References:

- Compilers : Principles,Techniques and Tools by Alfred V.Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ulman. https://www-2.dc.uba.ar/staff/becher/dragon.pdf
- Modern Compiler Implementation in C by Andrew W.Appel
- Modern Compiler Implementation in JAVA by Andrew W.Appel
- lex & yacc, 2nd Edition by Doug Brown, John Levine, Tony Mason
- Flex & Bison by John Levine
- http://dinosaur.compilertools.net/
- https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgg/index.html
- https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgt/index.html
- https://www.ibm.com/docs/en/zos/2.4.0?topic=lex-input-language
- https://silcnitc.github.io/lex.html
- https://web.stanford.edu/class/cs143/
- https://westes.github.io/flex/manual/
- https://www.gnu.org/software/bison/manual/html_node/index.html
- https://arcb.csc.ncsu.edu/~mueller/codeopt/codeopt00/y_man.pdf
- https://nxmnpg.lemoda.net/1/lex
- https://nptel.ac.in/courses/106104123

Tasks:

- Create each phases of a compiler for your programming language
    - **Lexical Analyzer**
        - **Regular Expressions for your language, Actions, Tokens**
        - **Handling of Errors**
        - **Symbol Table**
    - **Parser**
        - **Grammar**
        - **States**
        - **Transition Diagram**
        - **Parse Table of LALR parser**
        - **Synch for error recovery**
    - **Semantic Analyzer**
        - **Type checking and information addition to symbol table**
        - **Removing ambiguity in operators**
        - **variable/function declaration, definition mismatch checking**
        - **evaluation expressions**
        - **syntax tree**
        - **handling arrays**
        - **attributes and SDT**
    - Intermediate Code Generation
    - Code Optimization
    - Target Code

- Write a sample source program for calculator in the language you developed and compile the program by your compiler .

- Write test programs to check every statement of the compiler and show that it is working correctly.

Assignment 1: More detail

A. Develop the *components* of a programming language having all features similar to C. Your keywords should end with '_' followed by the initials of your name and each identifier should start with the last three digits of your roll number.
   a. Must have Keywords for Loop, Switch Case, If-Else, type of variables/numbers, structure
   b. Operators
   c. punctuations
   d. (,),{,},[,]
   e. identifiers, numbers, strings

B. Write regular expressions for each of them and draw the corresponding DFA

C. Write Lex code implementing the patterns and corresponding actions.

D. Write codes for handling errors during lexical analysis.

E. Compile the Lex code and create your own lexical analyzer (L).

F. Write a sample source program for a scientific calculator in the language you developed and do the following:
   a. Show that L is able to correctly recognize the tokens and handle errors correctly.
   b. Show output tokens in the print statement.
   c. Show the contents of your Symbol table after each token is processed.
   d. Write small programs in the language you have developed.

G. Further test with other sample programs in this new language to check every statement of the compiler and show that it is working correctly.
   a. Write sample program to do linear search and binary search
   b. Write sample program to implement any sorting technique
   c. Write programs containing array, functions, switch cases, if-else statements and loops.

Assignment 2:

A. Create a parser that can handle all the components of this programming language.
   a. Write the production rules of your grammar.
   b. Remove ambiguity using precedence and associativity.
   c. Build the state-automata and the parse table.
   d. Do error recovery using Synch symbols.

B. Show that this parser correctly parses the input token generated by your lexical analyser for the programs written in your programming language as well as identifies errors.
   a. Parse the programs written in Assignment 1 and show that your compiler is correctly detecting the tokens and report errors.
   b. Parse the program using your parser. Print step by step parsing process and draw the parse tree.

Assignment 3:

1. Generate syntax-directed translations for your grammar such that it does the following semantic checks:

    a. Declaration and definition: Whether a variable has been declared? Are there variables that have not been declared? What declaration of the variable does each reference use? Are all invocations of a function consistent with the declaration?

    b. Type: What is the type of the variable? Whether a variable is a scalar, an array, or a function? Is an expression type consistent? Add type information in the Symbol table

    c. Array: Is the use of an array like A[i,j,k] consistent with the declaration?

    d. Overloading: remove ambiguity. If an operator/function is overloaded, which function is being invoked?

2. What kind of attributes are you using? Is the grammar L-attributed or S-attributed. Write the corresponding semantic rules and write the appropriate actions.
3. Evaluate the expression of your calculator program using semantic rules.
4. Create a syntax tree using semantic rules for your input programs created in Assignments 1&2.
5. Show that your compiler (developed so far) can detect semantic errors which were not detected up to the Syntax Analysis phase.

Assignment 4:

1. Write syntax-directed translations to generate proper Three-Address Codes for your grammar. You need to generate the code by a maximum of two passes over the syntax tree:
    a. Add semantic rules to handle different types and declarations
    b. Considering your source program allows operations with type mismatch, generate semantic rules to
    c. Develop three-address codes considering you have expressions with arithmetic, logical and relational operators, arrays, if-block, if-else-block, loops, switch-cases, procedures and functions.
2. Apply the concept of backpatching to give the destination address of jump statements correctly so that the parsing, semantic analysis and intermediate code generation is done together in a single pass.
3. Show that for any types of arbitrary blocks of codes containing all types of statements (expressions, control flow, relational operator, array, function call, etc.) written in your programming language, your compiler is able to generate the appropriate three address codes.
4. Draw the annotated parse trees with the semantic translations and DAGs for the sample programs.