

The final Project Submission Date is **March 2nd, 2026 (Monday, till 8:00 p.m.)**

Language specifications are available in the attached file. The students are advised to read the details of constructs, their related patterns and their equivalent structures precisely. I will be available in my office (Tuesday and Friday) for doubts and discussions from 04:00 PM to 05:00 PM. The language supports the following features:

a) lexical patterns (Assignment operator, Comma, Colon, etc.)

b) Expressions (Arithmetic, Boolean)

c) Data Types (Integer type, Real type, Record type, Union Type, Type definition (Aliases))

d) Statements (Assignment Statement, I/P O/P Statements, Declaration Statement, Return Statement, Iterative Statements, Function Call Statements)

e) Variable identifiers, function identifiers, record identifiers, record field identifiers,

f) Function call statements, function input parameters, function output parameters

g) integer and real numbers.

Important Points:

- It is advisable that don't just start coding randomly.
- You must first understand the constructs of the language, identify their patterns and understand their structure.
- Before actual implementation of lexical analyzer, you have to do some paper work related to the construction of design of your transition diagram (DFA).
- Similarly, before going for the actual implementation of syntax analyzer, you need to understand the grammar rules and identify the discrepancies in the highlighted grammar rules. These discrepancies are left intentional in the language specification document to make you go through the constructs of the language and their structure properly.
- The implementation aspects of the project will now be discussed in the upcoming lectures. Hence, it is advisable that please don't miss any lecture.
- More details related to the Group number assignment, implementation, paper work, test cases, errata (if any), etc. will be regularly uploaded on the Nalanda.

Modules of the project:

- **Understanding the language features**

- **Constructing hand-drawn DFA for lexer**
 - **Lexical Analysis Module Development, Testing of lexer with given test cases and self-created testcases**
 - **Grammar Modifications, Grammar Representation (for Parser)**
 - **FIRST and FOLLOW sets automation**
 - **Predictive Parsing Table Creation**
 - **Testing the table (by printing)**
 - **Stack ADT and Parsing**
 - **Tree ADT**
 - **Parse tree creation while parsing**
 - **Testing (with given test cases), Documentation of the code (using comments)**
 - **Testing Error reporting, Parse tree printing etc.**
-

Lexical Analyzer: Develop the lexical analyzer module of compiler for implementing the given language. This module takes as input the file containing user source code written in the given language and produces the tokens. The lexical analyzer module scans the input only once and collects all relevant information required by the other modules of compiler. The lexical analyzer ignores comments and white spaces, while recognizes the useful lexemes as valid tokens. The lexical errors are reported by this module when it sees any symbol or pattern not belonging to the language. Your lexical analyzer must

1. Tokenize lexemes appropriately
2. Maintain all information collected during a single pass of the source code
3. Be efficient with respect to time and space complexity
4. Report lexical errors with line numbers appropriately

Syntax Analyzer: This module takes as input the token stream from the lexical analyzer module and verifies syntactic correctness of the source code. This uses predictive parser (using parsing table) to establish the syntactic structure of the source code. As the parser sees next token, verifies its correctness, it uses the token information to build a tree node and inserts appropriately in the parse tree corresponding to the input source code. If the source code (in given language) is syntactically correct, a corresponding parse tree is produced as the output. If the input is syntactically wrong, errors are reported appropriately. Your syntax analyzer (Parser) must

1. Ensure time and space efficiency and use a single pass of the token stream
2. Use predictive parser using parsing table
3. Produce as output the parse tree, if the source code is syntactically correct

4. Produce a list of syntax errors with appropriate messages and line numbers

Implementation Details

- Use C language (Linux/ Ubuntu based GCC) to implement the modules.
- Use of any other high level language or lexer/ parser generator packages is NOT allowed.
- Test your code with the test cases given in the language specification document.
- Generate more test cases and verify the correctness of your code.
- You will be given more test cases later.
- An appropriate interface support will be provided to you as you are through with the ground work.
- Instead of starting coding, first spend time in designing the structure of your compiler code.

Use **GCC version 13.3.0** and **Ubuntu 24.04.2 LTS** for compiler project code development.

Ensure that your code is compatible to GCC and Ubuntu version as specified above. Teams are advised to design data structures for token info, grammar, parse table, parse tree first and follow sets etc. and use names in self-explanatory form. Following are the suggested prototypes for better understanding of the implementation needs and are provided as support. However, the teams will have the flexibility to select the prototypes, parameters etc. appropriately.

1. File **lexer.c** : This file contains following functions

FILE *getStream(FILE *fp): This function takes the input from the file pointed to by 'fp'. This file is the source code written in the given language. The function uses an efficient technique to populate twin buffer by bringing the fixed sized piece of source code into the memory for processing so as to avoid intensive I/O operations mixed with CPU-intensive tasks. The function also maintains the file pointer after every access so that it can get more data into the memory on demand. The implementation can also be combined with getNextToken() implementation as per the convenience of the team.

tokenInfo getNextToken(twinBuffer B): This function reads the input character stream and uses efficient mechanism to recognize lexemes. The function tokenizes the lexeme appropriately and returns all relevant information it collects in this phase ([lexical analysis](#) phase) encapsulated as tokenInfo. The function also displays lexical errors appropriately.

removeComments(char *testCaseFile, char *cleanFile): This function is an additional plugin to clean the source code by removal of comments. The function takes as input the source code and writes the clean code in the file appropriately. [Note: This function is invoked only once through your driver file to showcase the comment removal for evaluation purpose. However, your lexer does not really pick inputs from the comment removed file. Rather, it keeps ignoring the comments and keep collecting token Info to pass to the parser. For showcasing your lexers ability, directly take input from user source code]

2. File parser.c : This file contains following functions

FirstAndFollowComputeFirstAndFollowSets (grammar G): This function takes as input the grammar G, computes FIRST and FOLLOW information and populates appropriate data structure FirstAndFollow. First and Follow set automation must be attempted, keeping in view the programming confidence of the team members and the available time with the teams. If teams opt not to develop the module for computation of First and follow sets, the same can be computed manually and information be populated in the data structure appropriately. However, all members of the team must understand that any new grammar rule for any new construct will then require their expertise in computing FIRST and FOLLOW sets manually.

createParseTable(FirstAndFollow F, table T): This function takes as input the FIRST and FOLLOW information in F to populate the table T appropriately.

parseInputSourceCode(char *testCaseFile, table T): This function takes as input the source code file and parses using the rules as per the predictive parse table T and returns a parse tree. The function gets the tokens using [lexical analysis](#) interface and establishes the syntactic structure of the input source code using rules in T. The function must report all errors appropriately (with line numbers) if the source code is syntactically incorrect. If the source code is correct then the token and all its relevant information is added to the parse tree. The start symbol of the grammar is the root of the parse tree and the tree grows as the syntax analysis moves in top down way. The function must display a message "Input source code is syntactically correct....." for successful parsing.

printParseTree(parseTree PT, char *outfile): This function provides an interface for observing the correctness of the creation of parse tree. The function prints the parse tree in inorder in the file outfile. The output is such that each line of the file outfile must contain the information corresponding to the currently visited node of the parse tree in the following format

lexeme CurrentNode lineno tokenName valueIfNumber
parentNodeSymbol isLeafNode(yes/no) NodeSymbol

The lexeme of the current node is printed when it is the leaf node else a dummy string of characters "----" is printed. The line number is one of the information collected by the lexical analyzer during single pass of the source code. The token name corresponding to the current node is printed third. If the lexeme is an integer or real number, then its value computed by the lexical analyzer should be printed at the fourth place. Print the grammar symbol (non-terminal symbol) of the parent node of the currently visited node appropriately at fifth place (for the root node print ROOT for parent symbol) . The sixth column is for printing yes or no appropriately. Print the non-terminal symbol of the node being currently visited at the 7th place, if the node is not the leaf node [Print the actual non-terminal symbol and not the enumerated values for the non-terminal]. Ensure appropriate justification so that the columns appear neat and straight.

Description of other files

lexerDef.h : Contains all data definitions used in lexer.c

lexer.h : Contains function prototype declarations of functions in lexer.c

parserDef.h : Contains all definitions for data types such as grammar, table, parseTree etc. used in parser.c

parser.h : Contains function prototype declarations of functions in parser.c

driver.c : As usual, drives the flow of execution to solve the given problem. (more details, if needed, will be uploaded soon)

makefile : This file uses GNU make utility, which determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them. The correctness of your make file depends on file dependencies used correctly.

NOTE:

1. A file using definitions and functions from other files must include interface files appropriately. For example parser.c uses functions of lexer.c, so lexer.h should be included in parser.c. Do not include lexer.h in lexer.c, as lexer.c already has its own function details. Also keep data definitions in files separate from the files containing function prototypes. In case of doubts, meet me and clarify your doubts. It is essential to place the contents in appropriate files and have correct set of files.

2. Use of any high level library other than standard C library is strictly not allowed.

Your driver must have the following choices. Press option for the defined task (Use a while loop to receive option choices till option 0 is pressed. Ensure independence of working of all options e.g. if option 3 is pressed, option 2 is not needed)

0 : For exit

1 : For removal of comments - print the comment free code on the console

2 : For printing the token list (on the console) generated by the lexer. This option performs [lexical analysis](#) and prints all tokens and lexemes line number wise. Here, the tokens are not passed to the parser, but printed on the console only. Each token appears in a new line along with the corresponding lexeme and line number. (invoke only lexer) [Ensure pretty printing with column justifications to increase readability]

3 : For parsing to verify the syntactic correctness of the input source code and printing the parse tree appropriately. This option prints all errors - lexical and syntactic, line number wise, on the console and prints parse tree in the file as mentioned in the command line below. (Invoke both lexer and parser).

4: For printing (on the console) the total time taken by your project code of lexer and parser to verify the syntactic correctness. Use <time.h> file as follows

```
-----  
#include <time.h>  
  
clock_t start_time, end_time;  
  
double total_CPU_time, total_CPU_time_in_seconds;  
  
start_time = clock();  
  
// invoke your lexer and parser here  
  
end_time = clock();  
  
total_CPU_time = (double) (end_time - start_time);  
  
total_CPU_time_in_seconds = total_CPU_time / CLOCKS_PER_SEC;  
  
// Print both total_CPU_time and total_CPU_time_in_seconds  
-----
```

Perform actions appropriately by invoking appropriate functions.

Also during its execution, the driver displays necessary information regarding the implementation status of your work at the beginning on the console such as

- (a) FIRST and FOLLOW set automated
- (b) Only Lexical analyzer module developed
- (c) Both lexical and syntax analysis modules implemented
- (d) modules compile but give segmentation fault
- (e) modules work with testcases 2, 3 and 4 only
- (f) parse tree could not be constructed

and so, on whichever is applicable.

Compilation:

The name of the make file should be makefile only as I will avoid using -f option always to make your file named something else (that includes searching for the file which is time taking). You can find documentation at the GNU website where you can learn how to write a make file (<http://www.gnu.org/software/make/manual/make.html>).

Please ensure compatibility with the GCC specifications provided in the recent notice.

Execution

The command line argument for execution of the driver should be as follows, for example

```
./stage1exe testcase.txt parsetreeOutFile.txt
```

where stage1exe is the executable file generated after linking all the files (your makefile should be absolutely correct). The file testcase.txt is the source code file in the given language to be analyzed and parsetreeOutFile.txt is the file containing parse tree printed as per the format specified earlier.

The inorder traversal for an n-ary tree can be seen as follows

Leftmost child --> parent node--> remaining siblings (excluding the leftmost)