Remote Dependency Analyzer

Operational Concept Document

DECEMBER 2018

Project 4

Souradeepta Biswas (SUID: 737449820)
CSE-681: Software Modelling and Analysis

Instructor: Jim Fawcett

Table of Contents

Executive Summary

The purpose of this project is to implement a Remote dependency Analyzer which it intended to perform dependency analysis and strong component analysis based on the request generated by the client process and code present on the remote server. On completion of the analysis process, the remove server sends the output which is a list of all the packages and their dependencies on other packages if any and the strong components formed between the packages based on Trajan's algorithm.

The key capabilities of the project would provide are:

- The client would send the requests using a message dispatcher and the server would receive them and reply back to the messages using its own message dispatcher.
- The test request would contain the list of files on which the operation is to be performed and the functionality to perform on them.
- Once a request to perform dependency analysis is received the files would be picked up from the server and then firstly they would be tokenized using a state-based tokenizer.
- With these tokens semi expressions would be made and then type analysis would be performed on the files. This would create a type table format which would aid us to perform our dependency analysis.
- Our output of the dependency analysis would be used to perform strong component analysis on the packages.
- Once the request is executed the results are sent back to the client.

The other users for this project include students, teachers, teaching assistants, professors, companies, testing engineers, developers and quality assurance engineers.

Below are some of the critical issues associated with this project, the solutions for which are discussed in detail in later sections. They are:

- Handling many special cases in tokenizer processing.
- .Net streams not supporting peeking at more than the first available character.
- Aliases interfering with the dependency analysis.

1. Introduction

In order to successfully implement big systems we need to partition code into relatively small parts and thoroughly test each of the parts before inserting them into the software baseline. As new parts are added to the baseline and as we make changes to fix latent errors or performance problems we will re-run test sequences for those parts and, perhaps, for the entire baseline. Managing that process efficiently requires effective tools for code analysis as well as testing. How we do that code analysis is through our Remove Package Dependency Analyzer project.
The project consists of extracting lexical content from source code files, analyzing the code's syntax from its lexical content, and forming semi expressions and creating the type table. Our project hence shall expand on the part of how the Lexical Scanner would first identify the tokens and then create semi expressions out of them.

1.1 Definitions

Token: Tokens are characters extracted from the source code. They can be either symbols, whitespaces, alphabets or numbers. Token boundaries are white-space characters, transitions between alphanumeric and punctuator characters, and comment and string boundaries. Certain classes of punctuator characters belong to single character or two character tokens so they require special rules for extraction.

Token State: For our project we shall consider the following to be our token states:

- WhiteSpaceState: This is for the white spaces in the code.
- AlphaState: This is for the alphabets and numbers without sign in the code. Eg. ‘a’, ‘B’, ‘9’ and ‘0’.
- PunctState: This is for the various punctuation marks and signs in the code. Eg. ‘@’, ‘;’ and ‘#’.
- SingleQuoteState: This is for the tokens which include characters inside single quotes. Eg. ‘asdf1234!@#$’.

- DoubleQuoteState: This is for the tokens which include characters inside double quotes. Eg. “asdf1234!@#$”.
- CCommentState: This is for the tokens which come along with the “//” characters which is a single line comment in C and C++. Eg. “//this is a comment”.
- CppCommentState: This is for the tokens which come between and including the characters “/*” and “*/”. This is akin to the multiline comments in C++. Eg. “/*asdfasdfasdfasdasd f1232345 !@#$!@#$!@#$asdf”.
- SpecialPunctState: This is for special states which are not covered in the above states. These include states like “==”, “+=”, “<<” or “!=”. Many programming languages have different special set of symbols.

Tokenizer: This is a package which extracts tokens.

Semi Expression: These are groups of tokens forming sets, each of which contain all the information needed to analyze some grammatical construct without containing extra tokens that have to be saved for subsequent analyses. Semi Expressions are determined by special terminating characters: semicolon, open brace, close brace, and newline when preceded on the same line with 'using' to name some.

TerminatorState: These are states when a semi expression is considered complete and they occur after extracting any of these single character tokens: semicolon, open brace and closed brace. These states are SemicolonState, OpenBraceState and CloseBraceState respectively. Also terminates on extracting newline if a '#' or a “using” is the first token on that line which are PoundState and UsingState. There is another special state called the ForState which provides a facility providing rules to ignore the two semicolons within parentheses in a for(;;) expression.

1.2 Objective

The objective of this project is to implement the Remote Dependency Analyzer to analyse dependencies between files and find strong components among them using a State-based Tokenizer. In project#2 we create the Lexical Analyzer which finds tokens based on our coded states to give the output of tokens. The we used these token to find and create semi expressions. In project#3 we use these semi expressions generated to create type tables. These type tables are the core of our dependency analysis since they will help identify user defined objects used across files. The output of our dependency list would have all the files selected by user and a list of files which depend on them if such files exist. This list would be used to find the strongest component. Project #4 directly extends this functionality to help create a client and server for us to be able to send requests remotely. We also would have a GUI frontend for ease of usage.

1.3 Organizing Principles

The project will be implemented in C# using the .Net framework. Windows Communications Framework(WCF) and Windows Presentation Framework(WPF) for establishing communication between the client and server and to create the GUI to interact with them respectively. The key modules are Tokenizer and the SemiExpression. The functionalities of each module are implemented through individual packages, which will be explained in detail in the later sections.

2. Uses

There are a lot of reasons you may wish to analyze source code. For example:

- **Building code analysis tools:** We can use our lexical scanner such that we can perform type analysis and dependency analysis. Type analysis tells us the various type definitions in our source code. For example for C# it could be classes, structs, enums, and aliases. Dependency analysis shows us all the files which depend on our stated source file by checking name of any time defined in the stated file present in other files.
- **Ownership of code files:** We can use our analyzer to perform plagiarism detection since we deconstruct all our tokens and expressions to barebones form.
- **Evaluating code metrics:** We can also find out number of embedded function calls and perform an estimated performance analysis of the code.

- **Code conversion:** Another use while very ambitious but possible would be have sufficient rules to convert code from one language to another. For example converting C# code to Java.
- **Code Trace:** A code trace is a method for simulating the execution of the code in order to verify that it works correctly before the compilation. Since the scanner can do type analysis on the file that can be expanded to perform a trace too.

3. Users

Based on the above stated uses of the project the users can be students, teaching assistants and professors, developers, and quality assurance (QA) engineers.
Students: Can verify their source code for plagiarism detection and as a basic performance analysis.
Professors: Can use it the same way as above.
Developers: Can use it for dependency analysis of source files.
QA engineers: Can use it on source file for performance analysis, to check adherence to a company mandated code structure.

3.1 Compilers

Compilers are computer programs that translate a program written in high level language into a program into a low level language. A compiler must first understand the source-language program and a lexical scanner is involved in first phase. This phase takes the source code as a stream of characters and identifies distinct words (tokens) such as variable names, keywords and punctuators. The second phase determines the validity of syntactic organization of the program and produces Abstract Syntax Tree (AST). Semantic analysis checks whether the AST follows the rules of a language.

4. Partitions

Here we discuss the package and class partitions of the project.

4.1 Package Partitions

Based on the functionality all the activities or tasks are split into different packages which have independent roles to play. The following are the packages that belong to the Lexical Scanner using State-Based Tokenizer.

- **Toker :** Extracts words, called tokens, from a stream of characters. Token boundaries are white-space characters, transitions between alphanumeric and punctuator characters, and comment and string boundaries. Certain classes of punctuator characters belong to single character or two character tokens so they require special rules for extraction.
- **SemiExp:** Groups tokens into sets, each of which contain all the information needed to analyze some grammatical construct without containing extra tokens that have to be saved for subsequent analyses.
- **Parser:** This contains the code for a parser. The parser detects the code constructs defined by the rules and actions. This lets us have the flexibility of changing parser between languages. Here IRulesandActions has the **p**ublic Interface for the RuleandActions. Rule defines the rules which which are to be tested against on the semi expressions that we have generated. Actions defines the actions which would be triggered once a particular rule is satisfied. This package also implements a generic stack to help track position in code scope by basic push, pop type functions.
- **FileMgr:** A file manager program which helps use maintain the input files using System.IO functions. Clients use the FileMgrFactory to create an instance bound to an interface reference.
- **Element:** This package defines a class which holds the user-defined datatypes like class,struct,enum etc.
- **TypeTable:** This manages the type information generated during the type-based code analysis

- **DependencyTable:** This manages the file information generated during type based code analysis,
- **CsGraph:** This is a class which implements a graph with nodes and vertices to be used for strong component analysis
- **Environment:** This has a set of variables needed to connect to the server and the client.
- **ImessagePassingCommService:** Interface for the message passing communication.
- **MessagePassingCommService:** This package constructs the sender and receiver instance which is required for our communication.
- **NavigatorClient:** This package defines the WPF application processing by the client.
- **NavigatorServer:** This package returns file and directory information about its server files. It also calls our core program functionalities
- **Executive:** This package creates our code analyzer instances and gives us outputs to our inputs
- **Display:** Has various display functions needed for console prints.

Below is the package diagram for the above packages.

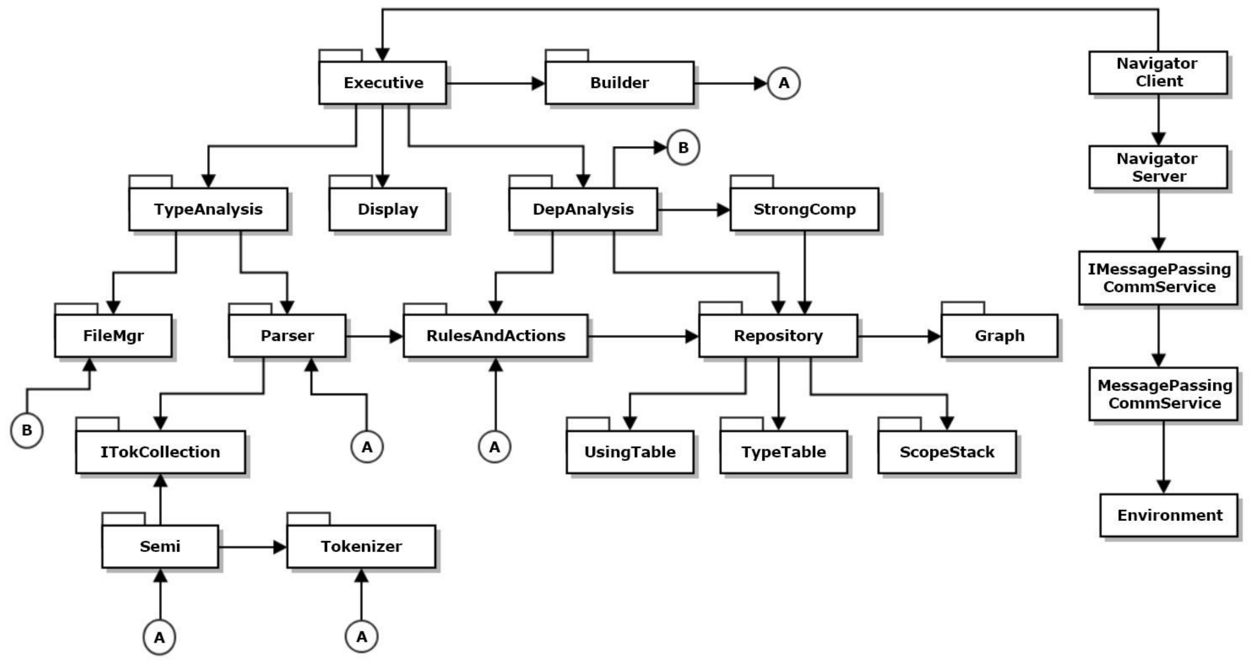


Figure 1: Package Diagram for Remote Dependency Analyzer

Explanation of the diagram:

The client from NavigatorClient sends a request to see the files available for processing. Once the server, NavigatorServer responds with the list, the client selects the files required and chooses the functionality needed. The client then goes on to call the Executive package functions which sends a request to TypeAnalysis which calls the Parser which accesses the semiexpressions generated on Semi package when sent from Toker package based on public interface ITokCollection. Once the type analysis is performed based on Repository package which needs the UsingTable, TypeTable and ScopeStack for our process, the outputs are sent to DepAnalysis and then either sent to StrongComp or to the executive based on the request from the client.

4.2 Class Partitions

4.2.1 Tokenizer Class

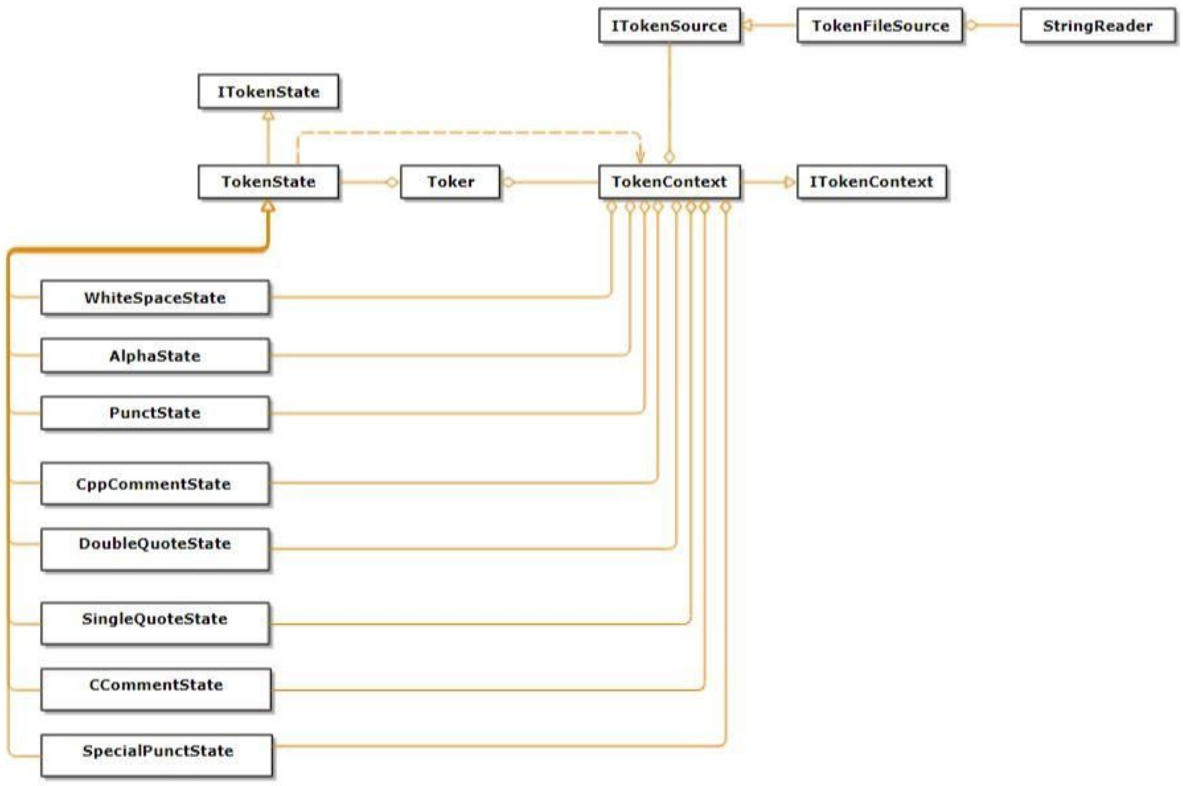


Figure 2: Class Diagram for Tokenizer

Explanation of the diagram:

- The Toker class implements the TokerState which is for detecting state pattern. Toker class contains the list of applications which would be needed to collect tokens.
- The TokenFileSource is the class which reads the input source file and extracts tokens.
- The StringReader class is used for reading byte order mark characters and alternate text encoding like UTF-32 characters. It has an aggregation relationship with the TokenFileSource, since it would not exist without the above class.
- ITokenSource does operations which would be needed to utilize our token source. It would typically contain functions like open source, close source, line count of token and peek function which would look at the source and not remove the character and next which would remove it.
- TokenContext holds the instance of all the states which were discussed above and the token source. This is a part of the toker class.
- TokenState, ITokenSource and ITokenContext are interfaces for their respective classes.
- TokenState is the class which is the base class for all the different token states possible in our program.

4.2.2 Semi Expression Class

For code analysis we collect tokens into grammatical sequences, called SemiExpressions, by terminating the collection on semicolons, curly braces, or newlines if the line starts with "#". Each of these collections is usually sufficient to detect a single grammatical construct without including tokens from the next grammatical entity.
Below is the class diagram for the SemiExpression showing the set of classes for that package.

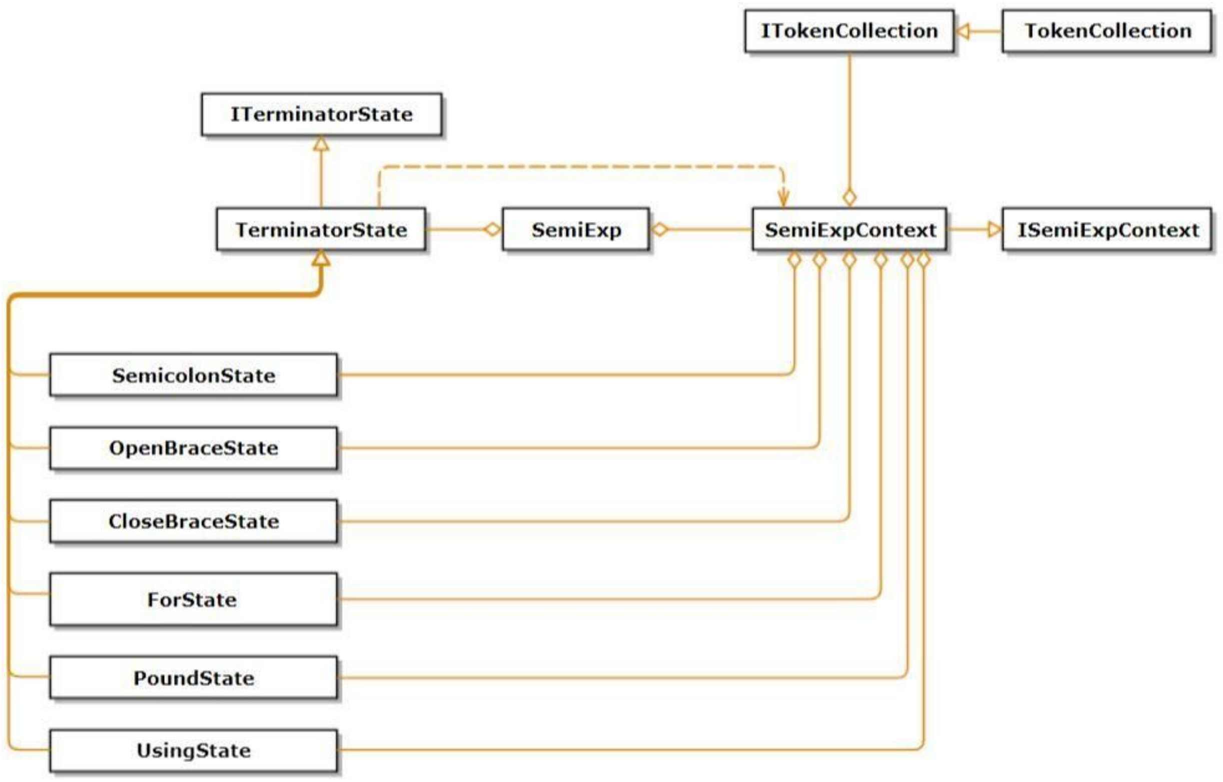


Figure 3: Class Diagram for SemiExpression

<u>**Explanation of the diagram:**</u>

- SemiExp class used to retrieve collections of tokens by calling Toker class functions repeatedly until one of the SemiExpression termination conditions is satisfied.
- The TokenCollection is the class which has the constructs to handle the list of tokens found from the output of Tokenizer.
- ITokenCollection does operations which would be needed to utilize our list of extracted token. It would typically contain functions like get token, remove token and find token.
- ITokenCollection, ITerminatorState and ISemiExpContext are interfaces for their respective classes.
- TerminatorState is the class which is the base class for all the different terminations possible during our expression detection.

4.2.3 Dependency and Strong Component Analysis Class

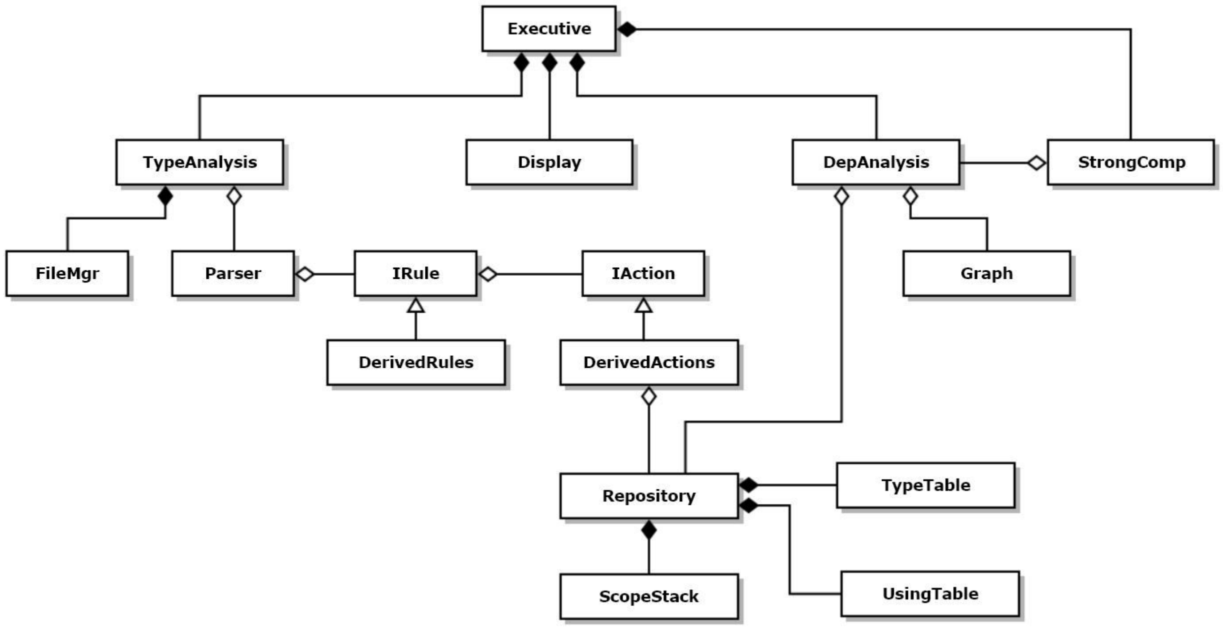


Figure 4: Class Diagram for Dependency and Strong Component Analysis

Explanation of the diagram:

- Once the semi expressions have been generated. The parser now receives them through ITokCollections and then they are checked against the rules and appropriate actions are triggered.
- The Repository structure is needed for creating our TypeTable which also contains a ScopeStack for push, pop functionality and a UsingTable for handling aliases.
- The output of the TypeTable is used by the DepAnalysis for dependency analysis which is in turn used by StrongComp for finding our strongest component along with graph creation using Graph class.

4.2.4 Message Passing Communication Class

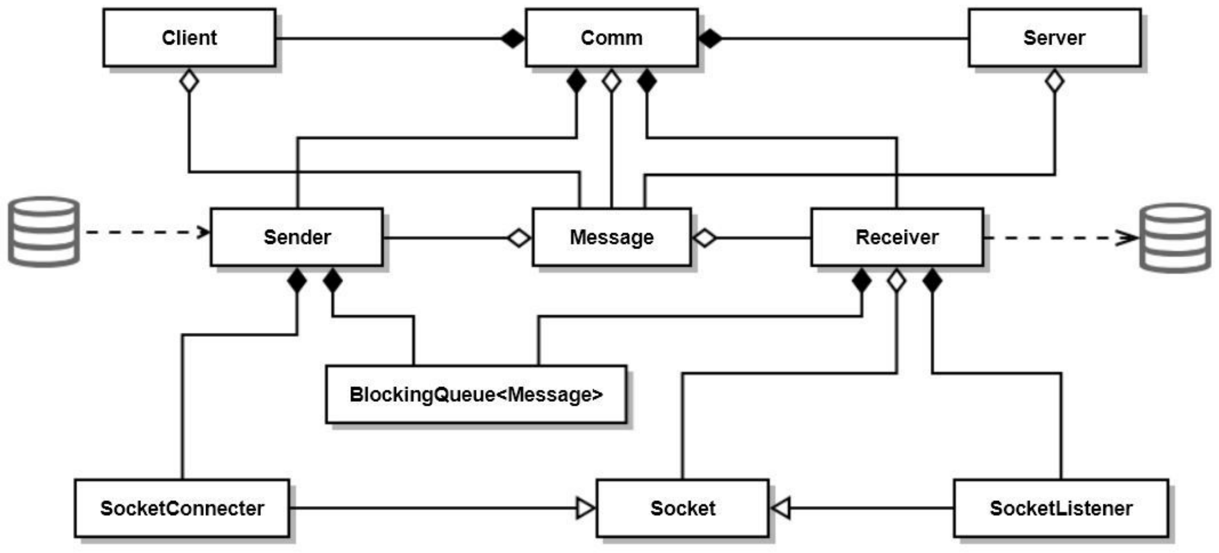


Figure 5: Class Diagram for Message Passing Communication

Explanation of the diagram:

- The client interacts with the server through message passing protocol.
- The client and server will start by listening on particular ports for messages and once it receives the message, both perform actions accordingly.
- The messages are stored on a BlockingQueue which helps us to keep track of the messages.
- This way thee communication channel is set between the client and server.

5. Application Activities

The main functionality of the project is to implement a lexical scanner using a state-based tokenizer. It will detect and perform tasks on recognized lexical patterns in text. The project would read the given input files and scan the source code to find tokens and these tokens would further be used to form expressions based on rules as set by the programmer.

5.1 Token collection

The token collection process is the first step and it starts with getting the source file path as input. Whitespace is set as the starting context of the file and the file is read token wise. Once a token is taken it is identified and the state of the token is set into the context. Another token is read and verified if it belongs to the current state. If it does then it is appended to the previously read token, if not then this current token is read as a new one and the state is changed to the state of the new token. This keeps on going until the end-of-file (EOF) is reached.

Below is the activity diagram for the tokenizer.

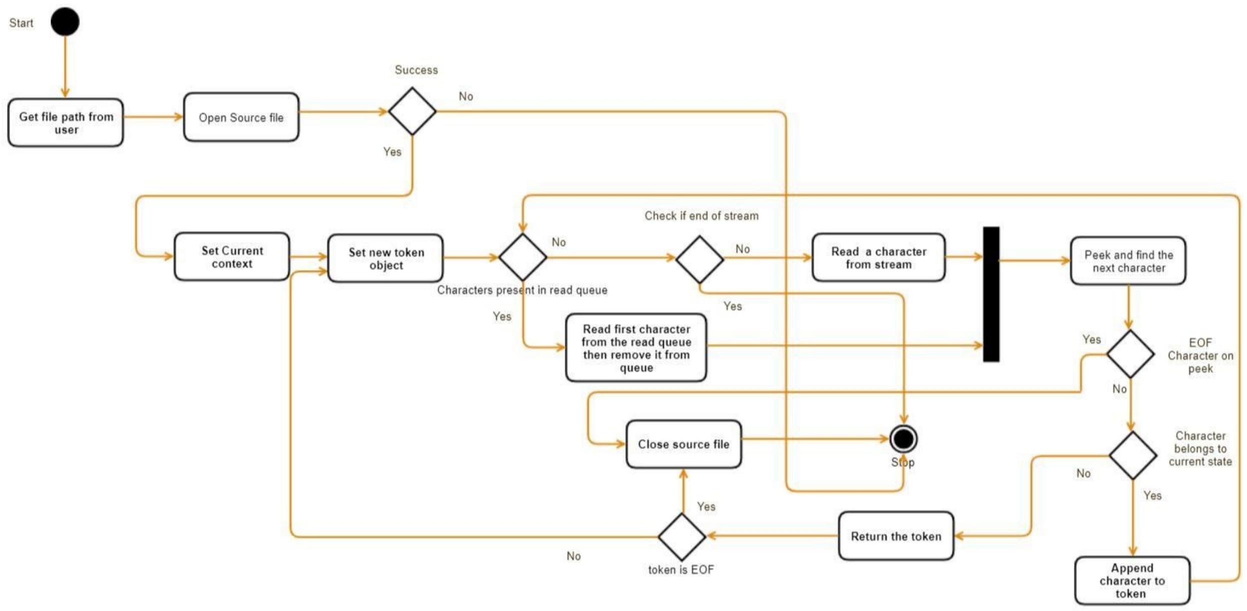

Start
Get file path from user
Open Source file
Success
No
Yes
Set Current context
Set new token object
No
Characters present in read queue
Yes
Check if end of stream
No
Yes
Read a character from stream
Read first character from the read queue then remove it from queue
Peek and find the next character
Yes
EOF Character on peek
No
Close source file
Stop
Character belongs to current state
No
Yes
Return the token
Yes
No
token is EOF
Append character to token

Figure 6: Activity Diagram from Tokenizer

<u>**Explanation of the diagram:**</u>

- First we get the file path from the user and then open the file. Input can also be a string of characters.
- If we are able to open the source file then we move ahead else we quit.
- Next we set the current context state and start our token collection process by setting a new token object
- We always look for the token on the character queue first and only when that is empty we look for tokens on the input stream. When the token is looked for in the queue it is not removed from the stream yet. It is copied on the queue for peeking into the next character which helps us compare if the next token state is same as

the current token state. If yes then we add it to our already read token. Else we return the token and change the state to reflect the state of the new token.

- Thus, characters are removed only when we read the next character from the stream and not when characters are peeked at.
- Peeking also helps us in checking if the next character is EOF character, then we close the file, display the token list to the user and quit the program.
- This way we get our group of tokens according to the seven states that we have defined and thereby getting our output from the tokenizer package which will be the input to the semi expression package.

5.2 Semi expression formation

This is the second step of the process. Once all the tokens have been gathered, they are read one after another and set against rules to check if they form an expression. A common example of this would be that most of the lines of code in C++ would end with a semicolon hence they would form an expression. But other cases need to be handled in the rules like statements beginning with '#' like "#include<stdio>" and comments. There are also a few other exceptional cases like for loops which is an expression containing multiple semicolons.
Below is the activity diagram for semi expression.

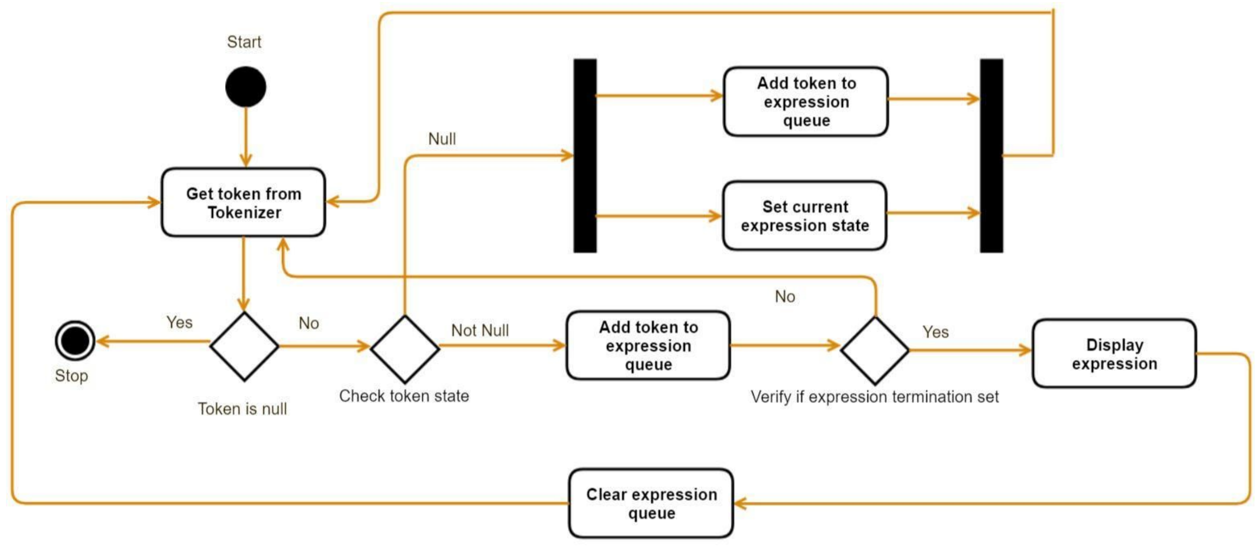


Figure 7: Activity Diagram from SemiExpression

Explanation of the diagram:

- Read a token from the token list and check if it is not equal to null indicating end of token list.
- Once the token is taken check the current state. If it is in SemiColonState it means that we need to look for Semicolon character as the expression terminator. If the token is in PoundState or UsingState, which means the token has started with a "#"" it will not end with semicolon and hence we will be looking for a newline character as the terminator. Another case would be for function and class definitions which would not end with a semicolon. Here we would look for the closed round bracket to find the termination of the token.
- Once a termination condition is hit the expression is formed and displayed and the queue containing it is cleared for the next expression to be detected.
- This way we form expressions as our output to the user.

5.3 Client Server Communication

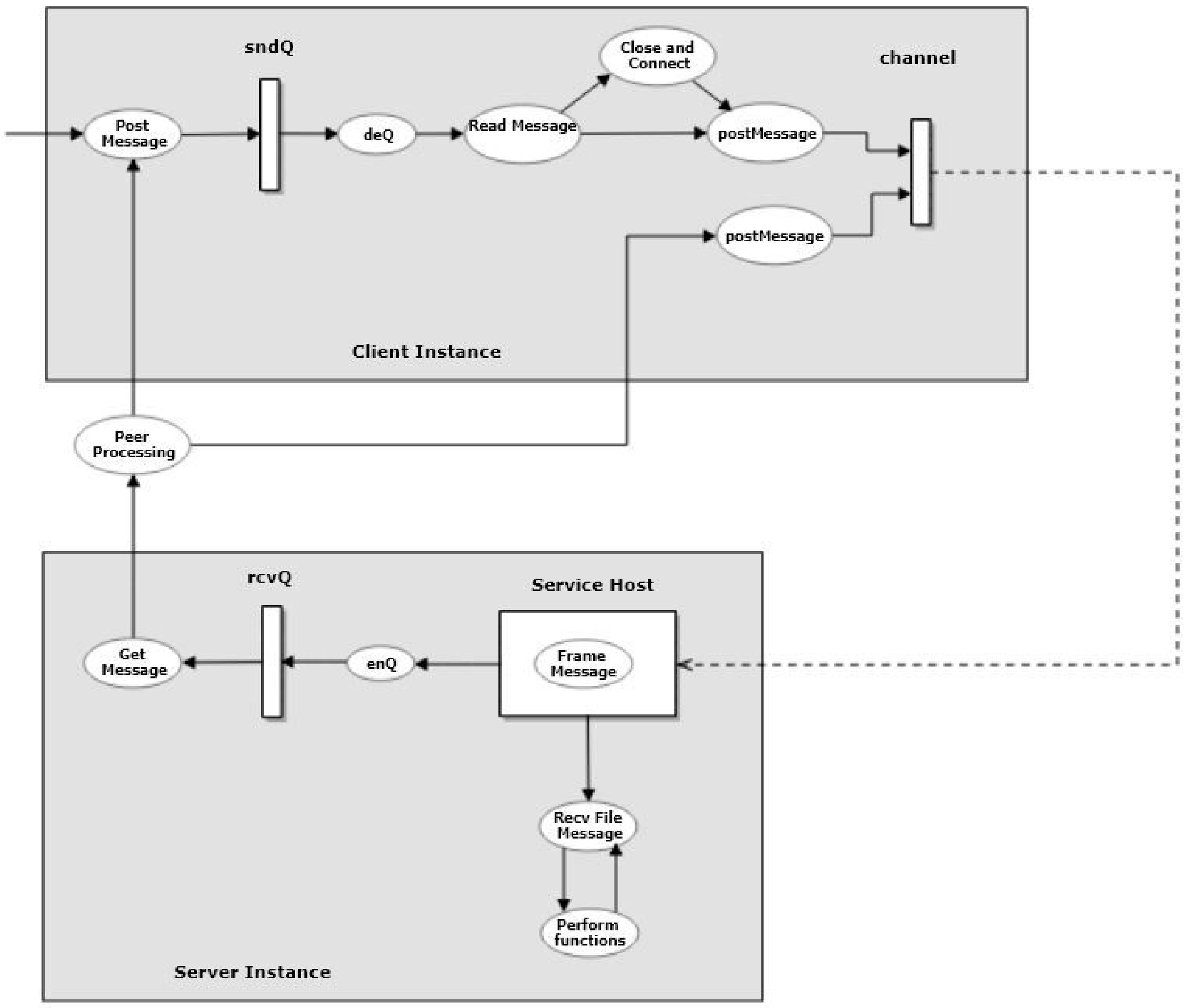


Figure 8: Activity Diagram from Client Server communication

Explanation of the diagram:

-

6. Critical issues

• Issue: Handling many special cases in tokenizer processing, e.g., special tokens, comments, quoted strings could become complicated.
Proposed Solution: Since we are using the State Pattern, we will implement one state for each of the major special cases.

• Issue: Deciding on a next state depends on peeking into the token source stream. Often that requires looking at the next two available characters, but .Net streams do not support peeking at more than the first available character.
Proposed Solution: Build a TokenSource class that uses a .Net stream augmented with a character queue. That will enable peeking at multiple characters without removing them from the Token Source instance, e.g., we read as many characters we need out of the stream, but then enqueue them, so they stay in the source. When we need to pull characters from the stream, we pull first from the queue, until it is empty, then pull from the stream.

• Issue: The parser will use the ItokenCollection interface to access semiExpressions. That avoids binding to concrete details of the Lexer. However, the parser rules need to access almost all of the Semi instance functionality.
Proposed Solution: Provide most of the public interface of Semi in the ItokenCollection interface. That will allow rules and actions to manipulate the token collections effectively.
Maintainability, flexibility, scalability and performance are some of the issues to be concerned with this project.

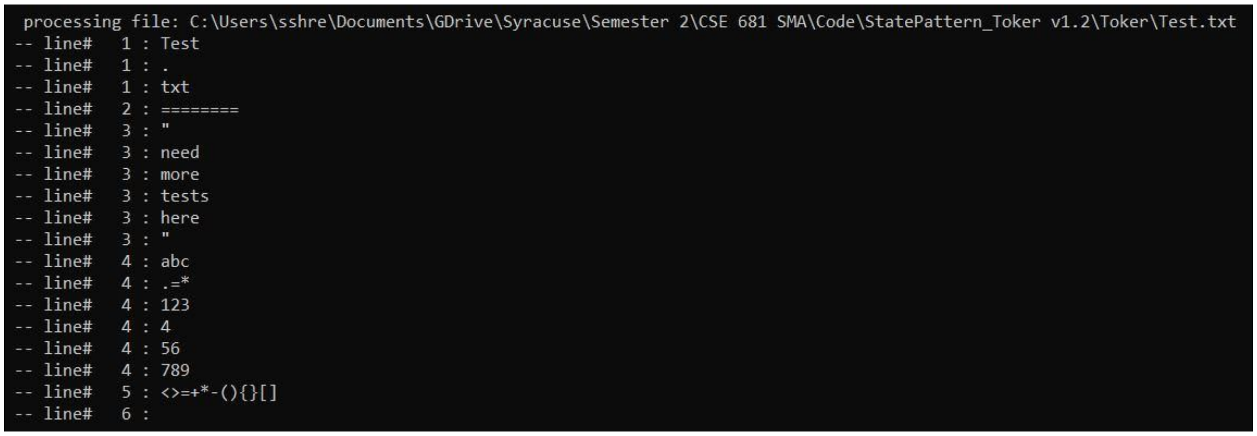
7. Appendix

For the prototype of the lexical scanner we shall use the demo which has been provided beforehand on the site below :
http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/cse681.htm
The demo takes a test.txt file as input and runs the program on the sample test file.
The below file had been passed as the input to the program:

Test.txt

========
"need more tests here"
abc.=*123 4 56 789
<>=+*-(){}[]

The output generated for the above file is:



```
 processing file: C:\Users\sshre\Documents\GDrive\Syracuse\Semester 2\CSE 681 SMA\Code\StatePattern_Toker v1.2\Toker\Test.txt
-- line#   1 : Test
-- line#   1 : .
-- line#   1 : txt
-- line#   2 : ========
-- line#   3 : "
-- line#   3 : need
-- line#   3 : more
-- line#   3 : tests
-- line#   3 : here
-- line#   3 : "
-- line#   4 : abc
-- line#   4 : .=*
-- line#   4 : 123
-- line#   4 : 4
-- line#   4 : 56
-- line#   4 : 789
-- line#   5 : <>=+*-(){}[]
-- line#   6 :
```

Thus we see how the prototype pulls out the tokens based on the three states mentioned - whitespace, alphanumeric and punctuation marks. We can see that the quote token has not been implemented since the entire text inside the double quotes and the text inside it should have been one token.

8. References

1) https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project3HelpF2018/
2) https://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project4HelpF2018/
3) ftp://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html
4)https://medium.com/dailyjs/gentle-introduction-into-compilers-part-1-lexical-analysis-and-scanner-733246be6738
5) https://en.wikipedia.org/wiki/Lexical_analysis
6) https://en.wikipedia.org/wiki/Compiler
7) http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/cse681.htm

8) https://blog.back4app.com/2017/05/26/scaling-parse-server/
9) http://users.csc.calpoly.edu/~jdalbey/101/Resources/codetrace.html