Compiler Construction

**Group Members:**

Muhammad Rehman Rabbani (137364)

Abdul Ghaffar Kalhoro ()

Ahmad Amjad Mughal (121672)

Hamad Nasir (120312)

**Instructor:**

Mr. Danial Ahmad

Construction of a Compiler using Java Language

Semester Lab Project

[Year]

**Introduction**

Compiler consists of both front end and back end stages. Front end stages comprise of Scanner, Parser, Standard Syntax Definition and Code Generation. Scanner accepts a high-level language code and returns a set of generated tokens. Then comes parser which makes parse tree by taking tokens as input. Stand Syntax Definitions performs semantic analysis on the selected code to understand what type of meaning this line conveys. Code Generation is last phase in which we either make Directed Acyclic Graph or Three Address code.

There are various operations a specific program code needs to go through.

1. Lexical Analyzer
2. Syntax Analyzer (Parsing)
3. Semantic Analyzer

**Lexical Analyzer (Hamad Nasir’s part)**

He has implemented lexical analysis that do analysis on code and it returns a set of tokens and it also ignores commenting and white spacing in this regard. The tokens are generated in a defined way (Token id, Lexeme). He tries to deal with every possible stream of character like identifiers, integers literals etc.

\* Letter: [a-zA-Z]

\* Digit: [0-9]

\* Identifier: letter (letter | digit)\*

\* Integer: digit+

\* Float: Integer . Integer

\* Boolean: true | false

\* Char: â€˜ASCII Charâ€˜

\* Literal: Integer | Boolean | Float | Char

\* ReservedWords: main | int | float | char| boolean | if | else | true | false | false | while

\* AssignmentOperator: =

\* EqualOperator: ==

\* NotEqualOperator: !=

\* RelationalOperator: <|<=|>|>=

\* ArithmeticOperator: + | - | \* | / | %

\* UnaryOperator: - | !

\* BooleanOperator: && | ||

\* Operator: EqualOperator| NotEqualOperator | RelationalOperator | ArithmeticOperator| UnaryOperator | BooleanOperator | AssignmentOperator

\* RightBrace: }

\* LeftBrace: {

\* RightParenthese: )

\* LeftParenthese: (

\* RightBracket: ]

\* LeftBracket: [

\* Semicolon: ;

\* Comma: ,

**Syntax Analyzer (Muhammad Rehman Rabbani’s part)**

Syntax Analyzer checks the syntax of each line of code. It tells whether the function definition is valid or not, whether declarations follow the right syntax or not, statements, check, loops, expressions follows the correct definition or not, Operators are used correctly or not. He implemented the syntax analyzer in a way so to deal with every possible syntax error that can be occur while parsing. Abstract Syntax Tree is generated as output along with no. of errors reported if any. The nodes of AST need to follow semantic check.

\* Program ::= int main () { Declarations Statements }

\* Declarations ::= { Declaration }

\* Declaration ::= Type Identifier [ [ Integer ] ] { , Identifier [ [ Integer ] ] }

\* Type ::= int | bool | float | char

\* Statements ::= { Statement }

\* Statement ::= ; | Block | Assignment | IfStatement | WhileStatement

\* Block ::= { Statements }

\* Assignment ::= Identifier [ [ Expression ] ] = Expression;

\* IfStatement ::= if ( Expression ) Statement [ else Statement ]

\* WhileStatement ::= while ( Expression ) Statement

\* Expression ::= Conjunction { || Conjunction }

\* Conjunction ::= Equality { && Equality }

\* Equality ::= Relation [ EquOp Relation ]

\* EquOp ::= == | !=

\* Relation ::= Addition [ RelOp Addition]

\* RelOp ::= < | <= | > | >=

\* Addition ::= Term { AddOp Term }

\* AddOp ::= + | -

\* Term ::= Factor { MulOp Factor }

\* MulOp ::= \* | / | %

\* Factor ::= [ UnaryOp ] Primary

\* UnaryOP ::= - | !

\* Primary ::= Identifier [ [Expression] ] | Literal | ( Expression ) | Type ( Expression)

\* Identifier ::= Letter { Letter | Digit }

\* Letter ::= a | b | â€¦ | z | A | B | â€¦ | Z

\* Digit ::= 0 | 1 | â€¦ | 9

\* Literal ::= Integer | Boolean | Float | Char

\* Integer ::= Digit { Digit }

\* Boolean ::= true | false

\* Float ::= Integer.Integer

\* Char ::= 'ASCIIChar'

**Semantic Analyzer (Ahmad Amjad Mughal’s part)**

Sematic Analyzer performs semantic checking of each Abstract Syntax Tree node. It can either be expression or simply a print statement. It tells what that line of code conveys meaning. Actual meaning is returned as output of semantic analyzer. It also looks for type checking, identifier checking, multiple line declarations, typecasting, already exist identifier, reference \* All referenced variables must be declared. We do the following checks.

\* All declared variables must have unique names.

\* ##### Rule two:

An Assignment is valid if:

1. Its target Variable is declared.

2. Its source Expression is valid.

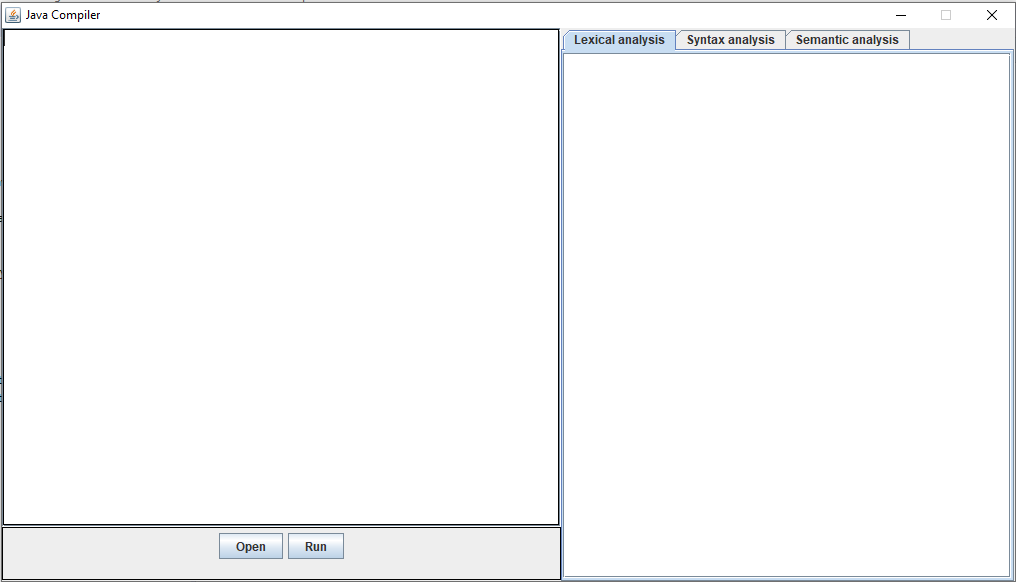
3. If the target Variable is float, then the type of the source Expression must be either float or int.

4. Otherwise if the target Variable is int, then the type of the source Expression must be either int or char.

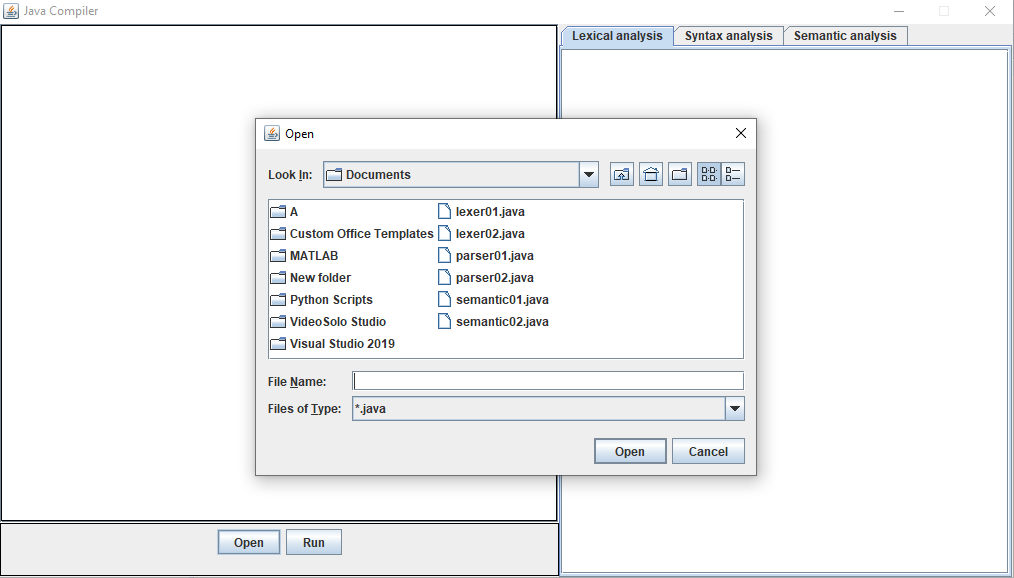
5. Otherwise the target Variable must have the same type as the source Expression. variables declarations. Various functions are defined for each checking.

**GUI (Abdul Ghaffar Kalhoro’s part)**

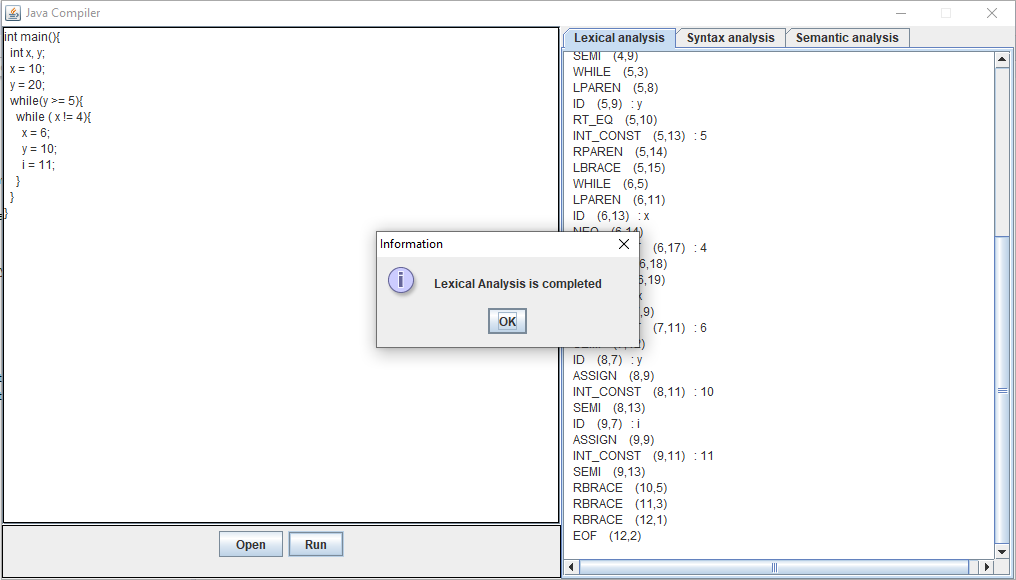
Then comes the interactive interface through which user have option to perform either lexical analysis or parsing or semantic. Each is assigned a separate button and a section shows an output of each phase.



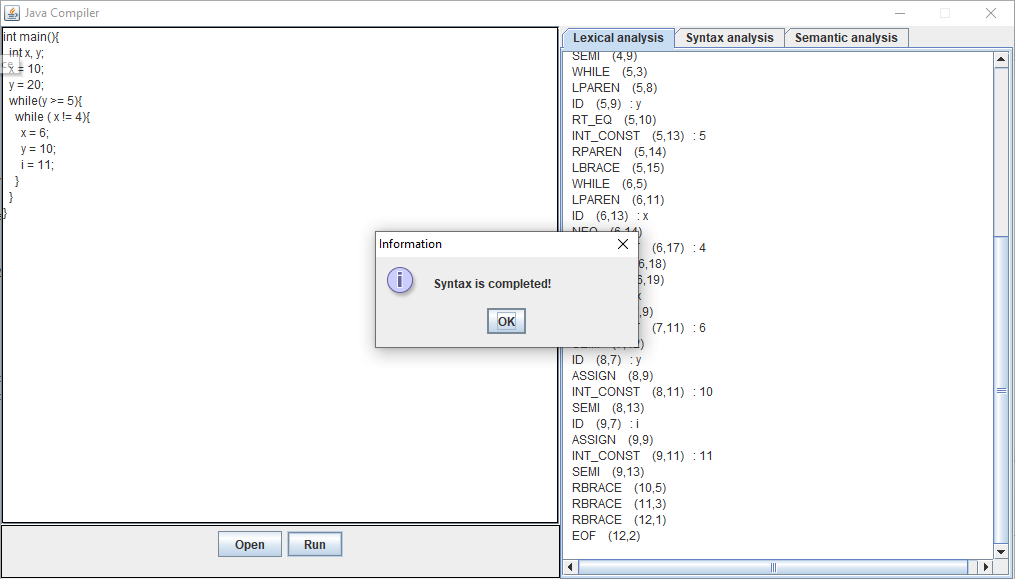
**Output**

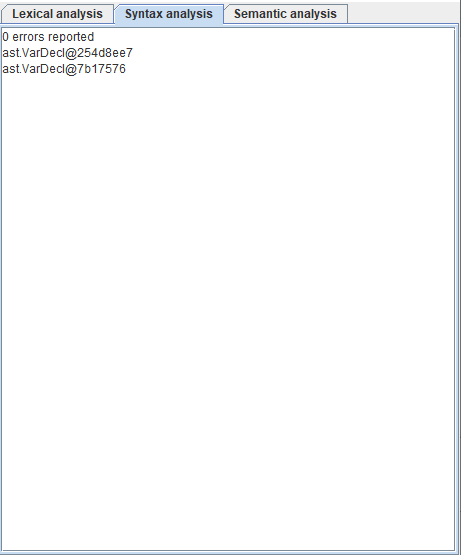
****

**Lexical Analysis**

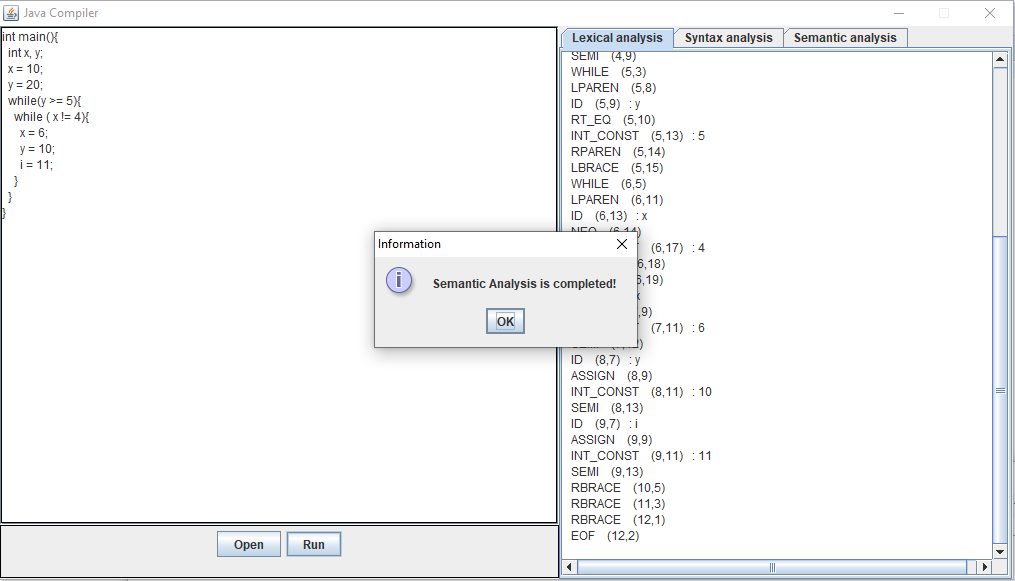
****

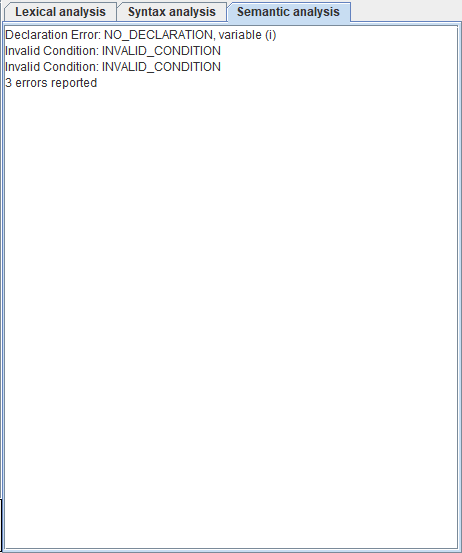
**Syntax Analysis**

****

****

**Semantic Analysis**

****

****