</td>
<td>0.00</td>
<td>0.23</td>
<td>59.86</td>
<td>60.09</td>
<td>83,223</td>
</tr>
</tbody>
</table>

Given the preliminary time trial we can see that in the preliminary trials that srcQL does scale quite well to handle a million lines of source code. The study shows that for simple traversals with no around 2 thousand matches we can locate those matches in around 1 second.
srcQL Internal Optimization and Execution Strategy

In order to make srcQL run fast on very large XML trees several optimization strategies and techniques are employed. The details of the techniques are presented here. There are many angles one can approach optimization from when it comes to program optimization. The main focus of optimization in the case of srcQL focuses algorithms and runtime, while leaving the LLVM Intermediate Representation optimization to LLVM. This chapter presents the computational model that governs the code generation of srcQL, the internal algorithm optimizations used for optimizing traversals, and the optimizations that are used with LLVM IR.

5.7 Computational Model

The computational model used internally by srcQL is one that's based off of a push down automata (or PDA). A more terse description of the PDA is presented in 0 of APPENDIX B. The computational model is one that is based on the evaluation of a predicate whose output is based on the following predicate or is said to be true if there is no following predicate.
The computational stack shown in Figure 5.3 depicts the basics of the internal srcQL computational model. In this instance the computational model of query evaluation does not represent an entire program, it represents a single query from within the program. The distinction between an entire program which can be viewed as a query in and of itself, and a srcQL query is that a srcQL query is either a single srcPat expression or starts with the FIND operator. Breaking the computation of the traversal down into a PDA allows for short circuiting during query evaluation because the compiler know what element is being selected and which elements are simply being evaluated as true or false. In doing this I am able to prevent multi-visitation of elements and reduce the overall number of visits needed in order to compute a match.

Let us consider the simple srcPat expression new int. While this may look benign in nature it is more complex than one would anticipate. Because of the simple nature of the query it does have an XPath representation.
//src:expr[
  src:operator[.="new"]
 ][
  src:name[.="int"]
 |  
  src:call[
    src:name[.="int"]
  ]
 ]

**Figure 5.4 The XPath representation used for the srcPat pattern new int.**

In Figure 5.4 we see that there is one clear element being selected and many elements being asserted against as predicates within XPath. Each XPath predicate does not have a 1 to 1 mapping into srcQL often it has a one too many mapping. This is to make computational easier to extend internally. The main difference between a typical XPath representation and the internal representation of srcQL is that typically a step has a traversal along an axis, a node test, and a set of predicates, our representation removes the concept of a single step and transformed the traversal, node test, and predicate set into a set of predicates within the internal representation.

The construction and evaluation of the srcQL Internal Representation (srcQL IR) is done through the evaluation of a predicate list. Each predicate takes a context which is defined as the current element being evaluated, and each predicate which modifies or moves a context produces a new context as input to the next predicate. The context is handled in this way to emulate the computational stack. Due to how the input and output contexts of predicates are handled every predicate knows exactly which element it was operating on when popping predicates off of the stack.

There are many types of predicates, each has one has its own use. The most basic types of predicate are move, expression, collection, and pure context changes. All
predicates have an input and output context, which is the current element within a
traversal, and a next predicate. The output context of a predicate is the input context to
the next predicate in the list. If there is no next predicate in the list than the predicate
considered to be true.

Move predicates are those which take an input context and produce an output
context at new location, and provided the ability to optionally short circuit when true. A
short circuit on true means that if the next predicate evaluates to true no additional moves
are computed and the current predicate evaluates to true as well.

Expression predicates evaluate an expression against the input context which
results in either true or false. In the event that the expression is true the next predicate is
pushed on to the computational stack otherwise false is returned to the previous predicate.
Expressions predicates may also contain sub queries, thereby containing more complex
predicate chains. An example of where an expression predicate may need a sub query
would be when there are two possible paths that we need to match, or two possible names
for a single element. While this sort of predicate does not exist as part of an XPath, it
was added as an optimization to srcQL IR because it allows for steps to select elements
within multiple different names at the same point during a query. Consider the query in
Figure 5.4, it contains a union in the second predicate of the expression. While this is
how one would write this sort of query in XPath this can be simplified within the srcQL
IR. For example, a subquery allows for predicates like that to be condensed into a single
step rather than a union. A basic representation is show in the following diagram.
Collection predicates are a type of predicate which reference backwards in the computational stack to the move predicate selected as the element to collect as the result of the query, and places that element into a collection. Any query which result in a node set contains a collection predicate. The collected element is determined by srcQL during compilation and can be computed as the last move type predicate which does not short circuit.

Pure contest change predicates are those which reference backwards in the computation stack to a move predicate and modify the context to be a copy of a context previously output by a move predicate. This type of predicate allows us to maintain stack like behavior during computation without backtracking through recursion in order to continue evaluation.

The srcQL IR computational chain is depicted in Figure 5.6. The computational chain executes from top to bottom and when a predicate evaluates to true it's push onto the stack.
Figure 5.6 The srcQL IR chain representation. This depicts the traversal taken by srcQL during the evaluation of the srcPat expression. The Follow-sib is short for following-sibling which maps directly to the following sibling axis of XPath.
We now show the process that's used in order to fully evaluate the srcQL Query
new int. Consider the following source code:

```c
int main() {
    int * i = new int;
    double* d = new double(6);
    int* j = new int(5);
    return 0;
}
```

**Figure 5.7 Source code to search for the pattern new int within.**

The source code available within Figure 5.7 contains two matches for the pattern
new int. The first on line 2 and the second on like 4. The first step towards evaluating
this using srcQL is to run the source code through srcML.
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src"
     xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5"
     language="C++" filename="ExampleSourceCode.cpp">
  <function><type><name>int</name></type><name>main</name><parameter_list>()</parameter_list>
  <block>{
    <decl_stmt><decl><type><name>int</name><modifier>*</modifier></type><name>i</name><init>=
      <expr><operator>new</operator><name>int</name></expr></init></decl>;</decl_stmt>

    <decl_stmt><decl><type><name>double</name><modifier>*</modifier></type><name>d</name><init>=
      <expr><operator>new</operator><call><name>double</name><argument_list>(<argument><expr><literal type="number">6</literal></expr></argument>)</call></expr></init></decl>;</decl_stmt>

    <decl_stmt><decl><type><name>int</name><modifier>*</modifier></type><name>j</name><init>=
      <expr><operator>new</operator><call><name>int</name><argument_list>(<argument><expr><literal type="number">5</literal></expr></argument>)</call></expr></init></decl>;</decl_stmt>

    <return><expr><literal type="number">0</literal></expr></return>
  }<block></function>
</unit>

Figure 5.8 Source code example after being run through srcML.
Now that we have both the computational model and the source code the next step is to push the first move predicate on the stack and give the XML document object to the srcQL query as input.

Figure 5.9 Initial starting state for computational traversal.

The descendant predicate traversal visits every element in document order and every element is rejected by the one of the two following predicates until an expr element is reached. In this example the first expr element located is shown in bold Figure 5.10.

```xml
<init>= <expr><operator>new</operator> <name>int</name></expr></init>
```

Figure 5.10 The first located src:expr element within the srcML document.

The computational stack now contains two expression predicates and the descendant predicate as shown in Figure 5.11. This means that the next piece of the traversal can be added to the computational stack which in this case is the Child Predicate.
Figure 5.11 Computational stack after locating the first <expr> element within the document.

After adding the child traversal, the stack contains the following.

![Computational Stack](image)

Figure 5.12 The stack after adding the child traversal.

Now that the child move predicate has been added to the stack we traversal all child elements of the input context which in this case is the expr element. This yields the following state once the correct element is located.

![Computational Stack](image)

Figure 5.13 The stack state after the operator element is located.
Once the operator is located the expression predicate all of the child elements are examined to see if they are text and they have the content "new". Once a string with the value of new has been located the context must be changed in order to attempt to continue the child traversal of the expr element. Once we have the next sibling element of operator we begin a following-sibling traversal in order to locate one of two possible predicates within the multiple sub query expression predicate. The current reference state of the stack between the XML document is show in Figure 5.14.

![Computational Stack](image)

**Figure 5.14** The reference structure between the XML tree representation and the current state of the computational stack of srcQL. The line colors only serve to visually differentiate lines between the stack and tree.

As can be seen from the arrows above the number of reference between the tree and the AST can get extremely complicated. This is only a simple example and it's becoming difficult to visually depict. In subsequent diagrams many of the reference lines have been omitted in order to make this easier to comprehend.
Now that the following sibling traversal visits the whitespace text node within the tree, which does not match any of sub queries within the sub query expression. After which it visits the name element of the tree which matches the first possible sub query and yields a true result the sub query expression and the namespace and name match. After which the computational stack has the following state.

![srcML XML Tree Representation]

**Figure 5.15** The state of the computational stack after evaluating the first sub query within sub query expression predicate which yielded a match of the name element.

The next and final piece of the traversal is testing the text element int for the proper text value of int. After the evaluation of the text node we change the context back to the starting expression element, and collect that context and store it into a container.
The next step within the traversal is to go back to the collection traversal because the end of the predicate chain was reached and in order to locate additional matches we need to simply skip all remaining elements within all short circuiting traversals when removing them from the stacks and continue traversing once top element of the stack is a move predicate that does not short circuit.

The computational stack after a short circuiting operation leaves only the initial descendant predicate on the stack which continue its traversal until it reaches another expr element within the tree. All srcQL queries are handled in this manner but slightly more complex in most cases.

While the automata are shown as a stack they are not implemented as a stack during LLVM code generation. Due to the nature of the srcQL language and the fact that we know all of the different possible paths through any given srcQL query binary code to

---

**Figure 5.16 Collecting the matched context from on the computational stack.**
execute the query is generated as a set of nested loops, if statements, and in push back operations, in the case of collection operations.

5.8 Traversal Algorithms

The two most important traversal algorithms are the descendant and child traversals because they are used the most often during code generation. One of the main issues with the traversal algorithms is that contain so many branches, especially the descendants algorithm. I have found that the traversal algorithm for descendants can be reduced down to the following:

```java
descendantTraversal(inputContext) {
    currentNode = inputContext
    goto DESCENDING
    VISIT
    VisitNext Predicate:
    DESCENDING
    if currentNode has children
        currentNode = currentNode.firstChild
        goto VISIT
    STRAIF
    if currentNode has next siblings
        if currentNode is an element
            builder.endElement(currentNode)
            currentNode = currentNode.nextSibling
            goto visit
        ASCENDING
        if currentNode is an element
            builder.endElement(currentNode)
        if currentNode == node
            goto EXIT
        else
            currentNode = currentNode.parent
            goto STRAIF
```
5.9 LLVM Intermediate Representation Optimization

The existing LLVM optimizers are leveraged rather than writing specialized ones to optimize the LLVM generated by srcQL. The same optimizations as clang does under level 3 optimizations with the exception of disabling the math optimizations and loop vectorization are used. The math optimizations are disabled because srcQL does not do any heavy weight mathematic so enabling them is a waste of compilation time.
CHAPTER 6

Language to Language Comparison

There are many different aspects that make up a good query language and a good programming language in general. We comprised research questions, presented in the following section which help us best identify different aspects of individual query languages, their abilities and how we can compare and contrast them with srcQL.

6.1 Research Questions

We conducted this experiment with the goal of answering the following questions.

RQ1. Do query language that operate on source code require explicit AST level interaction with a fully compiled AST or can this be handled using only pieces of source code?

RQ2. Is it possible to create a query language using only the syntax of the programming language, without requiring full compilation of the source code?

RQ3. Can we locate all of the different categories in which a particular pattern occurs to aid the developer in writing of complicated transformations?

RQ4. Is srcQL expressive enough to support developer needs?

RQ5. Is srcQL scalable and usable on large system?

The research questions here provide a means of best evaluating the different aspects of a query language. Our experiment is designed to best compare and contrast the difficulties or complexities associated with writing queries and provide a means of
comparing the overall complexities associated with composing a query. We devised and experiment, presented in the following section, which will assist in helping us in answering research question one, two, and four. The answers to the remaining research questions are presented as well.

6.2 Language to Language Comparison

A language to language comparison with existing source code query language technology and srcQL is now presented. The design of this experiment is as follows, for each language I attempt to answer a set of queries. Not every query language works one every programming language, because of this each query presented is given in a language it supports. One question has multiple forms one for C, C++, and Java, the question is answered based on the tools supported languages. Each query is compared for clarity with srcQL based on the overall length of the query. The only languages that are included within this study are those which meet the following criteria:

- Enough of the language must exist in order to compose a query.
  - While this may seem subjective, the tangible requirements of this are that some or all of the following artifacts be available in some form, either within the current paper, or available online.
    - The Language grammar
    - A language standard
    - A Tutorial
    - Examples
    - A comprehensive description of language features
The language must not be a library.

- Our research is only concerned with domain specific languages for querying source code.
- If the query tool is explicitly interacted with using a general purpose language such as JAVA, C++, or C then it's not a query language it's a library tool.
- An example of a library rather than a query language would be Clang's AST Matching library. While the AST Matching library does indeed match pieces of source code, the terms in which it's is done through C/C++ rather than being an actual language.

The queries were developed from several different angles, each designed to test the expressiveness of the language as well as the ability of the language to adapt to differing ways that developers view source code. The queries cover four different views of the source code.

1. The pure AST view. When the relationships between elements of the source code are expressed in purely in terms of the AST. While the argument could be made that all source code queries can be made in terms of the AST, and that's not wrong, the actual query is expressed in terms of the relationships between those elements which isn't expressible as a statement or explicitly as source code.

2. Statement view. When the relationship or entities are expressed in terms of different statements of executable source code.
3. The type system view. When the relationships are those between classes, structures or unions within the type system. This view of source code is how developers see classes relationships within a system.

4. Variable View. The variable view point is interested in locating variables, while this may seem similar to the statement view point, it's significantly different. The variable view is different from the statement view because the statement view point is only interested in relationships between actions, while the variable view is interested in the subject of those actions.

After taking the differing points of view into account we composed queries that would in some way fit into one or more point of view and showcase the different aspects of each query language. The queries were than constructed to test the following query language features to test query language capabilities. The nature of our research has led us to focus primarily on finding different pieces of source code rather than using a query language to compute things about a specific piece of source code.
### Table 6.1 List of test queries to implement in srcQL and other query language in order to compare and

<table>
<thead>
<tr>
<th>#</th>
<th>Test Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Find all Expression Statements.</td>
</tr>
<tr>
<td>2</td>
<td>Find all expressions which contain the expression A + B.</td>
</tr>
<tr>
<td>3</td>
<td>Find all for loops which are inside of another for loop.</td>
</tr>
<tr>
<td>4</td>
<td>Find all calls to function foo().</td>
</tr>
<tr>
<td>5</td>
<td>Find the variable i declared within a for loop.</td>
</tr>
<tr>
<td>6</td>
<td>Find all uses of operator new.</td>
</tr>
<tr>
<td>7</td>
<td>Find all functions with calls to foo(A) and bar(A) where A is any variable name.</td>
</tr>
<tr>
<td>8</td>
<td>Find all classes which start with a capital letter Q.</td>
</tr>
<tr>
<td>9</td>
<td>Find all classes which declare the following functions foo(), bar(), foobar(), fizz().</td>
</tr>
<tr>
<td>10</td>
<td>Find all function calls within the condition of an if statement.</td>
</tr>
<tr>
<td>11</td>
<td>Find all functions that contain a call to fopen() followed by a call to fclose().</td>
</tr>
<tr>
<td>12*</td>
<td>Find all functions that sets a variable using a call to malloc followed by freeing the same variable.</td>
</tr>
<tr>
<td></td>
<td>C++) Find all functions that call new and delete the same variable.</td>
</tr>
<tr>
<td></td>
<td>Java) Find all functions that contain a variable set using new followed by setting that variable to NULL.</td>
</tr>
<tr>
<td>13</td>
<td>Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.</td>
</tr>
</tbody>
</table>
Question 12 is selected based on supported language. If a query language supports C++, then the query is composed using C++. Not all query languages support querying on C++, so if the language supports Java the Java query is composed, and otherwise the query for C is chosen.

The following languages were selected as part of the study: srcQL, RASCAL, Coccinelle, JQuery, PMD, SCRUPLE, CodeQuest, Aria, and srcML. The reason that some other languages that are cited in this paper are left out is because no information exists on the language, the tool was vapor-ware, or because the language was simply a mathematical representation which isn't a Domain Specific Language for querying source code. One missing exemption from the example language is TXL and it was omitted because it would look only slightly different than RASCAL, but only because the grammars maybe defined differently.

### 6.3 srcQL

Question 1: Find all Expression Statements.

```sql
FIND src:expr_stmt
```

**Figure 6.1** srcQL: Locate all of the expression statements. This query simply searches on an XPath for all AST nodes with the default srcML namespace given as src and the element name expr_stmt.

Question 2: Find all expressions which contain the expression A + B.

```sql
A + B
```

**Figure 6.2** srcQL: Locating the expression A + B. This is done within srcQL by using the srcPat only query. srcPat knows that if there is no trailing ; on an expression or declaration to treat the expression as a partial expression rather than a statement.
Question 3: Find all for loops which are inside of another for loop.

```
FIND src:for WITHIN src:for
```

Figure 6.3 srcQL: Locate all for loops which are within another for loop. Search for all for loops using an XPath notation with an ancestor relationship with another for loop given by the WITHIN operator, which is also done using XPath notation.

Question 4: Find all calls to function foo().

```
FIND foo()
```

Figure 6.4 srcQL: Locate all calls to foo. One would think this query may be matched as an XPath expression because it meets the XPath grammar notation for function calls, but this works because XPath doesn't define a function foo so this is treated as srcPat instead. Due to the closed world principal that srcQL and srcPat operate under, this actually matches foo with any number of parameters.

Question 5: Find the variable i declared within a for loop.

```
FIND $T i FROM FIND src:for END FROM
```

Figure 6.5 srcQL: Find all of the variable declarations from within the set of for loops. The FIND operator provides a query which matches any variable declaration with the name i and searches all of the for loops for the variable declaration pattern which declares i.

Question 6: Find all uses of operator new.

```
FIND new $T
```

Figure 6.6 srcQL: Find all instances of the expression new $T. This searches for all calls to operator new.

Question 7: Find all functions with calls to foo(A) and bar(A) where A is any variable name.

```
FIND src:function CONTAINS foo($A) CONTAINS bar($A)
```

Figure 6.7 srcQL: Find all functions which contain calls to both foo and bar, and have the same variable name for their first parameter.
Question 8: Find all classes which start with a capital letter Q.

```sql
FIND class $T { };
WHERE match(/^ Q\s/); $T
```

Figure 6.8 srcQL: Locate all classes with a name that starts with a capital letter Q. srcQL handles this by allowing for the application of a regular expression to the normalized string values of syntactic variables.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

```sql
FIND class $T { };
CONTAINS $X foo();
CONTAINS $Y bar();
CONTAINS $Z foobar();
CONTAINS $A fizz();
```

Figure 6.9 srcQL: Locating all classes with any name which contain the function declarations of foo, bar, foobar, and fizz in any order.

Question 10: Find all function calls within the condition of an if statement.

```sql
FIND src:call WITHIN src:condition/parent::src:if
```

Figure 6.10 srcQL: Locate a call node with the ancestor that is condition which has parent that is an if stmt.

Question 11: Find all functions that contain a call to fopen() followed by a call to fclose().

```sql
FIND src:function CONTAINS fopen() FOLLOWED BY fclose()
```

Figure 6.11 srcQL: Locate all functions which contain calls to fopen followed by fclose. The srcQL followed by operator provides a relative association between two expressions while still being within the context of a function.
Question 12: Find all functions that call new and delete the same variable.

\begin{verbatim}
FIND src:function
   CONTAINS $I = \text{new} \ $T
   FOLLOWED BY delete $I
\end{verbatim}

Figure 6.12 srcQL: Find all instances of the variable $I being assigned to and being deleted, in relative order within the context of a single function.

Question 13: Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.

\begin{verbatim}
FIND $V \text{ FROM FIND } \text{F($V)} \text{ WITHIN src if($V) \{ \} END FROM}
\end{verbatim}

Figure 6.13 srcQL: Select the syntactic variable $V from within a sub-query given using the FROM operator, where the variable $V exists as parameter to a function as well as being used within the condition of an if statement. Due to the way that unification works within srcQL this may also match expressions that occur in both locations as well as variables.

6.4 RASCAL

The RASCAL language is a parser generator framework that's used to define parser for multiple languages. The matches handled by RASCAL are done through a type constructor, matched against the grammar.

Question 1: Find all Expression Statements.

\begin{verbatim}
(statement) `\langle Expression \ exp>\;`
\end{verbatim}

Figure 6.14 RASCAL: The Java grammar defined by RASCAL does not have a an explicit concept of an expression statement.

Due to how the Java grammar was defined by RASCAL it does not contain an explicit expression statement concept.

Question 2: Find all expressions which contain the expression A + B.

\begin{verbatim}
(statement) `A+B`
\end{verbatim}

Figure 6.15 RASCAL: Finding the expression A + B.
Question 3: Find all for loops which are inside of another for loop.

`for <Expression expr> <Block blk> for <Expression expr> <Block blk2>`

Figure 6.16 RASCAL: Locating a for loop within another for loop.

Question 4: Find all calls to function foo().

`(statement) `foo();``

Figure 6.17 RASCAL: Finding all expression statements to foo. This does not locate foo in all possible locations only when it is used as part of an expression statement.

Question 5: Find the variable i declared within a for loop.

`for (<Type tp> i; <Expression expr>; <Expression expr>)<Block stmt>`

Figure 6.18 RASCAL: Finding all of the for loops which contain a variable with the name i.
Question 6: Find all uses of operator new.

```
`<Blockstm* pre>
<Type t> <ID v> = new <Type t> <Expr exp>
<Blockstm* post>

<Blockstm* pre>
<Type t> <ID v>;
<ID v> = new <Type t> <Expr exp>
<Blockstm* post>

if(<Type t> <ID v> = new <Type t> <Expr exp>){} 
if(<ID v> = new <Type t> <Expr exp>){} 
while(<Type t> <ID v> = new <Type t> <Expr exp>){} 
while(<ID v> = new <Type t> <Expr exp>){} 
do{ }while(<ID v> = new <Type t> <Expr exp>);
for(<ID v> = new <Type t> <Expr exp>;;){}
for(<Type t> <ID v> = new <Type t> <Expr exp>;;){}
for(;;<ID v> = new <Type t> <Expr exp>;;){}
for(;;<ID v> = new <Type t> <Expr exp>){}
for(;;<ID v> = new <Type t> <Expr exp>){}
```

Figure 6.19 RASCAL: Locating all expressions that contain call to operator new, when using RASCAL one must enumerate all possible locations within the grammar that an expression could occur and match ALL of them in order to locate part of an expression. When it comes to matching specific parts of an expression, declaration, or statement rules in which that statement could occur within the grammar must be tested for.
Question 7: Find all functions with calls to foo(A) and bar(A) where A is any variable name.

In order to match every case within the grammar one must iterate through all of possible syntactic categories within the grammar in order to match every possible case where this could occur.

```
`foo(<Id var>) bar(<Id var>)`
```

Figure 6.20 RASCAL: Finding all calls to both foo and bar with the same variable.

Question 8: Find all classes which start with a capital letter Q.

```
`class <name:^Q>`
```

Figure 6.21 RASCAL: Find all classes that start with a capital letter Q.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

```
`<Class c>{
  <BlockStm* pre>
    for(stat <- p.body){
      if((Expression) `foo()` := stat){
        //found foo
    } else if((Expression) `bar()` := stat){
        //found bar
    } else if((Expression) `foobar()` := stat){
        //found foobar
    } else if((Expression) `fizz ()` := stat){
        //found bar
    }
  <BlockStm* post>
  }
``
```

Figure 6.22 RASCAL: Find all classes which implement foo, bar, foobar, and fizz.
Question 10: Find all function calls within the condition of an if statement.

```
`if(<Expression>())<Block stm>`
```

Figure 6.23 RASCAL: Find all if statements with a call within the condition.

Question 11: Find all functions that contain a call to fopen() followed by a call to fclose().

```
`<Function>(<arg_list lst>)
<Blockstm* pre>
fopen()
<Blockstm* mid>
fclose()
<Blockstm* post>`
```

Figure 6.24 RASCAL: Find all functions with calls to fopen followed by fclose.

Question 12: Find all functions that call new and delete the same variable.

```
//case with split decl
`<Function>(<arg_list lst>){
<Blockstm* pre>
<Id type> <Id var>;
<Id var> = new <Expression expr>
<Blockstm* post>
}`

//case with non split decl
`<Function>(<arg_list lst>){
<Blockstm* pre>
<Id type> <Id var> = new <Expression expr>
<Blockstm* post>
}`
```

Figure 6.25 RASCAL: Finding all of the variables that are either set using new or declared and assigned using new and set to null.
Question 13: Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.

```plaintext
if(<Expression exppre*> <Id var> <Expression* exppost>){
<Blockstm* pre>
<Id var>
<Blockstm* post>
}
```

**Figure 6.26 RASCAL: Finding all variables used within the condition of an if statement and within the block of the if statement.**

6.5 Coccinelle

Coccinelle is a diff based approach to program transformation. Because this language was originally designed to be a transformation language, but we are only interested in the pattern matching aspects, in some examples the code to transform the pattern into has been omitted. There are several caveats to some of the query representations, in that it isn't possible to correctly select internal portions returned as part of the result of another query, those cases are noted and a pattern is given to make the selection anyway.

Question 1: Find all Expression Statements.

```plaintext
@@
expression e;
@@
e;
```

**Figure 6.27 Coccinelle: Locate all expressions e that are part of a statement.**
Question 2: Find all expressions which contain the expression $A + B$.

```
A + B;
```

**Figure 6.28 Coccinelle:** Locate the expression statement $A + B$. There are caveats with this query due to the limitation of the language.

There is no real way to search for part of an expression within any context using Coccinelle. Unless that expression happens to be a function call, but matching an operator expression is outside of the Coccinelle language.

Question 3: Find all for loops which are inside of another for loop.

```
for(;;;;) {
  for(;;;;) {
  }
}
```

**Figure 6.29 Coccinelle:** Matching nested for loops. The ... within the pattern indicates that this pattern which matches everything.

Question 4: Find all calls to function foo().

```
foo();
```

**Figure 6.30 Coccinelle:** Match all calls to foo as an expression statement.
Question 5: Find the variable i declared within a for loop.

```
@@
type T;
identifier i
@@
(
  for(T i;...;...) {
  }
  |
  for(...;T i;...) {
  }
  |
  for(...;...;T i) {
  }
  |
  for(...;...;...) {
    T i;
  }
)
```

Figure 6.31 Coccinelle: Locating for loops with the variable named i within them. This is done using multiple different patterns because there is not means of specifying relationships between two statements other than parent child relationships and in a for loop there is multiple locations in which that pattern could occur.

Question 6: Find all uses of operator new

Coccinelle only works on the C programming language and does not support matching patterns containing C++ functionality.
Question 7: Find all functions with calls to foo(A) and bar(A) where A is any variable name.

```c
@r1 exists@
type RetType;
identifier FuncName;
identifier VarName;
@@
{
    RetType FuncName(...) {
        foo(VarName);
        ...
        bar(VarName);
    }
    |
    RetType FuncName(...) {
        bar(VarName);
        ...
        foo(VarName);
    }
}
```

**Figure 6.32 Coccinelle: Locate all functions containing calls to both foo and bar and both with the same variable.**

Question 8: Find all classes which start with a capital letter Q.

Coccinelle does not support C++ functionality only C.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

Coccinelle does not support C++ functionality only C.

Question 10: Find all function calls within the condition of an if statement.

```c
@@
identifier CallName;
identifier VarName;
@@
if(CallName(...)) {
```
Figure 6.33 Coccinelle: Locate all function calls within an if statement.

Question 11: Find all functions that contain a call to fopen() followed by a call to fclose().

```c
@r1 exists@
type RetType;
identifier FuncName;
identifier VarName, VarName2;
@@
RetType FuncName(...) {
    fopen(VarName);
    ...
    fclose(VarName2);
}
```

Figure 6.34 Coccinelle: Locate all functions with a call to both fopen and close in that order.

Question 12: Find all functions that sets a variable using a call to malloc followed by freeing the same variable.

```c
@r1 exists@
type RetType;
identifier FuncName;
identifier VarName;
@@
RetType FuncName(...) {
    VarName = malloc(...);
    ...
    free(VarName);
}
```

Figure 6.35 Coccinelle: Find functions that contain calls to both malloc and free of the same variable.

Question 13: Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.
Coccinelle cannot select individual variables within an expression but one could locate all functions that contain those variables of that type, but the problem is that we'd have to know how the variable was being used as part of the if statement in order to properly compose an accurate query. So the given query only selects the matched expression but doesn't select the individual variables.

```cpp
@@
identifier FuncName;
identifier CallName;
identifier VarName;
@@
if(VarName) {
    ...
    CallName(VarName);
    ...
}
```

**Figure 6.36 Coccinelle: Locating all if statements which contain a variable and contain a call containing that variable within the matched if statement.**

This would only match the case were the variable itself is tested for null but not if it was a parameter to a function call, declared inside of the condition to the if statement or if it's not the first parameter to the function CallName.

6.6 JQuery

The JQuery language only operates at the type system level of the language representation. JQuery views it's queries as those over models of the software without looking into function definitions. As such the following research questions cannot be answered using by JQuery, question one through question seven, and question ten through question 13. The remaining questions, eight and nine, which can be answered by JQuery are provided.
The JQuery language is one that's based on first order predicate calculus available through a DataLog-esque interface. JQuery provides a means of selecting types using a simple predicate and applying additional predicate operations depending on the relationships used.

Question 8: Find all classes which start with a capital letter Q.

\[
\text{type(?T), re_name(?T, "^Q.*") }
\]

Figure 6.37 JQuery: Find all types with a name that matches the regular expression requiring that the name start with a capital letter Q.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

\[
\begin{align*}
\text{type(?T),} \\
\text{method(?T, ?M1), name(?M1, "foo"),} \\
\text{method(?T, ?M2), name(?M2, "bar"),} \\
\text{method(?T, ?M3), name(?M3, "foobar"),} \\
\text{method(?T, ?M4), name(?M4, "fizz")}
\end{align*}
\]

Figure 6.38 JQuery: Find types which have methods with the name foo, bar, foobar, and fizz.

6.7 PMD

The PMD system is used for enforcing rules for enforcing style guidelines, coding standards, or preventing simple coding errors. PMD provides two ways for users to implement their rules, the first is through an XPath notation which operates over the AST of a Java program, the second is through implementing a Java extension. In this research we only focus on the first aspect provided through XPath. Due to the limitations of XPath some or many of the queries cannot be directly implemented using XPath but instead require Java implementation in order to match more complicated patterns. Because of the limitations of every question after question six, with the exception of
question ten, cannot be implemented. An explanation of why the query cannot be composed is given instead.

Question 1: Find all Expression Statements.

```
//StatementExpression
```

**Figure 6.39 PMD: Find all expression statements.**

Question 2: Find all expressions which contain the expression A + B.

```
//Expression[
  AddativeExpression[
    PrimaryExpression[
      Name[
        @Image="A"
      ]
    ]
  ][
    PrimaryExpression[
      Name[
        @Image="B"
      ]
    ]
  ]
]
```

**Figure 6.40 PMD: Find the expression A + B.**

Question 3: Find all for loops which are inside of another for loop.

```
//ForStatement[ancestor::ForStatement]
```

**Figure 6.41 PMD: Find all for statements which are within other for statements.**

Question 4: Find all calls to function foo().

```
//PrimaryExpression[PrimaryPrefix[Name[@Image="foo"]]]
```

**Figure 6.42 PMD: Find all calls to function foo.**

Question 5: Find the variable i declared within a for loop.

```
//ForStatement//LocalVariableDeclaration[
  VariableDeclarator[
    VariableDeclaratorId[
      
    ]
  ]
]
```
Figure 6.43 PMD: Find all variable declarations of a variable with the name which occur within a for loop.

Question 6: Find all uses of operator new

//Expression[./AllocationPrefix]

Figure 6.44 PMD: Find all expressions which contain the use of the AllocationPrefix of operator new.

Question 7: Find all functions with calls to foo(A) and bar(A) where A is any variable name.

This cannot be implemented using only XPath because XPath does not support back references or unification.

Question 8: Find all classes which start with a capital letter Q.

There is a limitation within PMD which prevents access to the name of a class, or interface when using XPath. So this query must be implemented using a rule extension to PMD.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

There is a limitation within PMD which prevents XPath from accessing method names. This query must be implemented using a Java rule extension to PMD.

Question 10: Find all function calls within the condition of an if statement.

//IfStatement/Expression/PrimaryExpression//PrimaryPrefix

Figure 6.45 PMD: Find all of the function calls within the condition of an if statement.
Question 11: Find all functions that contain a call to fopen() followed by a call to fclose().

It is impossible to implement a scoping operation within native XPath without the aid of complex extension functions and extremely altering the way in which a query is evaluated. This query must be implemented using a Java rule extension to PMD.

Question 12: Find all functions that contain a variable set using new followed by setting that variable to NULL.

XPath cannot implement back references or variable unification. This query must be implemented using a Java rule extension to PMD.

Question 13: Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.

This query cannot be implemented using XPath, because XPath does not provide a means of back referencing or unification which is required in order to implement this query. This query must be implemented using a Java rule extension to PMD.

6.8 SCRUPLE

A pattern matching regular expression like language that is used as a pattern matching language that operates on C. This language doesn't have a true concept of an expression, it stops at the statement level of granularity, meaning that parts of a statement cannot be located separately only the entire statement. Because the language was specifically designed the C programming language, and an older version of the language at that the structure of the queries given is given to reflect that. The version of C
SCRUPLE was created to work on required that all declarations be given at the beginning of a function block which is assumed in the queries presented.

Question 1: Find all Expression Statements.

The concept of an expression statement does not exist within the SCRUPLE language. The closest concept to matching all expression statements is the #_{alphanum}+ which will match any single expression. For example: #_1 Could match any expression statement, but not any sub expressions.

Question 2: Find all expressions which contain the expression A + B.

Matching this as an expression is not possible due to the structure of SRCUPLE. However, locating the statement where A + B is assigned to some variable is possible. The following is an example of matching the statement C = A + B, which is given instead.

```
C = A + B;
```

Figure 6.46 SCRUPLE: Match the assignment statement C = A + B.; SCRUPLE does not provide a means of matching partial expressions only statements.

Question 3: Find all for loops which are inside of another for loop.

```
for(#;#;#) @[FOR]
```

Figure 6.47 SCRUPLE: Match all for statements within other for statements.

Question 4: Find all calls to function foo().

```
foo();
```

Figure 6.48 SCRUPLE: Match all calls to foo, with no parameters. This only matches at the statement level and not the expression level.

Question 5: Find the variable i declared within a for loop.
This cannot be done in SCRUPLE because no variables can be declared in for loops in the version of C it was designed for.

Question 6: Find all uses of operator new.

SCRUPLE does not support operator new, it only works on C.

Question 7: Find all functions with calls to foo(A) and bar(A) where A is any variable name.

```c
$v_A = "***"

%%
@f_1() {
  $*
  @*
  @{* 
    @{*
      @{* 
        @{@{
          @{* 
            foo($v_A);
            *}
          @* 
          @{*
            bar($v_A);
            *}
        *}
      @* 
    }
  } 
}

$v_A = "***"

%%
@f_1() {
  $*
  @*
  @{* 
    @{*
      @{@{
        @{* 
          bar($v_A);
          *}
      @* 
      @{*
        foo($v_A); 
```
Figure 6.49 SCRUPLE: Find functions containing a call to both foo and bar in any order. SCRUPLE has a strict notion of ordering so in order to do this one must query for both possible orderings.

Question 8: Find all classes which start with a capital letter Q.

SCRUPLE only works on C and does not provide a means of inspecting the type declaration system so this is impossible to write to work on structs either.

Question 9: Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

SCRUPLE does not support classes.

Question 10: Find all function calls within the condition of an if statement.

```c
if (@f_1()) {
    @*
}
```

Figure 6.50 SCRUPLE: find all function calls within the condition of an if statement. SCRUPLE does not provide a means for narrowing specific selections but this pattern will locate all if statements with a single function call within the condition.

Question 11: Find all functions that contain a call to fopen() followed by a call to fclose().

```c
@f_1() {
    @*
    @*
    @*{
        @*
        fopen($v_A);
        *
    }
    @*
    @*
    @*
```
fclose($v_A);
 *}
 */
}

Figure 6.51 SCRUPLE: Locating all functions that contain a call to both fopen and fclose on the same variable within the same function.

Question 12: Find all functions that sets a variable using a call to malloc followed by freeing the same variable.

@f_1() {
 $*
 @*
 @{*  
 @{*  
 $v_A = malloc(sizeof($t_1));  
 *}
 @*  
 @{*  
 fclose($v_A);
 *}  
 @*}

Figure 6.52 SCRUPLE: Find the allocation and free of the same variable within a function.

Question 13: Find all variables which are used in a condition of an if-stmt, and then used in a call inside the if-stmt.

This cannot be done accurately by SCRUPLE, because there is no way to see if a variable exists or is used within an expression within this language. The closest one could get is to check if the variable was used in a function with a single parameter which is inside of an if statement where the conditional expression is accurately single variable.
Also this language does not allow the user to return specific variables only locate places in the source code such as functions, or statements.

```plaintext
if($v_1) {
    @f_1($v_1);
}
```

**Figure 6.53** SCRUPLE: Due to the limitations of the SCRUPLE language one cannot directly query for variable names, the only thing that can be done is to match the location and allow the user to visually select the desired variables instead.

### 6.9 CodeQuest

Due to the fact that no other information exists other than a two page paper that only contains two examples. This makes it impossible to reproduces the given queries accurately. In order to re-create the given queries a few assumptions have been made about the underlying structure of the data within datalog. The assumption made is that based upon the data given, that CodeQuest does not support queries on statements and shares the same structure as JQuery. This means that CodeQuest shares the same limitations as JQuery.

**Question 8:** Find all classes which start with a capital letter Q.

```prolog
type(?T) re_name(T, "^Q.*")
```

**Figure 6.54** CodeQuest: Locating a type with a name starting with a capital Q.

**Question 9:** Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

```prolog
type(?T)
method(T, M1) name(M1,"foo")
method(T, M2) name(M2,"bar")
method(T, M3) name(M3,"foobar")
method(T, M4) name(M4,"fizz")
```
Figure 6.55 CodeQuest: Find types which have methods with the name foo, bar, foobar, and fizz.

6.10 Aria

The Aria query language was designed in order to locate particular pieces of source code and generate test cases for those functions matched. Aria is a transformation like language in that it's queries have actions associated with them, in order to generate test cases. The actions have been omitted from the given queries because we are only interested in matching patterns, and not generation of test cases.

The Aria language leaves very little information in the way of language structure and documentation. This is due to the project stagnation combined with the age of the project. Because of the language of additional documentation only some of the queries could be composed. Only those queries which had enough available information are provided. The main component missing from the Aria language is the AST over which it operates.

Question 1: Find all Expression Statements.

```
ROOTPROC FindExprStmt

PROC FindExprStmt
ROOT CPPFile;
{
    [(? ExprStmt (PRINT stdout "Located expression statement at %s" $location)]
}
```

Figure 6.56 Aria: Find all expression statements and print them.

Question 2: Find all expressions which contain the expression A + B.

The way that Aria handles operators is from a compiler prospective, in that all operators undergo overload resolution to a specific function. Because the operators
undergo overload resolution there is no way to generically handle operators. The only way to interact with operators is at a functional level, and because this removes the specific name associated with an operator in favor of the mangled name created by the compiler. There is also no discussion of how to interact with built in operators.

Question 3: Find all for loops which are inside of another for loop.

```cpp
ROOTPROC FindNestedLoops
PROC FindNestedLoops
ROOT CPPFile;
{
    [(? ForLoop
       (IF (HAS-TYPE $parent ForLoop)
          (THEN (PRINT stdout "For loop %s defined at %s" $token $location))
       )]
}
```

**Figure 6.57 Aria: Find all nested for loops.**

Question 4: Find all calls to function foo().

```cpp
ROOTPROC FindFoo
PROC FindFoo
ROOT CPPFile;
{
    [(?FunCall
       <callname
       (IF (EQUAL-TOK $token "foo")
          (THEN
              (PRINT stdout "Function Call %s at %s" $token $location)
          )]
    )]
}
```

**Figure 6.58 Aria: Locate all calls to the function foo.**
The remaining queries cannot be implemented because of a lack of information.

### 6.11 srcML and XPath

The srcML tool does not have a custom query language, it instead leverages XPath and native XML tools for interacting with the XML archives it creates. The queries presented here are those which can be implemented using only native XPath functionality. In order to complete all queries additional extension functions or interactions from a general purpose programming language would be necessary. The queries which cannot be completed are omitted. Similar to srcQL the src namespace prefix is provided as a short hand for the srcML default namespace.

**Question 1:** Find all Expression Statements.

```xml
//src:expr_stmt
```

*Figure 6.59 srcML: Find all expression statements.*

**Question 2:** Find all expressions which contain the expression A + B.

```xml
//src:expr[
  src:name[.= "A"][
    following-sibling::node()[1][
      self::src:operator[.="+"]
    ]following-sibling::node()[1][
      self::src:name[.="B"]
    ]
  ]
]
```

*Figure 6.60 srcML: Find an expression containing A + B. There is a limitation to expressing this within XPath and that is that this is actually an expression that contains the variable A, an operator +, and the variable B, in that respective order.*

Finding the expression A + B, where A, +, and B are directly next to one another is difficult due to the inability of XPath to specify anything beyond a relative ordering of
sibling elements instead of limiting it to being the explicit next element. Some of the limitation is due to how srcML handles expression markup. srcML expression markup doesn't produce the same tree based relationships that one would expect within a fully compiled language. While a compiler produces an AST in which operator precedence is handled and a full tree is produced based on that, srcML simply creates a list of elements within an expression. The end result is that an expression within srcML consists of a list rather than a tree representation. The tree representation contains variables, operators, and function calls as child elements. The query above handles this by limiting following siblings to being the first following sibling element rather than simply looking for a following sibling.

Question 3: Find all for loops which are inside of another for loop.

```xml
//src:for[ancestor::src:for]
```

**Figure 6.61 srcML: Find all for loops with an ancestor that is also a for loop.**

Question 4: Find all calls to function foo().

```xml
//src:call[src:name = "foo"]
```

**Figure 6.62 srcML: Find all calls to the function with name foo.**

Question 5: Find the variable i declared within a for loop.

```xml
//src:for/src:decl[src:name = i]
```

**Figure 6.63 srcML: Find all declarations of a variable named I which occur within a for loop.**

This query is written in an optimized manner in that there are typically fewer for loops than there are variable declarations, therefore limiting the search space to the space within for loops will reduce the amount of time needed to search all child elements of all for loops. The alternative to this is to use the ancestor axis to check if any variable
declarations are within for loops. While both approaches are valid the one presented is faster.

**Question 6:** Find all uses of operator new.

```xml
//src:expr[src:operator = "new"]
```

*Figure 6.64 srcML: Find all expressions containing a use of operator new.*

**Question 9:** Find all classes which declare the following functions foo(), bar(), foobar(), fizz().

```xml
//src:class
  [.///src:function_decl[src:name = "foo"]]
  [.///src:function_decl[src:name = "bar"]]
  [.///src:function_decl[src:name = "foobar"]]
  [.///src:function_decl[src:name = "fizz"]]
```

*Figure 6.65 srcML: Find all functions that contain function declarations of foo, bar, foobar, and fizz.*

**Question 10:** Find all function calls within the condition of an if statement.

```xml
//src:if/src:condition/src:call
```

*Figure 6.66 srcML: Find all function calls within the condition of an if statement.*

The remaining queries cannot be implemented using purely XPath, additional XPath extension functions would be needed in order use XPath to complete the queries.

### 6.12 Findings and Discussion

We have presented a set of query languages and a set of queries completed by those languages. We now compare and contrast the different abilities of each language in order evaluate our research questions.
Table 6.2 Queries by language and a check mark indicates that the language was able to create a query which satisfied the query.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcQL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RASCAL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Coccinelle</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>JQuery</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PMD</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SCRUPLE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CodeQuest</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Aria</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>srcML</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

As we can see from Table 6.2 some language are not capable of completing the all queries, while others are capable of completing all queries. The language to language comparison can now be used to answer whether srcQL is as expressive other language querying technologies? The answer is, yes. In using srcQL, we show that leveraging only the syntax of the language one can comprise queries expressive enough to keep up with other languages such as RASCAL. We now present the answers to each research question.

**RQ1** Do query language that operate on source code require explicit AST level interaction with a fully compiled AST or can this be handled using only pieces of source code? The short answer is yes. srcQL provides a srcPat which allows users to leverage
pieces of source code, and not just at the statement level. srcPat provides the ability to match parts of expressions all without the need for the user to even know there is syntactic markup going on behind the scenes. Some language provided the ability to match anything through type constructors given within the grammar, srcQL and srcPat remove the need for users to look at the grammar. While some queries are simply shorter when using an AST based notation, which in the case of srcQL is akin to using XPath, any source code to source code relationship can be written using srcPat instead. The only time that one needs to directly interact with the AST is if the relationship between elements cannot be directly expressed using source code because it is a relationship between elements of the AST instead of being a relationship between two elements of source code.

RQ2 Is it possible to create a query language using only the syntax of the programming language, without requiring full compilation of the source code? srcQL does exactly this, srcQL does not require compilation or even compilable source code in order to compose or evaluate queries. While some query languages are actually a compiler framework such as TXL, and RASCAL, srcQL is not a compiler nor does it seek to replace one. srcQL leverages srcML, which is a tool that generates a markup for the syntax of the language. srcQL uses this markup in order to evaluate queries and because srcML does not require compilable source code, that means that neither does srcQL.

RQ3 Can we locate all of the different categories in which a particular pattern occurs to aid the developer in writing of complicated transformations? srcQL provides a
means of source code into their syntactic categories, or the categories in which they occur. While other languages may provide a means of getting this information from the AST srcQL simply use the AST from srcML to do this upon request. This feature allows transformation writers to locate all possible locations where a transformation could occur, and thereby compose transformations for only those frequently occurring categories, and manually update infrequent ones.

**RQ4** Is srcQL expressive enough to support developer needs? We have shown that in comparison to other query language srcQL is as expressive as other pattern matching tools when it comes to locating specific patterns within source code through our experiment. Our experiment shows that some languages are unable to express the same queries as srcQL. srcQL has shown that in its pattern matching ability is on par with existing technologies all without the need for type constructors or full compilation.

**RQ5** Is srcQL scalable and usable on large system? By leveraging the technology from within XPathVM srcQL is able to speed through even complex queries on extremely large systems. The main limiting factor in the ability to compute a query fast is being able to store the XML tree in memory. We have found that it usually takes longer to load the source code into memory than it does to evaluate a srcQL query on that code.
CHAPTER 7

Conclusions

In this dissertation, srcQL, a source code query language is presented. The language structure, implementation, prior work, and comparison to other languages is discussed in detail. Further, a study between multiple other languages which could be used for querying source code was presented. The contributions of this work are as follows.

In CHAPTER 2, we present a background and related work, within the domain of source code querying languages. We discuss the different approaches which are used by current tools. CHAPTER 3 presents previous work that lead up to the development of the srcQL language and the implementation strategies used within. We also present the XPathVM tool which was leveraged in order to compose a better understanding of exactly how LLVM can be used within a domain specific language in order to obtain even faster runtimes.

CHAPTER 4 presents a detailed description of the srcQL and srcPat language features. We show the full breadth and depth of the srcQL language. We layout the implementation, optimization, and execution strategies in of srcQL in CHAPTER 5. We present the way that queries are handled by srcQL internally, as well as present a detailed description of how that structure functions during the runtime of a query relative to the XML tree being queried.

CHAPTER 6 shows a language to language comparison between srcQL and the other querying technologies, and presents the difference in capabilities between the two.
The comparison also shows that of the languages studied only two of the languages studied were able to complete all of the given queries.

The srcQL language is still under development and is constantly improving. The future work of this project is constantly growing, in order to support new and or different ways that developers look at or understand source code. Everyone views source code in a slightly different way and we are always trying to accommodate different ways of selecting different pieces and parts of queries as providing more complex ways of specifying or relating different pieces of source code together.

7.1 Future Work

There is still more work to be done when it comes to using srcQL. A more complete language specification needs to be developed so that we can identify any missing or parts of the language which need to further fleshed out in order to provide users with a more extensive set of relationships between different aspects of source code. In the immediate future of srcQL to develop a tool called srcForm which will be comprised of two languages srcQL and srcTL [Newman, Bartman, Collard, Maletic 2017]. srcQL will provide the mechanism for locating transformation sites while the srcTL language will provide the transformation.

srcQL is a simple query language, it may in the future, be beneficial to further extend the capabilities of the srcQL language. So that it is capable of directly computing metrics within the language and go beyond querying. Other language such as TXL, and RASCAL both provide support for more general purpose operations, and in order to
provide more complex means of searching and even better results handling srcQL would need similar capabilities.

srcQL leverages srcML as the AST that it operates on. srcML provides a suite of other tools that annotate the XML in order to provide additional information such as type or slice information. Annotating elements such as functions or variables with type or def-use chain information would greatly extend the abilities of srcQL, to the point where it would be similar to working on a compiled AST. In order for srcQL to take full advantage of this srcQL would need to be extended so that it could interact seamlessly with these sorts of extensions. Currently srcQL can only interact with srcML annotations through language attributes while using an XPath representation of a query or through the WHERE operator of the language. It would be much easier if these sorts of annotations had internal language support, rather than simply interacting with XML through a predicate expression.

Determining if a source code query language is going to be able to be used by simple end users is still an open question and one that is not addressed in this paper. In order to determine how good a query language is one can only truly look at the capabilities of the language, and not as much at how quickly developers are able to learn a language, or how well developers are capable of composing queries. One of the definite avenues of research that is still available is to do a user study looking to see which query language developers find most useful, and easiest to use.

We have shown that srcQL can keep up with other languages, all while using less information then them when it comes to matching patterns in source code. srcQL is as
capable as other languages without the need for full access to an AST, and all while being able to tackle large systems.
APPENDIX A

srcQL Language Grammar

This section describes the srcQL language in detail. XPath grammar has been omitted from the srcQL grammar for brevity and replace with the rule title XPathExpr. For a reference to the XPath grammar please see for more information. The grammar is given in EBNF format.

```
start := srcQLProgram | srcOnly;
srcOnly := +anySourceCodeToken;
srcQLProgram := assignedVariableList mainQuery;
assignedVariableList := setVariable*;

setVariable := "SET" variableName
             (srcQLSubQuery | explicitXPath | srcUpToKw);

srcQLSubQuery := findOp
                (("INTERSECTION" | "MINUS" | "UNION") srcQLSubQuery)?;

mainQuery := srcQLSubQuery orderingOperators?;

findOp := findGuessContext
       | findImplicitFromContext
       | findVarWithinFrom
       | findVariableWithinContains
       | findXPathOrSrc

findGuessContext := "FIND" ( "/" | "/" | "*" | "." )
containsNoFollowedBy;

containsNoFollowedBy := "CONTAINS"
                        (explicitSrc | explicitXPath | srcUpToKw) withinOp? whereOp?;

findImplicitFromContext := "FIND" ("*" | ".")
```
findXPathOrSrc := "FIND" (explicitXPath | explicitSrc | XPath | srcUpToKw)

findVarWithinFrom := "FIND" logicalVariable

findVariableWithinContains := "FIND" logicalVariable containsNoFollowedBy;

containsList := containsOp*;

containsOp := "CONTAINS" (explicitXPath | explicitSrc | srcUpToKw)

followedByOp := "FOLLOWED BY" (explicitXPath | explicitSrc | srcUpToKw)

fromOp := "FROM" (srcQLSubQuery_expr | explicitSrc) "END FROM";

withinOp := "WITHIN" (explicitXPath | explicitSrc | XPath | srcUpToKw);

whereOp := "WHERE" expr;

orderingOperators := groupByOp | orderByOp | orderGroupsBy;

groupByOp := "GROUP BY"
   (functionCall | logicalVariable | "syntacticCategories");

orderByOp := "ORDER BY" (logicalVariable | functionCall);

orderGroups := groupByOp orderByOp;

srcUpToKw = /*
  Source code up to but not including the next keyword in srcQL
 */;

explicitSrc := "src" srcUpToKw;

explicitXPath := "XPath" XPath;

anySourceCodeToken := numberLiteral
 | stringLiteral
 | logicalVariable
 | /* Any other source code token */;

stringLiteral := "\"" character* "\"";
numberLiteral := /* any numeric literal floating point or integer */;

variableName := /* a C/C++ identifier */;

logicalVariable := "$" variableName;

expr := logicalOr;

logicalOr := logicalAnd ("||" logicalAnd)?;

logicalAnd := equalityOps ("&&" equalityOps)?;

equalityOps := comparisonOps ("==" | "!=") comparisonOps)?;

comparisonOps := additiveOps ("<" | ">" | "<=" | ">=") additiveOps)?;

additiveOps := multDivMod ("+" | "-") multDivMod)?;

multDivMod := uniaryNegation ("*" | "/" | "<<" uniaryNegation)?;

uniaryNegation := "-"? primaryExpr;

primaryExpr = numberLiteral
  | logicalVariableRef
  | setVariableRef
  | stringLiteral
  | functionCall
  | ("(" expr ")")
  | XPath

setVariableRef := logicalVariable;

logicalVariableRef := logicalVariable;

functionCall = cppIdentifier "(" parameterList? ")";

parameterList := expr ("," expr)*;
APPENDIX B

A srcQL Language Specification

Language Examples

The following is a basic set of examples with a general description of what they match within source code.

new int - Find all of the instances of an expression containing a call to operator new with type int. This includes arrays of int's as well as int(), as well as calls to nothrow or inline new as well.

new $T - Find all expressions that contain operator new with any type or call. The $T represents a place holder known as a syntactic variable.

$T* $Var = new $T - Find all declarations statements which are initialized with new and have the same declared type as the type given with new.

void $T() { } - Find all function definitions that return void and have zero or more parameters.

**FIND** new $T - Find all expressions containing operator new. This is equivalent to the second example without FIND.

**FIND** $R $F(){ } - Find all function definitions with any return type, any name, and any number of parameters.

**FIND** src:function - Find all functions. XPath notation.

**FIND** * CONTAINS $R $F(){ } - Find all function definitions with any return type, any name, and any number of parameters.

**FIND** $T CONTAINS new $T - Find all types used with operator new.
**FIND** src A * B - Find all declarations of A* with the name B. This shows disambiguation between XPath and srcPat which could be treated as XPath if not disambiguated.

**FIND** src:function CONTAINS $var = new $T CONTAINS delete $var - Find all functions that contain expressions that new a variable and delete a variable with the same name.

**FIND** src:function CONTAINS $X = fopen() FOLLOWED BY fprintf($X) FOLLOWED BY fclose($X)- Find all functions that use a stream access pattern of open stream print to stream and close stream in that order.

**FIND** src:function WITHIN src:class - Find all functions defined within the body of a class.

**FIND** class $T { }; WHERE match("^ foo[0-9]"$, $T) - Find all classes with the name foo and zero through nine.

**FIND** class $T : $InheritsFrom { }; FROM FIND $T CONTAINS typedef $ClsNm $T; END FROM WHERE $T = $InheritsFrom - Find all classes that inherit from a typedef.

**FIND** class $T { }; GROUP BY $T ORDER BY Count() - Find all classes group them by name, and sort those groups according to the number of elements within them.
srcQL Structure

The srcQL language is actually made up of three languages: XPath, srcPat, and srcQL. The language is broken down into three pieces because of how it was developed and the different problem each piece of the language is solving. While XPath is capable of specifying simple srcPat expressions the complications that arise from writing those patterns by hand is not suited for an average user of XPath. srcPat was invented to solve that problem it allows for patterns of source code to be used for matching patterns which would otherwise be hundreds if not thousands of characters long in XPath representation. While srcPat solves the issue of having to write languages by hand it creates another problem, we don't have a means of relating these patterns together while searching.

srcQL provides a means of specifying relationships between patterns of srcPat and XPath in terms of source code. The relationships provided allow the user to use XPath where they can't specify a query in terms of source code relationships and provide a means for locating trivial pieces of the AST such as all functions or all if statements.

While XPath doesn't solve all of the different ways one can interact with the AST srcQL does provide a way for the user to interact with specific patterns through syntactic variables.

srcQL: Source code Query Language

Types

srcQL types are as follows:

- Int
  - An signed integer number 64 bits in length.
- **Number**
  - A double wide floating point number.
- **String**
  - A null terminated string of characters this is the same as a c-style-string.
- **Boolean**
  - A true or false value.
- **Char**
  - A single character 8 bits in width. The UTF-8 encoding of XPath is upheld by srcQL.
- **Node Set**
  - A set of unique set of nodes from within the XML DOM.
- **Syntactic Variable**
  - A pair of references to sibling elements within the XML DOM. The first element is the start of elements matched from a syntactic variable and the second is either another element or null of all sibling elements are to be included in the match.
- **String Set**
  - A set of unordered strings.
- **Syntactic Variable List**
  - A list of syntactic variables in any order.
- **Match**
A match consists of a root element or of a matching pattern within srcQL, srcPat, or XPath expression and a map of syntactic variables names to syntactic variable list.

- **Match Set**
  - A set of match types ordered by the root element's document order.

- **Set Of Node Sets**
  - A set of node sets types

- **Set of Match Sets**
  - A set of match sets.

- **XML DOM Object types** (see XML DOM specification for additional information on xml node types and their relationship [Recommendation 1998])
  - **xmlDoc**
    - An XML document.
  - **xmlNode**
    - A single element within the XML DOM. All parts of the XML tree are considered to be xmlNodes
  - **xmlNs**
    - An XML namespace element within the XML DOM.
  - **xmlElement**
    - An XML element
  - **xmlProp**
    - An XML property within an XML DOM.
An XML text node within the XML DOM.

There is a complex relationship between several of the different types in this section in that all of the XML related types are convertible into an xmlNode type. This subclass type relationship allows for any type from within an XML document to be easily handled through a single interface until down cast into the appropriate type.

*Mapping into XPath representation*

srcQL is a super set of XPath, because of this, all of the XPath types exist within the srcQL language. This includes all of the XML node types listed within the XML DOM specification. all srcQL does is provide an extension to the XML type system.

*Internal Representation*

The internal representation is used in order to make it significantly easier explain and present the srcQL language elements and how they map to specific semantics of the language. This representation allows mappings between complex comparisons to be handled more easily and presented in a way which allows others to easily understand how the syntax and semantics of the language are implemented.

The internal representation for srcQL is constructed to be a push down automata(PDA) where each automata models an operation on an input which results in either true or false, in the case of true the next automata is entered on to the stack and in the case of false you return to the previous element on the stack. When no remaining
automata are to be pushed on to the stack the expression evaluate to true and each automata is popped off of the stack until one is reached that continues its evaluation.

This representation is used to normalize all of the different parts of the srcQL language include the entire XPath language and srcPat. This allows for optimizations and analysis within srcQL to be done without the need to deal with a large number of different Abstract Syntax Trees (AST) from multiple different languages.

The internal representation of srcQL of several high level types of elements, these make up the base part of the srcQL internal representation:

- srcQLProgram
- srcQLQuery
- Predicate
- Expression

**srcQLProgram**

A srcQLProgram is the root of all srcQL query programs. It contains the following:

- External Parameters
  - Inputs set external to the program which are of one of the types presented in
- Starting Element
  - An xmlNode which is the input parameter to the first predicate of the PDA and given as an input to the program.
- Pre-query
o An ordered set of srcQLQueries to be evaluated prior to the main query. The result of which may be used in subsequent operations.

o The input to a pre-query operation is always the

- Main Query
  
o The query for which the result is the result of the program.
  
o post processing of the result of a query shall take place during the post-query operations

- Post-query
  
o An ordered set of operations, which operate on the result of the main query producing the final result of the program.

A srcQLProgram shall be evaluated in the following order.

1. Pre-Query operations
2. Main Query
3. Post-Query operations

The result type of evaluating a srcQLProgram is the result type of the main query after all post query operations have been applied to it. The result of applying operations to the result of the main query can result in an alteration of type depending on the operation preformed. So the result type of a srcQLProgram f is the result of the main query m transformed to m' through the application of post function P where P(m) -> m' for each P.
srcQLQuery

The srcQLQuery is the starting element to a chain of predicates. The list of predicates represents a list of automata which when evaluated yield a result of a given type and a Boolean value indicating if the predicate chain reached the end of all predicates in the chain.

A srcQLQuery contains the following:

- Input Node
  - The starting location of a PDA evaluation

- Result
  - The result determines a few things, 1) the type of the output returned to the user, 2) the behavior of how the results of a query are collected, and 3) what happens when a PDA evaluates the final predicate as true.
  - Set of valid result types of a srcQLQuery:
    - Result of normal query evaluation
      - Boolean
      - Number
      - String
      - Node set
      - Match set
    - Only valid as the result of a function evaluated within the PDA.
      - See the section on expressions for more details
      - Set of Match Sets
• Set of Node Sets
• Set of Strings
• int

  o No other types shall be returned as the result of evaluating a query.

• Predicates

  o Shall be a list of predicates comprising a PDA evaluation.

A srcQLQuery shall be represented as a function Q which takes two parameters, the input context of the program C, the xmlNode for which to begin the evaluation of a query I, and R the result of the query. The evaluation of a query is formalized as $Q(C, I, R) \rightarrow R', B$ where $R'$ is the result of evaluating PDA and B is the result of evaluating all predicates.

The evaluation of a srcQLQuery is the result of $P(C, I, R) \rightarrow R', B$ where P is the first a predicate in the PDA.

*Predicates*

A predicate within srcQL is a single automata that performs and operation on an input and passes that onto the next predicate. All predicates evaluate some function against the current input context given to them and provide an output context. The function that's used to evaluate against the current input context is called the predicate evaluation function.

All predicates contain the following:

• Input Context
o An input context is the xml context element referenced as part of the XPath 1.0 standard. This shall be the current element yielded from the evaluation of a previous predicate or the starting element from a srcQLQuery

- Input Principal Type
  o The principal type associated with the input context refers to a valid base class of an xmlNode type. The principal node type comes from the XPath 1.0 standard, and indicates which operations are valid when used on a given input context.

- Input Index
  o The input index is the number associated with an XPath step traversal. This is used to refer to the Nth node in a traversal or step.
    o Initial value is 1 when being evaluated as the first predicate from a srcQLQuery.

- Output Context
  o Shall the result of applying a transformation to the input context.
  o The output context of a predicate is the input context to the next predicate

- Output Principal Type
  o If the result of evaluating the predicate evaluation function yields a different xmlNode type then the principal node type is used to show that part of the output.

- Output Index
The index of the current element within a traversal.

- Next Predicate
  - If there is no next predicate than if the evaluation of predicate considered to be true than the result of evaluating a null next predicate true.

Predicate visitation of the Next Predicate is different for each type of predicate in that some predicates have a specific visitation point during the predicate evaluation function at which to invoke the next predicate, while others simply perform a visitation after predicate evaluation function. This distinction is presented in the following sections.

Types of predicates are enumerated in the following subsection.

Types of predicates

There are several predicate subtypes. All predicates fall in to one of the following categories: Action, Move, Expression. Action Predicates are those which modify the program context or return value. We break predicates down in the following manner because they the share common traits and behaviors.

Action Predicate

An action predicate modifies either the program input context or the return value. Formalized as \( P(C, I, R) \rightarrow C', I, R \) or \( P(C, I, R) \rightarrow C, I, R' \). These types of action predicates allow for collection of the program or return value without altering the input or output context's of a predicate operation. Action predicates always produce a Boolean
return value based on the Next Predicate. This means that it always evaluates to true if it doesn't have a Next Predicate.

For all action predicates the predicate evaluation function is invoked after the predicate evaluation function.

CollectMatchPredicate is an action predicate which given a reference to a move predicate modifies the result to contain the output context of the move predicate.

The CollectMatchPredicate P has the following signature P(C, I, R, M) where M a previous move predicate, and applying P(C, I, R, M) ->C, I, R', where R contains M.OutputContext.

The collection of matches is an important operation. Determining the move context to from which to locate a context to collect is handled by locating the last move predicate in the current list of predicates which does not short circuit. Short circuiting is discussed more in the section regarding move predicates.

Move Predicate

A Move Predicate is a predicate which takes the input context and creates a new output context at a given location. The location may be the same as the input context depending on the predicate evaluation function. A move predicate is a means of generalizing the of a traversal through an XML tree, or through a set of elements. Movement through the set of elements is the result of applying the predicate evaluation function on the output context of the predicate until it returns true. If no additional nodes within the traversal exist and the next predicate is evaluates to false the result of a move predicate shall be false.
Each move predicate has additional attributes: short circuiting, and visitation point. A short circuiting traversal is a traversal which when the Next Predicate evaluates to true no longer traverses additional elements through repeated application of the predicate evaluation function but instead simply returns true. Short circuiting represents an optimization that not all query evaluations require the creation of a node set or match set in order to have behave correctly with respect to query evaluations. The rule for short circuiting is that if the result of a traversal would ever need to be returned to the user of srcQL, given as a parameter to a function, or if it's used as an intermediate result, for example as the result of a pre query or filtering operations.

The visitation point is the place within a traversal in which Next Predicate is evaluated.

A move predicate is formalized in the following manner. Move predicate P, P(C, I, R) -> C, I', R. Where I' is the output context of the move predicate.

The follow is a list of different types of move predicates within srcQL which are used to support the XPath axes:

- AncestorPredicate
  - Visit each ancestor of the input context.
- AncestorOrSelfPredicate
  - Visit input context and each ancestor of the input context.
- AttributePredicate
  - Visit all attributes of the input context.
- ChildPredicate
- Visit all children of the input context.

- **FollowingPredicate**
  - Visit all following nodes of the input context in document order.

- **DescendantPredicate**
  - Visit all descendant elements of the input context.

- **DescendantOrSelfPredicate**
  - Visit the input context and all descendant elements.

- **FollowingSiblingPredicate**
  - Visit all following siblings of the input context.

- **GoToRootPredicate**
  - Visit the xmlDoc of the input context.

- **PrecedingPredicate**
  - Visit all elements of the document which occurred in document order before a given element.

- **PrecedingSiblingPredicate**
  - Visit all siblings that occurred prior to the input context.

- **SelfPredicate**
  - Visit the input context.

- **ParentPredicate**
  - Visit the parent element of the input context.
The predicates which map into XPath directly can best be summarized in the following diagram:

There are additional predicates that are used in order to extend this representation beyond XPath and to fully support all of the needs of srcQL.

- **ScopedFollowedByPredicate**
  - The ScopedFollowedByPredicate performs the same traversal as a following predicate with the exception of taking a stopping point as additional input to the traversal.
  - The stopping point stops the traversal at a given point known as the search context. The search context is explained later on.

- **ReverseFollowingSiblingsPredicate**
Using the input context of the predicate locate the last sibling and visit all
siblings until the input context is reached.

The ReverseFollowingSiblingsPredicate provides an optimization for
locating the last occurrence of a pattern within a list of siblings. This is
only used with SelectChildrenUpTo and SelectSiblingsUpTo.

- ForEachNode
  - Given the output of another srcQLQuery which yields a node set, match
    set, or list of syntactic variables, visit each of the elements in the given
    order.
  - The ForEachNode also plays a part when it comes to handling XPath
    implementation details, as it is also used to implement a filtering step
    expression within XPath.

- NAryPathPredicate
  - An NAryPathPredicate which contains two possible paths within PDA.
  - This allows for extensions where matching multiple paths are valid but
    they both lead to the same Next Predicate.
  - Short circuiting occurs after one of the paths evaluates to true.
  - There are N visitation pointer one after each possible path within the PDA.

**Syntactic Variable Selecting Traversals**

Selecting of ranges of nodes is handled by the following predicates. Due to the
nature of syntactic variables collection, only the predicates SelectSiblingsUpTo
and SelectChildrenUpTo have a non-null ending node. This is due to the fact specific
patterns used for matching syntactic variables either match everything or everything up to a specific pattern. The syntactic variable selection is presented in further detail in the section describing srcPat.

Syntactic variable predicates modify all modify the program contents of a variable, storing information about the syntactic variable's first element into that context. Predicates which require an ending node to be selected also modify the input context of the syntactic variable.

- SelectAllChildren
  - Selects all of the children of the given context as a syntactic variable.

- SelectAllFollowingSiblings
  - Select all of the following siblings after a given input node.

- SelectChildrenUpTo
  - Selects the first child and uses ReverseFollowingSiblingsPredicate to locate the last element of the syntactic variable.
  - This predicate modifies the output context to be the first child of the input context.

- SelectSiblingsUpTo
  - Select the next sibling element of the input context and uses ReverseFollowingSiblingsPredicate to locate the last element of the variable range.
  - This predicate modifies the output context to be the next sibling element of the input context.
Set Operations

Additionally there are predicates which can be used to evaluate complex relationships between sets of nodes or matches. These operations extend the simple union operation available within XPath to include set union, subtraction, and intersection. This provides a means for computing set difference between complicated expressions which would otherwise be impossible to express using only XPath. The result of evaluating a set operation yields a visitation to every element within the resulting set.

- SetIntersectionPredicate
  - Compute the set intersection of two srcQLQuerys.
- SetMinusPredicate
  - Compute set A - B on two srcQLQuerys.
- SetUnionPredicate
  - Compute the union of two srcQLQuerys.

Expression Predicates

Expression predicates are simply predicates which assert an expression against the input context, and the program context. An XPath like notation is used to describe the expressions, a detailed listing of the grammar is given later on in this document. The results of any expression held by an expression predicate is always implicitly converted to a Boolean value using the conversion rules given at a later point in this document.
Expression predicates never modify input context, program context, or result of a query, they only ever evaluate to true or false. An example of an expression predicate is one that test's the type of node of the input context, the namespace of the input context or the name of the node of an input context. The only modifiable alterable output of an expressions predicate is to refine the principal node type of the input context, to a more specific type.

**Notation**

Due to the complexities and depth of nesting needed to represent srcQL's formal internal representation a simplified notation that leverages XPath-like syntax is used instead. Extensions to the XPath language have been made in order to fully represent all of the necessary operations allowed by srcQL which cannot be represented in pure XPath. The language is referred to as IR or intermediate representation. The representation uses the XPath language with the additions to an XPath step.

Additional axes have been added:

- **reversed-following-siblings::**
  - iterate over all siblings in reverse document order.
- **select-all-children(syntactic-variable-name)::**
  - Select all children for a syntactic variable.
- **select-all-following-siblings(syntactic-variable-name)::**
  - Select all following siblings after the input context.
- **select-all-children-up-to(syntactic-variable-name)::**
Select all children up to the next part of the pattern.

- select-all-following-siblings-up-to(syntactic-variable-name)::

Select all siblings up to the next part of the pattern.

Alteration of the step syntax to include back references, n-ary predicate syntax, and use of functions as individual steps. Original Syntax for an XPath step included the following:

```
step := (AxisName '::*')? NodeTest Predicates*
   | AbbreviatedStep;
```

**Extension Syntax:**

```
AxisName := 'ancestor'
   | 'ancestor-or-self'
   | 'attribute'
   | 'child'
   | 'descendant'
   | 'descendant-or-self'
   | 'following'
   | 'following-sibling'
   | 'namespace'
   | 'parent'
   | 'preceding'
   | 'preceding-sibling'
   | 'self'
   | 'reversed-following-siblings'
;
```

```
SyntacticVariableAxisName :=
   | 'select-all-children'
   | 'select-all-following-siblings'
   | 'select-all-children-up-to'
   | 'select-all-following-siblings-up-to'
;
```

// The second variable is a context back reference
// the this allows the axis to start at another location
// rather than just the output from a previous step.
SyntacticVariableAxis := SyntacticVariableAxisName
'(' VariableReference ( ',' VariableReference)? ')'
;

// Allowing for a context back reference to be set
// as the starting location of the axis traversal
// instead of using the output context of the previous step.
Axis := AxisName ('(' VariableReference ')')?
;

ExtendedAxis := (Axis | SyntacticVariableAxis) '::' ;

step :=
{ (ExtendedAxis? NodeTest)
 | ( '(' RelativeLocationPath ( '|' RelativeLocationPath)*
'')' )
} // the -> syntax allows the output context form a single step to
// be referenced again as the input to a function or another
// axis.
('->' VariableReference)?
Predicates* ;

This language extension allows for a direct mapping into the formal
representation presented in the previous sections.

**srcQL XML Processing Models**

srcQL shall operate on an XML document object model. Evaluating a srcQL
query is the process of executing it with an XML document or node as input. The
processing model for srcQL is designed with speed in mind and differs from the XPath
query evaluation mode. This is because srcQL has a different set of axis traversals and
back references which XPath does not have.

In order to understand the difference between XPath and srcQL's processing
models one must first understand the processing models themselves. The XPath model of
handling XML queries is handled in a breadth first manner. For example, consider the following XPath:

```
expr_stmt/expr
```

XPath first searches for all child elements of `expr_stmt` and creates a node set than for each element of the set searches for children of those elements with the name `expr`. We call this model breadth first because it traverses the breadth of the query rather than the depth of the query.

The srcQL model of processing elements is slightly different in that it handles state based transitions to and from each step. Considering the example above srcQL would traverse each child element until it encountered an `expr_stmt` and than search all child elements of the `expr_stmt` for a child with the name `expr`. If srcQL locates a child with the name `expr` it collects it and continues processing the second set of children until all children of the `expr_stmt` are processed. After which it returns to continue processing the remaining siblings of the matched `expr_stmt` for other `expr_stmts`. This method of matching XPath patterns is called depth first, because it traverses all parts of the query in depth first order.

The differences between the two language's processing models is that XPath uses a breadth first search rather than a depth first search. While both styles have their advantages the depth first search is used with srcQL because it supports the addition of required features needed to implement language features such as back references and allows for optimizations such as short circuiting of queries on a true match.
Another reason why the depth first processing model is used is that, the way that srcPat generates queries maps directly onto this strategy. For example, consider the following srcPat query:

```
new int
```

This query maps into the following intermediate representation:

```
//src:expr[src:operator[.='new']][{following-sibling::src:name | following-sibling::src:call/src:name}]
```

In this case each of the elements are predicates which evaluate to either true or false instead of a node set like in XPath. This means that an if a match is found no additional searching of the pattern is necessary. This allows for quick matches and short circuiting and all of the elements are close together within the tree. Due to the way that srcPat is handled and the way the language features that srcQL needs in order to be implemented the depth first processing model has been chosen.
**srcPat Query Generation**

srcPat is a query language for generating queries from a pieces of source code. srcPat provides the means for generalizing queries through the use of syntactic variables. Presented in this section is how srcPat queries are constructed and used as part of srcQL.

srcQL doesn’t have a grammar the same way that srcQL or XPath does, it only has a beginning and an ending. The beginning and ending of a srcPat query in the context of srcQL is done through the use of keywords or the end of input. When srcPat is used without srcQL all srcQL keywords are ignored and considered to be part of the srcPat input and the entire input is given to srcPat.

The srcPat language was designed to work specifically with srcML and as such does not have any understanding of the source code it’s being beyond that of an input string. srcPat treats srcML as a black box that creates an AST which it turns into an intermediate representation which can be evaluated.

The srcPat algorithm is presented at a high level and then each portion of the algorithm is explained in detail.
```java
QueryPattern srcPat(sourceCodeInput) {
    Locate the starting node of a pattern within the AST.
    Traverse all nodes that are part of the pattern and build a query
    using the intermediate representation.
    Return the generated query from the previous step.
}
```

The process for obtaining the first node of a traversal is best defined using srcML
because there is no formalization which it can be directly mapped to without talking
about srcML. The particularities of determining the first node of a pattern is specific to
srcML and as such is presented in terms of srcML. The first node of a query is defined as
follows:
processingDoc = run pattern through srcML.
If processingDoc's unit element has 2 or more children
    return first child element of unit
else if the unit has only a single child element
    if the first child element is an expr_stmt or decl_stmt
        if the expression didn't contain a ';' 
        append a semi-colon to the expression 
        processingDoc = run expression + ';', through srcML 
        return the expr or decl element 
        else 
            return first child element of unit 
    else 
        return first child element of unit 
else 
    return first child element of unit 

Should an expression or declaration statement not be suffixed by a semi-colon it shall be processed as if the user wishes for the expression or declaration rather than the statement variant itself.

The process of traversing a srcML document and building a query is presented, again in terms of srcML using a SAX like interface:
bool isFirstElement = true;
map<xmlNodePtr, Predicate> nodeToPred;

bool startElement(xmlNodePtr node) {
    if (node is a syntactic variable) {
        if (isFirstElement) {
            // the first XML element can not be a
            // syntactic variable due to srcML
            // structure.
            exit(1);
        }
        xmlNodePtr prevElementSib = getPreviousElementSibling(node);
        xmlNodePtr nextElementSib = getNextElementSibling(node);
        if (nextElementSib && prevElementSib) {
            // insert set context to previous sibling.
            query->appendSetContext(nodeToPred[prevSib])
            query->appendSelectSiblingsUpTo();
        } else if (!nextElementSib && prevElementSib) {
            query->appendSetContext(nodeToPred[prevSib])
            query->appendSelectAllFollowingSiblings();
        } else if (nextElementSib && !prevElementSib) {
            query->appendSelectChildrenUpTo();
        } else if (!nextElementSib && !prevElementSib) {
            query->appendSelectAllChildren();
        }
        return false;
    } else {
        // The first element always needs to be the self axis because
        // that's
        // how the queries actually work correctly with
        // other queries proceeding them.
        if (isFirstElement) {
            isFirstElement = false;
            create ancestor, or descendants based
            on where the srcPat is located
            within srcQL

            return true;
        }

        xmlNodePtr previousSib = getPreviousElementSibling(node);
        if (node->name == "name") {
            if (node->children == nullptr) {
                if (node->parent->name != "name") {
                    if (previousSib) {
                        query->appendSetContext(previousSib);
                        nextPredicate = query->createFollowingSiblingPredicate(true);
                    } else {
                        nextPredicate = query->createChildPredicate(true);
                    }
                } else {
                    nextPredicate = query->createSiblingPredicate(true);
                }
            } else {
                if (previousSib) {
                    query->appendSetContext(previousSib);
                    nextPredicate = query->createFollowingSiblingPredicate(true);
                } else {
                    nextPredicate = query->createChildPredicate(true);
                }
            }
            return true;
        } else {
            if (previousSib) {
                query->appendSetContext(previousSib);
                nextPredicate = query->createSiblingPredicate(true);
            } else {
                nextPredicate = query->createChildPredicate(true);
            }
            return true;
        }
    }
}
srcPat patterns use the following rules for developing patterns from a given input:

If the input specifies part of an expression or declaration than that expression or declaration searched for in all contexts, not just as part of a statement. This shall be
indicated within the pattern by the absence of a semi-colon at the end of the input to a single expression or declaration statement.

If a pattern gives multiple statements than those statements must be siblings within the XML DOM and must be in the order specified.

All other pieces of the srcPat language do not need any special handling beyond the given algorithm.

All srcPat queries many change the initial axis associated with the starting element, but the default is to use descendant:: as the starting axis. The axis is changed to ancestors:: when a srcPat query is used as the input to a WITHIN operator, and scoped-followed-by::node()/descendant:: as when used with the FOLLOWED BY operator.

**srcQL Syntax and Semantics**

Presented in the following is the srcQL grammar and the semantics associated with that grammar. The intermediate representation is used to explain any additional processing requirements within srcQL. The srcQL grammar is presented using EBNF style notation.

\[
\text{start} ::= \text{srcQLProgram} \mid \text{srcPatOnly} \; ;
\]

- The start rule is the entry point of the language grammar.

- The language shall revert to srcPatOnly syntax if the input query does not start with the keyword SET or FIND and in the event that a program does start with SET or FIND and does not meet requirements set forth by the grammar the program is not well formed.
If a srcQLProgram does not start with "SET" or "FIND" than the input shall be processed srcPat.

- This allows for easy specification of simple patterns which only consist of a source code example.

- If srcQLOnly is matched than it shall be the main query of a srcQLProgram.

srcPatOnly := +anySourceCodeToken ;

- srcPatOnly processing of source code shall ignore any srcQL keywords.

- The result of processing the pattern shall be the result evaluating the pattern generated by srcPat and shall start with /descendant:::

  - This means that any pattern given as srcPat shall search the entire XML DOM.

srcQLProgram := assignedVariableList mainQuery;

- The result of the main query shall be the result of the program.

- All variables must be declared prior to uses.

  - This breaks from traditional rule based or declarative language in that it's procedural in that all of the statements and queries are handled in order within srcQL.

**SET Variable Operator**

assignedVariableList := setVariable*;

setVariable := "SET" variableName (srcQLSubQuery | explicitXPath | srcPat);

- The set operator creates a value that can later be used during the evaluation of another following query.
– The declaration of a SET variable shall add a variable to the program state so as to allow reference to that variable by other queries.
– Sorting of set variable shall not be allowed because the type associated with the variable shall be deduced from the context in which it's used. Which may require special sorting in order to optimize evaluation of subsequent queries.

**Set operations**

```plaintext
srcQLQuery := findOp ("INTERSECTION" | "MINUS" | "UNION") srcQLQuery) ?;
```
– The evaluation order for set operations shall be from left-to-right with the left most being evaluated first.

```plaintext
mainQuery := srcQLQuery orderingOperators?;
```
– The result type of the main query shall be a set of matches sorted in document order unless ordering operations are used.
– All main queries shall start with the FIND operator.

**FIND operator**

The FIND operators is the first operator within every srcQL query. It is used to specify the search context or what should be returned depending usage. The search context is defined as being the location to search for a patterns within and/or the name of the root element. The search context may be specified by name by using XPath or implicitly derived using a srcPat query.
findOp := findGuessContext
  | findImplicitFromContext
  | findVarWithinFrom
  | findVariableWithinContains
  | findXPathOrSrc
  | findXPathOrSrcInFrom
;

findGuessContext := "FIND" ( "/" | "/" | "/" | "." )
containsNoFollowedBy;

containsNoFollowedBy := "CONTAINS"
  (explicitSrcPat | explicitXPath | srcPat) withinOp?
whereOp?;

- All possible values of ( "/" | "/" | "/" | "." ) shall have the same effect.
- The search context is defined as the one that's located within the following contains operator.
- The following two examples are semantically equivalent
  - FIND new int
  - FIND * CONTAINS new int

findImplicitFromContext := "FIND" ("*"| ".") fromOp withinOp?
whereOp? containsList ;

- Queries of the following form shall provide a filtering on result of the FROM operator and leverage the root element of each match yielded by the FROM operator as the search context selected with FIND.
- The WHERE operator following the FROM operator shall have access to syntactic variables located within the FROM operator and within the current query.
- Comparing a syntactic variable from within the FROM operator and within the current query shall unify those variables be the same in both queries.
- The result type of findImplicitFromContext shall be the same type as is returned by the FROM operator.

```plaintext
findXPathOrSrc := "FIND" (explicitXPath | explicitSrcPat | XPath | srcPat) withinOp? whereOp? containsList;
- Queries of this form shall have their search context specified using XPath or srcPat.
- The CONTAINS operator shall search within the search context given by the FIND operator.
- The WITHIN operator shall search relative to the search context.

- Queries of this form shall have their search context specified using XPath or srcPat.
- The search context shall be located within the search context yielded by the FROM operator.

findVarWithinFrom := "FIND" logicalVariable fromOp;
- Locating a syntactic variable from within a FROM operator leverages the FROM operator.
- FIND operators which select syntactic variables from within a FROM operator shall not provide additional operators other.
- The result type of queries of this form shall be a set of syntactic variables.
- The syntactic variable given shall exist within the FROM operator or the program is said to be not well formed.

\[
\text{findVariableWithinContains} := \text{"FIND" logicalVariable containsNoFollowedBy;}
\]
- The selected variable shall exist within the CONTAINS operator or the program is not well formed.
- The result type of queries of this form shall be a set of syntactic variables.
- The search context for queries of this form shall be implicitly derived from the XPath or srcPat expression given to the CONTAINS operator.

**CONTAINS Operator**

The CONTAINS operator provides a means of specifying unordered patterns within a search context or to provide the search context itself in special cases.

CONTAINS provides the means of looking for multiple statements in any order within a search context. The "any descendant" relationship between a pattern and the search context is given using the CONTAINS operator.

\[
\text{containsList} := \text{containsOp*;}
\]
\[
\text{containsOp} := \text{"CONTAINS" (explicitXPath | explicitSrcPat | srcPat)}
\]
- The CONTAINS operator shall implicitly and explicitly except srcPat and explicitly accept XPath.
- Explicit and implicit patterns are described in a later section.
CONTAINS shall implicitly prefix all patterns of XPath and srcPat with the axis descendant::

**FOLLOWED BY Operator**

The FOLLOWED BY operator provides an ordered scoped relationship between to patterns within a search context. It provides a means of relating patterns in an ordered fashion, and scoped to within the search context.

\[
\text{followedByOp} := \text{"FOLLOWED BY"} \ (\text{explicitXPath} \ | \ \text{explicitSrcPat} \ | \ \text{srcPat}) \\
\text{withinOp? whereOp? followedByOp?};
\]

- The FOLLOWED BY operator shall implicitly and explicitly except srcPat and explicitly accept XPath.
- Explicit and implicit patterns are described in a later section.
- The FOLLOWED BY operator shall only follow a CONTAINS operator or another FOLLOWED BY operator.
- The FOLLOWED BY operator's search shall start at the next element in document order after the previous CONTAINS or FOLLOWED BY operator.
- All patterns given to the FOLLOWED BY operator shall be implicitly prefixed with axis scoped-followed-by::node()/descendant::

**FROM Operator**

The FROM operator is the sub-query operator which allows for advanced filtering and selection operations. These operations include providing a means of selecting patterns within other patterns, selecting syntactic variables from within complex expression which cannot be specified using other means, and provides a means of simple
pre-query evaluation. The relationship between a FIND and a FROM is defined in the previous section presenting the FIND operator.

\[
\text{fromOp} := \text{"FROM" (srcQLQuery | explicitSrcPat)} \text{"END FROM";
}\]

- The FROM operator shall denote a sub-query to select elements from within.
- The FROM operator's result type is based on the query it's contains.
- When the FROM operator receives explicitSrcPat as input the result type shall be a set of matches.
- When the FROM operator receives srcQLQuery as input the result type shall be the same as the query.

**WITHIN Operator**

The WITHIN operator provides a means of saying that a pattern must have an ancestor which matches a given pattern without altering the search context. WITHIN is a simple means of specifying child to parent relationship that does not alter the search context and which may exist within the search context.

\[
\text{withinOp} := \text{"WITHIN" (explicitXPath | explicitSrcPat | XPath | srcPat);
}\]

- The WITHIN operator shall not modify the search context of a query.
- All patterns given with the WITHIN operator shall be implicitly prefixed with ancestor:: axis.

**WHERE Operator**
The WHERE operator provides a means of asserting against patterns using syntactic variables, XPath, operators, and functions to further refine srcQLQuery by using aspects of the pattern which may be impossible to specify as srcPat and extremely difficult to specify using XPath. WHERE accepts expressions which are evaluated in the context of the operator that it's associated with.

```plaintext
whereOp := "WHERE" expr;
```

- The WHERE operator's expression shall be evaluated after the operator which it follows.
- The context of the WHERE operator shall be the match context element of preceding operators pattern.
- Any syntactic variable which occurs as part of an expression shall have been used in any pattern prior to the evaluation of the WHERE operator, if a variable is not use or is not given as input parameter to the program then the program is not well formed.
- The result of the expression shall be converted to a Boolean value for which true shall be considered a match and false shall not be a match.

**Ordering Operators**

```plaintext
orderingOperators := groupByOp | orderByOp | orderGroupsBy;
groupByOp := "GROUP BY" (functionCall | logicalVariable | "syntacticCategories");
```

- The result type of an order by operator is always a group type where the first element is a string and the second element is a set of values.
  - The set of values shall be of one of the following types:
- Set of Strings
- Set of Syntactic Variables
- Set of Matches
- node set

- Grouping of elements is a transformation on the result of a query into another type.

- GROUP BY functionCall shall require an external function from an external library and is defined as an extension mechanism of the srcQL language.

- Grouping by syntactic category shall not be proceeded by an ordering statement, if it is the program is not well formed.

- Logical Variable grouping shall be done by converting the value of a syntactic variable into a string and placing all variables with the same string value into the same group.

- All grouping operations shall maintain relative order of the result type given to them, but the default order of the groups is implementation defined.

- Syntactic category of a node is defined as the following:
  - The document order of all ancestor element until a category terminator is reached.
  - The following element names are category terminators:
    - class
    - unit
    - for
- if
- while
- struct
- union
- do
- expr_stmt
- decl_stmt
- function
- property
- interface
- function_decl

orderByOp := "ORDER BY" (logicalVariable | functionCall);

- The ordering of types shall be based on an extension function or be handled by syntactic variable.
- ordering by syntactic variable shall be determined by the first occurrence of a syntactic variable converted to a string and compared lexically with other syntactic variables of other matches.
- The default ordering of a query shall be document order.
- The default ordering when ordering by syntactic variable is alpha numeric ordering.

orderGroups := groupByOp orderByOp;
The ordering of groups by a function shall be allowed through the use of a ORDER BY operation.

The following ORDER BY operation is only allowed to use an extension function or the srcQL provided count function which sorts the groups by the number of elements which they contain from greatest to smallest.

**Pattern Handling**

```plaintext
srcPat := /* Source code to process as a srcPat. */;
explicitSrcPat := "src" srcPat;
explicitXPath := "XPath" XPath;
anySourceCodeToken := numberLiteral |
| stringLiteral |
| logicalVariable |
| /* Any other source code token */ |
```

Handling of pattern matches associated with operators shall be given using either an XPath representation or by using a srcPat expression

Explicit pattern specification is allowed and not considered as part of a given pattern.

The explicit patterns allow for disambiguation where a source code pattern and an XPath pattern may overlap in terms of language grammar.

For a full description of srcPat see the section on srcPat.

For a full description of the XPath language please see the XPath 1.0 language standard.

Source code tokens are defined as being a non-srcQL keyword, when a language is tokenized using its language standard.
Expressions

The srcQL language provides a means for both XPath style comparisons and C++ style comparisons both have the same affect.

Conversions Semantics

Conversions follow the same basic rules as XPath with a few extensions. The extensions are presented within this section. Conversions are provided through implicit conversions which are applied when using operators and functions and is detailed in subsequent sections or through explicit conversions using a conversion function which is described in the built-in functions section of this document. Any conversions which are not part of XPath and not listed in this section shall result in a program which is not well formed.

1. String

In addition to the XPath conversion rules the following conversions are also valid upon a string type.

a. String to Character

   i. The conversion from string to character shall be the first character of the string. If the string does not have a first character the value of the character shall be '\0' or zero.

b. String to String Set

   i. The result of converting a string to a string set shall be a string set of size one containing the string.
2. Int
   
a. Int to String Set
   
i. The conversion of an Int into a String Set shall be the result of converting the Int into a string followed by the conversion that string into a String Set.

b. Int to Number
   
i. The conversion of an Integer into a floating point number shall be the result of converting the Int into the equivalent floating point representation.

3. Boolean
   
a. Boolean to Character
   
i. The conversion of a Boolean type into a character shall result in a character with ordinal value of 1 if the Boolean is true and 0 if false.

b. Boolean to String Set
   
i. The conversion of a Boolean into a String Set shall be the result of first converting the Boolean into a string and converting the string to a string set.

4. Number
a. Number to Int
   i. The conversion of a number into an integer shall be the result of truncating the number equivalent to applying a floor operation and the resulting value converted into an int.
   ii. A conversion of the value NaN, or positive or negative infinity shall yield an integer value of zero.

b. Number to String Set
   i. The result of converting a Number into a String Set shall be the result of converting the number into a string and converting the string in to a String Set.

5. Character
   a. Character to String
      i. The conversion of character to a string shall result in a string with size one iff the character value is not '0', in which case the result shall be a string of length zero.

   b. Character to Int
      i. The conversion of a character to an Int shall result in the ordinal value of the character's in integer value according to the UTF-8 standard.

   c. Character to Boolean
i. The conversion of a character to a boolean shall result in true iff the character is not \'0\'.

d. Character to Number

i. The conversion of a character to a Number shall be the result of first converting the character to an Integer and converting that Int into a number.

e. Character to String Set

i. The conversion of a character shall be the result of first converting the character into a string than converting that String into a String Set.

6. Node Set

In addition to the XPath conversion rules for a Node Set type a node set is convertible to a String Set, Match, Match Set, Set of Node Sets, and a Set of Match Sets.

a. Node Set to Int

i. The result of converting a Node Set to an Int shall be the result of converting the first element of the Node Set into a String representation followed by the conversion of the String into an Int. If the Node Set is empty the result of applying the conversions shall be the integer value zero.

b. Node Set to Character
i. The conversion of a Node Set to Character shall be the conversion of the first element of the Node set into a string representation, followed by the conversion of resulting string into a Character. If the Node Set is empty the result of the conversion to a character shall be '\0'.

c. Node Set to String Set
   i. The conversion of a Node Set into a String Set shall be the result of converting all nodes within the Node Set into their string representations and placing them into a String Set.

d. Node Set to Match
   i. The conversion of a Node Set to a Match shall be the result of converting the first node of the node set into a match. If the Node Set is empty the result shall be null, which is defined as a Match with the root element set to null.

e. Node Set to Match Set
   i. The result of converting a Node Set into a Match set shall be the result of converting each node within the Node Set into a Match and placing the elements of the Node Set into the Match Set in document order.

f. Node Set to Set of Node Sets
i. The conversion of a Node Set into a Set of Node Sets shall result in a Set of Node Set for which the starting Node Set shall be the first element.

g. Node Set to Set of Match Sets

i. The conversion of a Node Set into a Set of Match Sets is the result of first applying the conversion from Node Set to Match Set followed by the conversion of the resulting node set into a Set of Match Sets.

7. Syntactic Variable

a. Syntactic Variable to String

i. The conversion of a Syntactic variable into a string shall be the result of converting all nodes within the syntactic variables range into their normalized string representation and concatenating those values in corresponding document order.

ii. Normalized string representation shall be the result of placing a space after and before each element within the XML DOM followed by the replacement of multiple whitespace outside of string literals and with a single whitespace.
iii. If a Syntactic Variable has no nodes (first node == last node) the result of the conversion shall be an empty string.

b. Syntactic Variable to Int

i. The conversion of a syntactic variable to an Int shall be the result of converting the syntactic variable into a string than converting that string into an integer. If the value of the resulting string is not an integer zero is returned.

c. Syntactic Variable to Boolean

i. The conversion of a syntactic variable into a Boolean shall be that the first node of a syntactic variable does not equal the last node of a syntactic variable.

d. Syntactic Variable to Number

i. The conversion of a Syntactic Variable to a Number shall be the result of converting the Syntactic Variable into string than converting the String into a Number.

e. Syntactic Variable to Character

i. The conversion of Syntactic Variable to Character shall be the result of converting the Syntactic Variable into a String and converting the string into a Character.

f. Syntactic Variable to Node Set

i. The conversion of a Syntactic Variable to Node Set shall be the conversion of a Syntactic Variable into a Node which
simply takes the first node of the Syntactic Variable range and placing it as the first node within a Node Set.

g. Syntactic Variable to String Set
   i. The conversion of a Syntactic Variable to String Set shall be the result of converting a Syntactic Variable to a String and converting the resulting String into a String Set.

h. Syntactic Variable to Syntactic Variable List
   i. The conversion of a Syntactic Variable to a Syntactic Variable List shall yield the a Syntactic Variable List of one element with the first element being the Syntactic Variable.
   ii. If the Syntactic Variable contains no nodes the resulting conversion yields an empty Syntactic Variable List.

i. Syntactic Variable to Match
   i. The conversion of a Syntactic Variable to Match shall be a Match for which the root node is the first node of the Syntactic Variable.

j. Syntactic Variable to Match Set
   i. The conversion Syntactic Variable to Match Set shall be the result of converting a Syntactic Variable into a Match than converting that Match into a Match Set.
ii. If the Syntactic Variable contains no nodes the resulting conversion yields an empty Match Set.

k. Syntactic Variable to Set of Node Sets

i. The conversion of a Syntactic Variable to Set of Node Sets shall be the result of first converting the Syntactic Variable to a Node Set followed by the conversion of the resulting Node Set into a Set of Node Sets.

ii. If the Syntactic Variable contains no nodes the result of the conversion shall be an empty Set of Node Sets.

l. Syntactic Variable to Set of Match Sets

i. The conversion of a Syntactic Variable to Set of Match Sets shall be the result of converting the Syntactic Variable to a Match Set followed by the conversion of the resulting Match Set into a Set of Match Sets.

ii. If the Syntactic Variable contains no nodes the result of the conversion shall be an empty Set of Node Sets.

8. String Set

The only valid conversion on a String Set is the conversion to a Boolean type. This is because String Sets are unordered and may contain values
which don't occur as part of the XML DOM. Because of this the normal conversion rules do not apply to a String Set.

a. String Set to Boolean

   i. The conversions of a String Set to a Boolean shall be if the collection contains at least one String.

9. Syntactic Variable List

   a. Syntactic Variable List to String

      i. The conversion of a Syntactic Variable List to String shall be the result of converting the Syntactic Variable List to a Syntactic Variable and converting the Syntactic Variable to a String.

   b. Syntactic Variable List to Int

      i. The conversion of a Syntactic Variable List to Int shall be the result of converting the Syntactic Variable List to a Syntactic Variable and converting the Syntactic Variable to an Int.

   c. Syntactic Variable List to Boolean

      i. The conversion of a Syntactic Variable List to Boolean shall be the result of converting the Syntactic Variable List to a Syntactic Variable and converting the Syntactic Variable to a Boolean.
d. Syntactic Variable List to Number
   i. The conversion of a Syntactic Variable List to Number shall be the result of converting the Syntactic Variable List to a Syntactic Variable and converting the Syntactic Variable to a Number.

e. Syntactic Variable List to Character
   i. The conversion of a Syntactic Variable List to Character shall be the result of converting the Syntactic Variable List to a Syntactic Variable and converting the Syntactic Variable to a Character.

f. Syntactic Variable List to Node Set
   i. The conversion of a Syntactic Variable List to Node Set shall be the result of converting each element of the Syntactic Variable List into a node using the first node of each Syntactic Variable and placing each node into the Node Set.

g. Syntactic Variable List to String Set
   i. The conversion of a Syntactic Variable List to String Set shall be the result of converting each element of the Syntactic Variable List into a String and placing each String into the String Set.

h. Syntactic Variable List to Syntactic Variable
i. The conversion of a Syntactic Variable List to Syntactic Variable shall the first element of the Syntactic Variable List. If the list is empty than the resulting Syntactic Variable shall contain no nodes within its range.

i. Syntactic Variable List to Match

i. The conversion of a Syntactic Variable List to Match shall be the result of converting the first Syntactic Variable within the Syntactic Variable List into a Match. If the Syntactic Variable List contains no elements the result of the conversion shall be a Match with a null root element.

j. Syntactic Variable List to Match Set

i. The conversion of a Syntactic Variable List to Match Set shall be the result of converting each element of the Syntactic Variable List into a Match using the first node of each Syntactic Variable and placing each Match into a Match Set.

k. Syntactic Variable List to Set of Node Sets

i. The conversion of a Syntactic Variable List to Set of Node Sets shall be the result of converting each element of the Syntactic Variable List into a Node Set than converting the Node Set into a Set of Node Sets.

l. Syntactic Variable List to Set of Match Sets
i. The conversion of a Syntactic Variable List to Set of Match Sets shall be the result of converting each element of the Syntactic Variable List into a Match using the first node of each Syntactic Variable and placing each Match into a Match Set than converting the Match Set into a Set of Match Sets.

10. Match

a. Match to String
   i. The conversion of a Match to a String shall be the string representation of the root element of a match.

b. Match to Int
   i. The conversion of a Match to an Int shall be the result of converting the Match to a String and converting the String into an Int. If a Match has no root element than the result of the conversion shall be an empty string.

c. Match to Boolean
   i. The conversion of a Match to a Boolean shall be the result of converting a Match to a String and converting the String to a to a Boolean. If the Match has no root element than the conversion shall yield a false result.

d. Match to Number
i. The conversion of a Match to a Number shall be the result of converting a Match to a String than converting the String to a Number.

e. Match to Character

   i. The conversion of a Match to a Character shall be the result of converting a Match to a String and the String to a Character.

f. Match to Node Set

   i. The conversion of a Match to a Node Set shall be the creation of a Node Set where the first element of Node Set shall be the root element of the Match. If the Match has no root element the conversion shall yield an empty Node Set.

g. Match to String Set

   i. The conversion of a Match to a String Set shall be the result of converting a Match to a String and the String to a String Set.

h. Match to Match Set

   i. the conversion of a Match to a Match Set shall be a Match Set where the first element of the Match Set is the Match.

i. Match to Set of Node Sets
i. The conversion of a Match to a Set of Node Sets shall be the result of converting a Match to a Node Set and converting the Node Set to a Set of Node Sets.

j. Match to Set of Match Sets

i. The conversion of a Match to Set of Node Sets shall be the result of converting a Match to a Node Set and converting the Node Set to a Set of Node Sets.

11. Match Set

a. Match Set to String

i. The conversion of a Match Set to a String shall be the result of converting the first Match within a Match Set to a String. If the Match Set contains no matches than the result shall be an empty string.

b. Match Set to Int

i. The conversion of a Match Set to Int shall be the result of converting the first Match into an Integer value. If the Match Set contains no elements the result of the conversion shall be zero.

c. Match Set to Boolean

i. The conversion of a Match Set to a Boolean shall be the result of converting the first Mach to a Boolean value. If
the Match Set contains no matches the result shall be a false value.

d. Match Set to Number
  i. The conversion of a Match Set to a Number shall be the result of converting the first Match to a Number. If the Match Set contains no elements the resulting value shall be NaN.

e. Match Set to Character
  i. The conversion of a Match Set to a Character shall be the result of converting the first Match to a Character. If the Match Set contains no matches the result shall be '0'.

f. Match Set to Node Set
  i. The conversion of a Match Set to Node Set shall be a Node Set containing the root element of each Match within the Match Set.

g. Match Set to String Set
  i. The conversion of a Match Set to a String Set shall be the result of converting all Matches to String and placing them into a String Set.

h. Match Set to Match
  i. The conversion of a Match Set to a Match shall be the first match of the Match Set. If there are no elements within the
Match Set the conversion results in a Match with no root element.

i. Match Set to Set of Node Sets
   i. The conversion of a Match Set to a Set of Node Sets shall be the result of converting a Match Set to a Nodes Set and then converting the Node Set into a Set of Node Sets.

j. Match Set to Set of Match Sets
   i. The result of converting a Match Set to a Set of Match Sets shall be a Match Set where the first element shall be the Match Set being converted.

*Constant Expressions*

stringLiteral := "\" character* "\";

numberLiteral := /* any numeric literal floating point or integer */;

integer */;

variableName := /* a C/C++ identifier */;

logicalVariable := "$" variableName;

- A logical variable shall name a syntactic variable, set variable, or a parameter from the input to the srcQL program.
- A variable shall not have the same name as another set variable or program parameter. If a variable shares a name with one of these types of variables the program is not well formed.
A logical variable is one that occurs within a srcPat pattern, a reference to that
variable is one that occurs as part of an expression, instead of within a pattern.

**Logical Operators**

\[
\text{expr} := \text{logicalOr};
\]

\[
\text{logicalOr} := \text{logicalAnd} \ ("||" \ \text{logicalAnd})?;
\]

\[
\text{logicalAnd} := \text{equalityOps} \ ("&&" \ \text{equalityOps})?;
\]

**Comparison Operators**

\[
\text{equalityOps} := \text{comparisonOps} \ ("==" | "!=") \ \text{comparisonOps})?;
\]

\[
\text{comparisonOps} := \text{additiveOps} \ ("<" | ">" | "<<" | ">=") \ \text{additiveOps})?;
\]

**Arithmetic Operators**

\[
\text{additiveOps} := \text{multDivMod} \ ("+" | "-") \ \text{multDivMod})?;
\]

\[
\text{multDivMod} := \text{uniaryNegation} \ ("*" | "/" | "%") \ \text{uniaryNegation})?;
\]

\[
\text{uniaryNegation} := "-"? \ \text{primayExpr};
\]

\[
\text{primayExpr} = \text{numberLiteral}
\]

\[
\mid \ \text{logicalVariableRef}
\]

\[
\mid \ \text{setVariableRef}
\]

\[
\mid \ \text{stringLiteral}
\]

\[
\mid \ \text{functionCall}
\]

\[
\mid ("(" \ \text{expr} ")")
\]
Variables

setVariableRef := logicalVariable;
logicalVariableRef := logicalVariable;

A variable within the srcQL language shall be either the result of a SET statement or the one that is given as a parameter to the srcQL program.

Functions

functionCall = cppIdentifier "(" parameterList? ")";
parameterList := expr ("," expr)*;

A function call is similar to C-style function class in that it has the same representational syntax as C-style languages. Functions in srcQL are either built in or shall be provided as an extension to the srcQL language and must be defined outside of a srcQL query program. The extension mechanism is implementation defined.

Built-in Functions

The srcQL language provides the following functions.

srcQL provides all XPath functions usually available within XPath 1.0 as well as providing the match function in order to interact with syntactic variables rather than just interacting with simple text.
The srcQL match function is a function which takes two parameters the first parameter shall be a string to match against and the second shall be a regular expression to use for matching. An example function signature for the srcQL match function in C-style representation would look like the following:

```c
bool match(string str, string pattern)
```

This allows for anything that is convertible to text to be compared against a regular expression.

Other functions are underdevelopment and will be added at a later time in order to extend language capabilities.
REFERENCES


[libASTMatchers] libASTMatchers, "libASTMatchers",


