Interface Requirements *[posted on March 04, 2013]*

Implement stage 1 modules as per the following details

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

- **FILE *getStream(FILE *fp, buffer B, buffersize k):** 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 efficient technique to bring the fixed sized piece (of size k) of source code into the buffer B 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.
- **tokenInfo getNextToken(FILE *fp):** 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 (list of errors to be uploaded soon). The input parameters are made flexible and students can select appropriate input parameters. The function also takes care of ignoring the white spaces and comments.

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

i. **void createParseTable(grammar G, table T):** This function takes as input the grammar G, uses FIRST and FOLLOW information to populate the table T appropriately.

ii. **parseTree 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. 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 errors as per the list of error specifications (will be provided to you soon) 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 " Compiled Successfully: Input source code is syntactically correct!!" for successful parsing.

iii. **void printParseTree(parseTree PT, char *outfile):** function provides an interface for observing the correctness of the creation of parse tree. The function prints the parse tree in **depth first order** 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 **lexemeCurrentNode lineno token 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 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 nonterminal symbol of the node being currently visited at the 7th place, if the node is not the leaf node [Print the actual nonterminal symbol and not the enumerated values for the nonterminal].** Ensure appropriate justification so that the columns appear neat and straight.

iv. **void createAbstractSyntaxtree(parseTree T, abstractSyntaxTree A):** This function takes as input the parse tree T and traverses that in appropriate order to generate the abstract syntax tree A. You should write all semantic rules to generate the nodes of the AST corresponding to each production of the original grammar G. Name this file as AST_rules.txt.

v. **void printAST(abstractSyntaxTree A, char *outFile, int *totalAllocatedMemory):** The order of traversal of the AST again is in **depth first order** and prints the AST in the file outFile. Print the non leaf nodes and leaf nodes appropriately. The credit will be given to the better output representation of the AST. This function also computes the total allocated memory to the abstract syntax tree A as it traverses the complete tree A. Remember that the size of the nodes of A may vary depending upon the production of G which created it.

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)

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. [if files are not appropriately used and contents are not placed in appropriate files, this may lead to a penalty of 5 marks for consuming my extra time]

2. Make a folder as batch# (e.g. batch52, batch07, batch11, batch01 etc for batches 52, 7, 11 and 1 respectively). Use lower case letters only for batch, followed by the batch number. Do not use blank or underscore (batch_5 or Batch5 or batch 5 will

be wrong)

3. Keep all your files in the above folder. No subdirectory must exist within batch folder. All test cases, makefile and driver file alongwith all implementation and interface files should reside in batch folder.

4. Each file must contain team members names and IDs at the top as comments.

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