### **Compiler project: Language specifications**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 2 February 2023, 8:01 PM

Number of replies: 0

Language specifications are available in the attached file. The students are advised to read the details and discuss if they have any doubts.

**Important Dates:**

* Stage 1 submission : **March 2, 2023 (8:00 p.m.)**
* Stage 2 submission: **April 12, 2023 (tentative)**
* Online exam: **April 23, 2023**

### **Compiler project: Team formation, important dates and ground work**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 4 February 2023, 9:17 AM

**Team Formation:**Form teams of 3-5 students on your own and register team members' names on February 6, 2023, Monday, from 4:30 p.m. to 7:30 p.m. A link will be made available on Nalanda for registering the team members names and IDs. Presence of all members is advised at the time of registration. Registering together as a team, the team members display confidence on each other and are expected to work on the compiler project together. The team members will not be changed in any circumstances throughout the semester till completion of the compiler project.

**Important Dates (Stage 1)**

Date of posting language specifications: **February 2, 2023**

Team formation: **February 6, 2023** [During 4:30 p.m. to 7:30 p.m]

***Paper submissions***

DFA neatly drawn on A3 sheet: **February 12, 2023** [During 6:30 p.m. to 7:30 p.m., Venue 6121-Z]

Complete and Modified grammar on A4 sheet: **February 19, 2023**[During 6:30 p.m. to 7:30 p.m., Venue 6121-Z]

FIRST & FOLLOW sets on A4 sheet:  **February 19, 2023**[During 6:30 p.m. to 7:30 p.m., Venue 6121-Z]

***Submissions through Nalanda***

Coding Details in given proforma: **March 2, 2023**[During 6:30 p.m. to 8:00 p.m.]

Code Submission Due on :  **March 2, 2023**[During 6:30 p.m. to 8:00 p.m.]

***Ground Work***

1. Read the language specifications document carefully.
2. Understand patterns and tokens.
3. Construct DFA based transition diagram (on paper) for recognizing the patterns in the source code.
4. Understand the features of the language and the grammar.
5. Complete the set of production rules to incorporate all constructs of the given language.
6. Next two weeks lectures will be on parsing techniques. Students are advised to attend classes regularly and discuss their doubts.
7. As we will progress with parsing details in the class, students can keep on preparing themselves to do the following groundwork.
8. Remove ambiguity, left recursion and perform left factoring for the productions that need to be modified.
9. Make the grammar LL(1) compatible.
10. Compute FIRST and FOLLOW sets. Verify any conflict due to ambiguity persisting in the grammar.
11. Design suitable and efficient data structure for representing and processing the language.
12. Implement lexical analyzer first and test the tokens generated.
13. Use a temporary driver to test lexical analyzer.
14. Design representation of the grammar appropriately. [ To Avoid hard coding of rules, keep the grammar in one file and populate the data structure representing the grammar.]
15. Once your team members put efforts in designing the grammar rules, and submit after best efforts, you will be given support of modified grammar to correct your own grammar where ever needed.
16. Do not start coding randomly. First design the functionalities of your compiler front end and verify correctness of your DFA and grammar, then implement gradually your code.
17. Once you pour in some creativity in the design of your functionalities, you will be given support of function prototypes.
18. There will be a flexibility in the usage of these prototypes which can be changed based on your design.
19. The Linux/ Ubuntu based GCC version will be specified for the C programming language you will use for your code within a few days. Do not use just any C compiler on your own. Eventually the versions of Linux/ Ubuntu and GCC will have to be the same as those used in IPC labs.
20. Revise your concepts regularly by reading your text book chapters.
21. Feel free to fix up a time with me for a meeting whenever there is any doubt regarding design and implementation of the compiler.
22. As senior undergraduate Computer Science students, you are expected to be adaptive for making changes in your code anytime as you advance with newer concepts in compiler construction. Various data structures will be required to be redefined or modified as we progress.
23. I will conduct few special sessions on implementation issues in which we will discuss complexity of data structure design and algorithms.

### **Late registration of teams**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Friday, 10 February 2023, 10:15 AM

The students who missed the team registration deadline earlier are given an opportunity to register the team details at the link given on the course page during 4:30 p.m. to 7:30 p.m. today. It is advised that the teams which have already registered must not register again.

Students are advised to follow the deadlines strictly throughout the semester as no exemptions in this regard will be considered hereafter.

### **Stage 1: Modules**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Friday, 10 February 2023, 9:04 PM

Develop the following modules of the compiler for implementing the given language.

**Lexical Analyzer:** This module takes as input the file containing user source code 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 the compiler. The lexical analyzer ignores comments and white spaces while recognizing 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

* Tokenize lexemes appropriately
* Maintain all information collected during a single pass of the source code
* Be efficient with respect to time and space complexity
* Report all lexical errors (with line number)

**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

* Ensure time and space efficiency and use a single pass of the token stream
* Use predictive parser using parsing table
* Produce as output the parse tree, if the source code is syntactically correct
* Produce a list of all syntax errors with appropriate messages and line numbers

**Implementation Details**

* Use C language ( Linux/ Ubuntu based GCC) to implement the modules [exact version to be announced shortly].
* 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 right immediately, spend time designing the structure of your compiler code first.

### **Stage 1: DFA construction**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 11 February 2023, 8:43 PM

The DFA for the patterns given in the language specification document should be drawn on A3-sized paper and submitted tomorrow (i.e. on February 12, 2023, Sunday) during 6:30 p.m. to 8:00 p.m. in room number 6121-Z.

1. The DFA for all patterns must have only one start state.
2. You can use pencil to draw the DFA.
3. Transition to trap states can be avoided in this submission.
4. Keywords can be implemented using a lookup table and therefore the DFA need not have separate transitions . (Mark the action at appropriate accept state)
5. Length of the string is an implementation aspect and is not represented at abstract level of DFA.
6. Ensure that there is only one transition on each input from a state (to make it deterministic)
7. Each accept state must have associated actions mentioned on paper (use pen preferably). You can write the actions as psudocode as well.
8. Remember that DFA captures the patterns (regular expressions) and does not exhibit implementation aspects (e.g. length of the lexeme, transition from accept to start state etc.)
9. You will be allowed to make changes in your design of DFA, if required while you progress with your code.
10. Write the team members' names and IDs at the upper right corner of your paper.
11. Any one member of the team can submit paper assignments, but the presence of all members of the team will be essential during code submission (stage 1 and stage 2 both).
12. You will be allotted the group number at the time of submission of your DFA.
13. You can keep a snapshot of your DFA for your future reference needed while coding your lexical analyzer

### **Stage 1: Interface details**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 16 February 2023, 9:28 PM

Complete compiler project must be developed using C programming language. Ensure that your code is compatible to **GCC version 9.4.0  on Ubuntu 20.04.4.**

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 file pointer 'fp'. This file is the source code written in the given language. The function uses efficient technique to bring 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.

**getNextToken(  ):** 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 and passes to the parser on demand. The function also displays *all* lexical errors appropriately.

**removeComments(char \*testcaseFile, char \*cleanFile):** This function is an additional plug in 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. Ensure that the line numbers of code in the cleanFile are same as original line numbers of the same code in testcaseFile. [Note: The function is invoked only once through your driver file to showcase the comment removal. However, your lexer does not really pick inputs from comment removed file. For showcasing your lexer's ability, directly take input from user source code]

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

**ComputeFirstAndFollowSets (grammar G, FirstAndFollow F):** This function takes as input the grammar G, computes FIRST and FOLLOW information and populates the appropriate data structure F. 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. The credit for the above is only 4 marks out of 45 marks reserved for stage 1 module. 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 (especially during online exam). Note: While First and Follow computation from grammar can be skipped at the cost of 4 marks, and data structure F can be populated manually, it is yet mandatory to populate the parse table automatically using the following function.

**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). Take the input file name and buffer size at command line.

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.

Please feel free to discuss with me any queries regarding implementation

### **Stage 1: Milestones**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 18 February 2023, 8:52 AM

* Understanding language features           February 6
* Constructing hand drawn DFA for lexer   February 12
* Lexical Analysis Module Development     February 14
* Testing of lexer with given test cases       February 16
* Grammar Modifications                            February 19
* Grammar Representation (for Parser)       February 21
* FIRST and FOLLOW sets implementation February 22
* Predictive Parsing Table Creation             February 23
* Testing the table creation                         February 23
* Stack ADT and Parsing                              February 24
* Tree ADT and error reporting                    February 24
* Parse tree creation while parsing              February 25
* Testing (with given test cases)                   February 26
* Testing (With self created test cases)        February 27
* Documentation of the code                      February 28
* Testing Error reporting                              March 1
* Parse tree printing etc.                              March 1
* Any other (coding details etc.)                  March 2
* Ready to Submit the code                         March 2

### **Stage 1: Some updates**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 18 February 2023, 2:44 PM

1. A switch statement must have at least one case statement.
2. Include unary plus and minus operators for arithmetic expressions. Extend grammar to support + and - with single operand of any number, real number, variable identifier and expressions. Few valid expressions using these unary operators are: +5, -value, -20, +29.5, -30.45, -(a+b), -(20+x) <78 and so on.
3. The arithmetic operators \* and / have more precedence over + and -
4. The logical operators AND and OR are of the same precedence.
5. Include boolean constants true and false (in lowercase letters). These can be the operands of any boolean expression. E.g. t:= false AND list[10]+34.5<=newarr[1]; is a valid assignment statement.
6. The input statement get\_value can read only variable identifier but cannot read any static constant (integer, real or boolean) or any array element. E.g. get\_value(A[i]) and get\_value(5) are syntactically incorrect (error).
7. The output statement print can print variable identifier, a static constant (integer, real, boolean) and an array element as well. E.g. print(a), print(true), print(false), print(5), print(56.34), print(a[4]), print(a[k]) etc. are syntactically correct. However, the expression cannot be used to access an array element while using print statement, e.g. print(A[k\*2+m]) is syntactically incorrect.
8. Array element formation cannot be recursive which means that an array element cannot be used as an index at syntax level. E.g. A[A[i]], A[B[k]], and A[A[i] + 5] etc., are invalid.
9. For loop range can have only positive integers as is mentioned in the specifications and cannot use variable identifier names to define the loop range. E.g. for (k in b..10) is a invalid syntax as b is a variable identifier.
10. The use of module keyword is essential in function call statement. E.g. [s]:= use module arraySum with parameters A,k; is a valid statement.
11. A comment is neither tokenized nor passed to the parser.
12. Input parameter list of a module cannot be empty.
13. Any module not having any statement in it is syntactically correct.
14. There is no need to create any symbol table for variable identifier during stage 1. However, you must create a hash table for keyword implementation at lexical analysis level.
15. The null and unit productions are essential part of the grammar and should not be removed. [Refer Backus Naur Form (BNF) Sebesta's book section 3.1.3 or Compiler's textbook section 2.2 for grammar representation]

### **Stage 1: Grammar modification**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 18 February 2023, 9:55 PM

* Paper work on grammar modification (with respect to completeness and LL(1) compatibility) and computation of FIRST & FOLLOW sets is due to be submitted tomorrow i.e. on **Sunday, February 19, 2023,** during **6:30 p.m. to 8:00 p.m**.
* Team members must write their assumptions, if any, on the sheets.
* The teams will be allowed to modify the grammar, if needed, while developing the stage 1 code afterwards.
* Any one member of the team can submit the handwritten sheets (properly stapled) as per the schedule above.
* Students can meet me tomorrow, during 6:00 p.m. to 6:30 p.m., to discuss any doubts.

### **Stage 1: Modified grammar**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Monday, 20 February 2023, 10:26 PM

A document with a detailed analysis of the given grammar is attached herewith for your reference. This is provided as a support only and students are advised to read the document to make necessary changes in their grammar, if required.

### **Precautions**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Monday, 20 February 2023, 10:55 PM

* Do not leave your code anywhere other than your own protected and secured system/server account.
* Do not share system passwords with your friends.
* Do not share code with your team partners over email or other unprotected communication links as the security of these might not be perfect.
* Do not leave anywhere your pen drives carrying your code.
* Make sure that the data structures and program design are your own and should not be disclosed to the other teams.
* While discussions on concepts are encouraged across the teams, the implementation details must be discussed within the team only.
* Work together with the team partners instead of distributing the modules among them.
* Observe that their contribution does not appear all of a sudden. It might be a plagiarized piece of code.
* Do not copy from anywhere ‐ seniors, batch mates, internet, book and so on.
* CREATE YOUR OWN CODE ONLY

### **Stage 1: Errata**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Tuesday, 21 February 2023, 10:33 PM

1. The underscore at the beginning of the identifier is valid and lexeme \_Abc is valid.

2. Consider rule numbers 27-c and 27-d in the given modified grammar

* **<AnyTerm> → <arithmeticExpr> <N8> | <boolConstt> <N8>**               …27 c
* **<N8> → <relationalOp> <arithmeticExpr><N8> | ε**                        …27 d

Here the nonterminal <AnyTerm> is constructing an operand for logical expression. The possible atomic operands can be a boolean expression a+b < c\*(b-d), true, false and even a simple arithmetic expression say p+q+r etc. We will modify 27-c comprising of <boolConstt> as follows by removing <N8>. The intention is to create a meaningful boolean expression only using <AnyTerm>.

* **<AnyTerm> → <arithmeticExpr> <N8> | <boolConstt>**

At the same time we need to modify rule 27-d which is at its present form creating spurious expressions of type a+b <= c- d <= f.

Therefore, we remove the recursive occurrence of <N8> from this rule. The new rule becomes

* **<N8> → <relationalOp> <arithmeticExpr> | ε**

Through the above rules, we try to create one of the three possibilities

* an atomic boolean expression comprising of arithmetic expressions combined with any relational operator
* an atomic boolean constant (true or false) or
* just an arithmetic expression (using <N8>→ ε)

The nonterminal <AnyTerm> however can combine syntactically with other terms using logical operators irrespective of whether it is a boolean expression or an arithmetic expression or a boolean constant. Any wrong type combinations will be reported as errors only at the stage 2 using type checking.

Also rule 32 needs minor correction here

* **<factor> →BO <arithmeticOrBooleanExpr> BC**

Also in rule 26, read the non terminal **<output\_plist>** as <idList>.

Students must verify the grammar and inform me immediately if any discrepancy is noticed in the modified grammar.

### **Stage 1: Errata**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 23 February 2023, 10:30 PM

**1.** Modify rule 33h such that it does not construct the RNUM for use as an index for the array element.

The rule

<arrFactor>    --> <id\_num\_rnum> | <boolConstt> | BO <arrExpr> BC

 becomes

**<arrFactor>   --> ID | NUM | <boolConstt> | BO <arrExpr> BC**

**2.** The actual parameter list must include ID, NUM, RNUM, TRUE, FALSE and array elements. The rule 24 given below is required to be modified.

 <moduleReuseStmt>   --> <optional>  USE MODULE ID WITH PARAMETERS <idList>SEMICOL

Here <idList> is changed to <actual\_para\_list>, and becomes

**<moduleReuseStmt> --><optional>  USE MODULE ID WITH PARAMETERS  <actual\_para\_list> SEMICOL**

where <actual\_para\_list> is given as follows  

**<actual\_para\_list> --> NUM| RNUM | <boolConstt> | ID <N\_11>**

**<N\_11> -->  SQBO <element\_index\_with\_expressions> SQBC | ε**

**3.**  The array access in print statement is modified to include signed values and identifier for index.

<array\_element\_for\_print> --> ID SQBO <new\_index> SQBC                                           ........16 c

changes to

**<array\_element\_for\_print> --> ID SQBO <index\_arr> SQBC**

(using rule 23 a for signed access to array elements )

**4.** The nonterminal <var\_id\_num> used in rules 27.2 and 27.3 has following rules.

**<var\_id\_num>-->  ID  | NUM  |  RNUM**

**5.** The values used for the case segments are unsigned integers and boolean constants true and false only (e.g. case 2, case true,  case false are valid, but case -2, case +3 are syntactically incorrect)

*Ensure that the grammar rules reside in* ***grammar.txt*** *file and any modifications required should be handled appropriately. Please inform me of any discrepancy in the modified grammar, if it exists.*

### **Stage 1: Test cases**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Sunday, 26 February 2023, 5:01 PM

Number of replies: 0



The test cases are uploaded for your reference (please refer the attached file).

The details are as follows:

* **t1.txt: For testing lexical analyzer output**
* **t2.txt : Syntactically Correct**
* **t3.txt : Syntactically Correct**
* **t4.txt : Syntactically Correct**
* **t5.txt : Syntactically Correct**
* **t6(with\_syntax\_errors).txt : With Errors**

Inform me of any discrepancy or syntactic error found in the test cases 2-5 immediately.

Any other error left out from mentioning in the test case 6 should also be informed to me.

Ensure that you report all errors line number wise.

### **Stage 1: Driver, Compilation and Execution details**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Tuesday, 28 February 2023, 3:24 PM

**Driver**

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 (Ensure that the line numbers of original code are preserved)

**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)

The format for printing each token is as follows.

 Line\_number          lexeme           Token\_name

Also print the lexical errors with lexemes and line numbers appropriately.

**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 stage 1 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 the driver displays necessary information regarding 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 which ever 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 earlier.

**Execution**

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

$./stage1exe testcase.txt  parsetreeOutFile.txt  size\_of\_buffer

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 size\_of\_buffer is the size of one buffer for twin buffer.

The inorder traversal for an n‐ary tree can be described as follows

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

### **Stage 1: Submission Guidelines**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Wednesday, 1 March 2023, 9:52 PM

1. Stage‐1 code submission is due tomorrow i.e. on **March 2, 2023 (Thursday)** during **6:30 p.m. to 8:00 p.m. Ensure that you are not submitting any piece of plagiarized code.**

2. Confirm your group number with my list if needed. You can meet me tomorrow during 12:00 p.m. ‐1:00 p.m. (March 2, 2023, Thursday) and verify your group number. A submission with wrong group number will not be considered for evaluation.

3. Any one member of the team will be allowed to submit the code on behalf of the team. All members of the team will be required to be present at the time of code submission.

4. Create a folder and name it as your group number (Group\_01, Group\_12, ....etc.). This folder must contain the following files

* grammar.txt
* lexerDef.h
* lexer.c
* lexer.h
* parserDef.h
* parser.h
* parser.c
* driver.c
* makefile
* testcases (should be available in your group folder directly)
* any other (token file and so on which you might have used)
* Coding details pro forma (Download the attached docx file, type in the details and submit)

5. Group folder should not have any subfolder in it.

6. Do not include \*.o and stage1exe in the above folder for uploading.

7. Files for each of the testcases generated by you should be named using the prefix *testcase* followed by the number, followed by the extension 'txt', e.g. testcase5.txt, testcase2.txt, etc

8. Submit only those testcases with which your code works well. I will be using my testcases as well for evaluation, but asking you to submit the testcases gives me a scope to verify your lexer and parser with your testcases.

9. Ensure that the makefile is named as makefile (and no other name is used for it). Check the file dependencies to ensure correct working of the makefile.

10. Ensure that the driver supports the five options for execution as were earlier mentioned.

11. Verify that each file (except any input file you use in your code) has in its first 6 lines, your group number, Ids and names of all team members in comment. Do not forget to comment these. Unnamed files will not be evaluated.

12. Ensure the compatibility of your code with the GCC specifications given earlier.

13. Refresh the course webpage (Nalanda) to verify that you have not missed out any updates. Go through all details and ensure that nothing is left out which you would have preferred to incorporate in your code.

14. Compress the group folder using zip ( of type Group\_#.zip, where # represents your group number) to upload. (Other formats such as tar, gz, and rar etc. will not be accepted)

15. Fill up the given coding details proforma with necessary details and submit along with all other files.

16. Ensure that the code you are intending to submit is not plagiarized. (Refer course handout for more details)

17. Any one of the team members' Nalanda account should be used for submission. A duplicate submission may carry a penalty in terms of deduction of marks.

18. If you choose to avail the life line, you can submit the code on March 3, 2023 (Friday) during 6:30 p.m to 8:00 p.m. without any penalty. (Refer course handout for more details)

19. Submit your file Group\_#.zip (# represents your group number) through <https://nalanda.bits-pilani.ac.in/>. The link will be made available as per the specified date and time. Avoid last minute submission. If you find any problem in uploading the files through the given link, please do not panic and do not send any code through email. Code sent through email will not be accepted . In case of any problem in uploading your file, you can write to me over an email.

20. Teams not following the submission guidelines, will be penalized appropriately.

21. Submission will be complete after successfully uploading the zip file as per the above details.

In case of doubts, please contact me through email.

### **Stage 1: Submitted code**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Wednesday, 8 March 2023, 6:44 PM

If any minor issues are observed by the teams with their stage 1 code, after its submission, the same can be rectified by allowing **at most 4 lines**of changes at a minor penalty of **0.5**marks each. The issues could have existed due to incompatibilities of versions of GCC or UBUNTU, used by the teams, or use of other operating system, or makefile not working, or any other.

You are advised to find the best solution to resolve your problem in minimum lines of changes and fill in the form available at the course page. The portal for receiving the input will be available from **today 7:00 p.m. to Friday, March 10, 2023, 7:00 p.m.**Teams willing to submit the changes in this duration must submit the details in the given link.

If the solution is not readily available with some teams seeking this minor correction in their code, they can do so after the mid semester tests, but will require to fill in their willingness to do so by March 10, 2023 in the same link as above.

### **Stage 2: Modules**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Wednesday, 8 March 2023, 8:49 PM

Develop the following modules of the compiler for implementing the ERPLAG language.

**Abstract Syntax Tree (AST):** This module takes as input the parse tree generated in stage 1. The abstract syntax tree is generated by eliminating unnecessary details such as semicolon, colon, comma, parenthesis, square brackets, range operator, assignment operator etc. Any node in the abstract Syntax Tree retains the information about the non terminal symbol that would have derived the corresponding subtree (which is essential to keep the syntactic structure intact with you throughout, during front end ). The AST retains only those children that are essential later for semantic analysis, while those appearing as a linear chain, are collapsed. Any child node corresponding to the operator (+, ‐, <= , AND etc) can be collapsed and the information regarding the terminal token such as PLUS, MINUS, LE, AND etc. is fetched up to the parent node. Implement an AST with the following general rule

 (parent)(children list)

where parent is the name of the construct representing the sub-tree and can be either any new name or the same non‐terminal symbol that exists in the parse tree. The children list contains the nodes which are meaningful. The leaf nodes of the AST still continue to contain the tokens and other relevant information extracted during stage 1. The AST later helps in traversing the tree faster than the parse tree, traversing the meaningful nodes only. You must

* Prepare the semantic rules to derive the abstract syntax tree structure
* Modify the structure of the parse tree node (at least a link to the ST can be added) to use for AST.
* Ensure a single pass of the parse tree.
* Produce as output the Abstract Syntax Tree, if the source code is syntactically correct.

**Symbol Table (ST):**This module takes as input the AST generated as above. Symbol table is a special data structure that maintains the information about the identifiers (variables that participate in computation as the source code is executed). The information gathered during semantic analysis phase is extremely valuable for generating the assembly language code for the input source code. Variables declared in different static scopes can be maintained in separate symbol tables. Since the scope of the variables is known only when the syntactic structure is established, i.e. after parsing, the symbol table links can be established to the function definitions after syntax analysis.

Implement Symbol Table to incorporate following information for all identifiers.

* Type
* Scope
* Offset etc.

Identifiers corresponding to the function names must be maintained separately. The information such as whether the token ID corresponds to a variable or it corresponds to the function name is obtained by the syntactic structure of the sub-tree that contains this identifier.

**Type Extractor and Checker:** Type of an identifier is extracted from the declaration statement that declares the identifier. The data types supported in the language you are implementing are: integer, real, boolean and array. The type of an element of an array is the type of the data the array refers to. For instance, consider the statement declare a: array[1..15] of integer;, then the type of an element a[5] is integer. The type checker verifies the type of an expression appearing at the right hand side of the assignment statement such as value := (a+b‐c );  and checks if it matches with that of the identifier on the left hand side. An arithmetic operator can have two operands of the similar type, where types can be integer and real data types. Example, if a, b, c and d are declared as integer then,

An expression in a statement a := (b\*2‐4\*c)+5\*d; its RHS has a type integer and since the identifier a is also of the same type, the type checker approves the expression assignment

An expression in a statement a:= (b\*2 +c)/d <= c\*10; has a boolean valued expression which is assigned to the integer valued identifier, hence the expression value assignment is wrong.

***Static type checking rules****:*

* The type of an identifier is the type appearing while declaring the variable.
* The type of NUM is integer.
* The type of RNUM is real.
* The type of true or false value is boolean.
* The type of an array variable A of type array [12..20] of real (say) is defined as an expression <real, 12, 20>. The type of an array element A[13] (say) is real if 13 is within the bounds [12, 20].
* The type of a simple expression (say E) of the form expression(say E1) <operator> Expression(say E2)
* is integer, if both expressions are of type integer and the operator is an arithmetic operator PLUS, MINUS or MUL.
* is real, if both expressions are of type integer, or one is integer and the other is real, or both are real, and the operator is an arithmetic operator DIV.
* is real, if both the expressions are of type real and the operator is arithmetic operator PLUS, MINUS or MUL.
* is boolean, if both expressions are of type integer and the operator is a relational operator
* is boolean, if both expressions are of type real and the operator is relational.
* is boolean, if both expressions are of type boolean and the operator is logical.
* The type of the expression is ERROR, if the above rules do not derive the type of E appropriately.
* Types of expressions using unary - or + operators are same as those of their operands.
* Type checking rules for array construct are as follows:

The operations +, -, \*, / and all relational and logic operators, cannot be applied on array variables of array type. For example, consider the declaration statement, declare A, B: array [12..20] of real; then the type of A and B are both <type, 12, 20>. The expression A+B, A-B, A\*B and A/B are invalid and the type of these is assigned as ERROR.

The assignment operator applied to two array variables of the same type is valid. For example, if A and B are the array variables of type array[12..20] of real, then A:= B; is a valid statement. This applies to dynamic arrays of type array[a..b] of real as well.

Consider array elements with index represented by integer identifier say A[k]. Here type checking of variable k and A[k] are done at compile time, but the bound checking of A[k] is done at run time. If the type of k is integer, then it is valid, else it is reported as an error. Also, the type checking of A[13], for type of index (NUM), is done at compile time. The bound checking of A[13] where A is a static array is done at compile time. If A is a dynamic array, then bound checking of A[13] is done at run time [see below]

The type of an identifier or an expression is computed by traversing the abstract syntax tree. for declaration statement construct.

***Dynamic type checking rules:***

Following type checks are dynamic and your code generator takes care of dynamic type checking module.

* Address computation and type checking of variables of dynamic arrays such as in declare A: array[a..b] of integer; The offset computation is dependent on values of a and b and is done at run time.
* Bound checking of elements A[10] and A[n] for dynamic arrays, and of A[n] of static arrays is done at run time.
* Bound checking of array elements using arithmetic expression as index is done at run time. For example, an element A[p+q\*r] is an array element whose index type is checked at compile time, but the bound checking is done at run time.
* The type checking of the arithmetic expression on the right hand side of the assignment operator in the statement m := n+A[p+q\*r]; is done at compile time. Similar is done for the type checking of the operands of the assignment operator, which are m and n+A[p+q\*r].

**Semantic Analyzer:** This module verifies the semantics of the code. Following are the rules that ERPLAG supports.

* An identifier cannot be declared multiple times in the same scope.
* An identifier must be declared before its use.
* The types and the number of parameters returned by a function must be the same as that of the parameters used in invoking the function.
* The parameters being returned by a function must be assigned a value. If a parameter does not get a value assigned within the function definition, it should be reported as an error.
* The function that does not return any value, must be invoked appropriately.
* Function input parameters passed while invoking it should be of the same type as those used in the function definition.
* A switch statement with an integer typed identifier associated with it, can have case statement with case keyword followed by an integer only and the case statements must be followed by a default statement.
* A switch statement with an identifier of type real is not valid and an error should be reported.
* A switch statement with a boolean type identifier can have the case statements with labels true and false only. The switch statement then should not have a default statement.
* Function overloading is not allowed.
* A function declaration for a function being used (say F1) by another (say F2) must precede the definition of the function using it(i.e. F2), only if the function definition of F1 does not precede the definition of F2.
* If the function definition of F1 precedes function definition of F2(the one which uses F1), then the function declaration of F1 is redundant and is not valid.
* A for statement must not redefine the variable that participates in the iterating over the range.
* The function cannot be invoked recursively.
* An identifier used beyond its scope must be viewed as undefined etc. (More semantics will be made available in the test cases)

**Code Generator :** This module takes as input the abstract syntax tree (AST) as intermediate representation. The function generates 8086 assembly code. Only trivial optimization such as avoiding redundant code, appropriate register usage etc. is needed while the detailed code optimization techniques are not expected to be implemented. The code generator generates the code for dynamic type checking as well. Your compiler must generate equivalent assembly code (in file code.asm) with instructions taken from instruction set of the NASM simulator (linux based, exact version will be provided later).

### **Stage 2: Important dates**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 9 March 2023, 4:27 PM

Stage 2 of the project will have 28 days from March 8, 2023, to April 12, 2023, excluding the mid-semester test week of March 12-18, 2023.

**1.** Date of posting stage 2 module description: **March 8, 2023**

***2. Paper submission***Semantics rules for AST creation: **March 26, 2023**  [During 6:30 p.m. to 7:30 p.m., Venue 6121-Z]

***3. Submissions through Nalanda***

Coding Details in given proforma:**April 12, 2023 [During 6:30 p.m. to 8:00 p.m.]**

Integrated compiler code submission due on :**April 12, 2023 [During 6:30 p.m. to 8:00 p.m.]**

**4. Online exam: April 23, 2023** (Sunday)

### **Stage 2: Ground work**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 9 March 2023, 4:48 PM

1. Read the language specifications document carefully and understand semantic specifications.

2. Formulate rules for creation of abstract syntax tree. Work out the AST creation for expression grammar as discussed in the class.

3. Focus on individual constructs and understand the need for meaningful information for code generation.

4. Many issues needed for your stage 2 implementation will be discussed during regular lectures.

5. Students are advised to attend classes regularly. As we progress with semantic analysis and symbol table creation concepts in the class, students must keep on preparing themselves to do the following groundwork.

6. Read more to understand semantics of individual constructs such as expression, boolean expression, assignment statements, control statements, return statement, parameter list, activation record structure, name binding and scopes, type checking etc. [You can refer book on Concepts of Programming Languages - by Robert W. Sebesta, Tenth edition, Pearson Publication, to refresh concepts of Principles of Programming Languages]

7. Students are advised to read the text book and solve problems regularly.

8. Work out rules for abstract syntax tree generation and prepare the document.

9. If there were issues with your lexer and parser, keep modifying the code. Discuss with me if you need help in identifying the flaws in your code.

10. Regularly discuss your understanding with me and clarify your doubts. [after the class or by fixing over email a mutually convenient time for discussion]

### **Stage 2: AST rules format**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Wednesday, 22 March 2023, 9:16 PM

The semantic rules for AST creation, due for submission this Sunday, are required in handwritten format only. However, the team's original grammar rules can be submitted in printed form for reference so as to avoid rewriting the grammar rules. This will need only the number of the grammar rule to be mentioned before the corresponding AST rules.

The format for AST rules can be as follows 

4. { <write here traversal order>: < write all rules here>.......................

                        ...........................................

   <write here traversal order>: < write all rules here>.......................

                        ...........................................

    }

5. ......

Make sure the printed grammar rule numbers match with those of your handwritten semantic rules for AST creation. Use only the address attributes as discussed in the class. These could be inherited or synthesized. Use suffix \_inh and \_syn respectively for the attributes. An address could be in terms of the linked list, a node address or any other form of address.

The AST once coded and output by your semantic analyzer will be used as input for remaining tasks in the second stage later. Minor variations in the format of AST rules are acceptable provided the grammar rule number and evaluation orders are also mentioned correctly.

Please feel free to write to me if you have any questions.

### **Stage 2: Milestones**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Thursday, 23 March 2023, 7:02 PM

* Understanding the problem and modules:  March 11
* Design of Semantic rules for AST creation:  March 26
* Abstract syntax tree construction: March 28
* Design of suitable data structure for Symbol table: March 29
* Symbol table implementation: March 30
* Type checker implementation: March 31
* Learning of ISA of NASM simulator (<http://www.nasm.us/):> April 1
* Semantic rules implementation:  April 3
* Design of rules for code generation: April 5
* Code generation implementation: April 7
* Code generation for dynamic type checking: April 8
* Testing with your own test cases: April 9
* Testing with the test cases uploaded later:  April 10
* Integration of stage1 and stage 2: April 11
* Compatibility checks and any pending issue: April 12
* Ready to submit code: April 12

### **Stage 2: Submission of semantic rules for AST creation**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Sunday, 26 March 2023, 12:21 PM

Write your Group number, IDs, and the names of all team members at the top of your sheets. Any one member can submit on behalf of the team the handwritten semantic rules for AST creation along with the grammar rules today as per the following schedule.

* Date: March 26, 2023
* Day: Sunday
* Time: 6:30 p.m. to 7:30 p.m.
* Venue: 6121-Z

Students are advised to retain a copy of the rules with them for future reference while coding.

If any traversal order is difficult to be specified, create a dependency graph for the construct in a separate sheet and mention the figure number in place of the traversal order prior to the corresponding semantic rule.

In case of any query, please write to me.

### **Stage 2: Implementation updates**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Tuesday, 28 March 2023, 4:04 PM

**Function Prototypes and File Names**

Function prototypes are flexible. Students are advised to use names of data structures such as ast, parseTree, symbolTable etc. appropriately. You can select names of implementation files appropriately from file names ast.c, symbolTable.c, typeExtractor.c, semantics.c, codegen.c etc. You can have additional files, if you need as support, but the name of the file must be indicative of the contents within it.

The function prototype declarations should be in file \*.h corresponding to the implementation file name [ For example , if you are naming the interface file for the functions in symbolTable.c, name that interface file as symbolTable.h]. The data definitions should also be split in the files appropriately as symbolTableDef.h. etc.

*Intermediate Code Generation*

Teams can generate assembly language code by first constructing the intermediate code using instructions given in text book or can generate the code directly. [The process of code generation through intermediate code generation is more systematic and produces correct code while direct code generation may be erroneous.] However, there is no extra credit for IR (Intermediate Representation) creation as it is left to the decision of teams as to whether to generate code through IR or by skipping IR . The correctness of the generated code will be of significance.

**Instruction Set**

Your compiler must generate equivalent assembly code (in file code.asm) with instructions taken from instruction set of the NASM simulator (linux based). Download NASM (Version 2.16.01)/ from  <https://www.nasm.us/> to verify correctness of the output obtained by executing assembly code generated by your compiler for the user source code in given toy language. Make your code generator module to emit 64-bit code in assembly language for the given test case. As has been mentioned earlier that code generator is not expected to implement any optimizations except for any trivial ones in regard to register assignment or instruction selection etc. Use simple templates for mapping the IR code to equivalent assembly code as was discussed today in the class.

**Efficiency**

Efficiency (Time and Space ) is an expected feature of your compiler code. Design efficient data structure for symbol table etc. Abstract Syntax Tree (AST) is a copy of the user source code in concrete form and all other work including type checking, symbol table, semantic analysis and code generation etc. are expected to be done by traversing the AST only instead of traversing the parse tree. While constructing AST from parse tree, the unused nodes of the parse tree should be freed. The semantic analysis, type checking, symbol table construction and code generation rules should be based on the constructs (sub trees) of the AST.

Compatibility with the Ubuntu and GCC versions specified during stage 1 must be ensured.

### **Stage 1: Marks**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Tuesday, 4 April 2023, 6:20 PM

**Marks Distribution**

**Lexer‐ 20 Marks**

* *Code logic - 5 Marks*
* Paper work‐1 Mark
* DFA - 4 Marks
* *Lexer output - 15 Marks*
* Comment removal output - 2 Marks
* Tokens and lexemes output - 8 Marks
* Lexical errors detection– 5 Marks

**Parser‐25 Marks**

* *Code logic ‐ 12 Marks*
* Paper work‐1 Mark
* First and follow automation‐5 Marks
* Parse table creation - 2 Marks
* parsing‐ 2 Marks
* Parse tree creation‐2 Marks
* *Parser output ‐ 13 Marks*
* Syntactically correct test case - 2 Marks
* Parse tree printing - 2 Marks
* Syntactically Incorrect test cases (with errors) ‐ 6 Marks
* error recovery - 3 Marks

Some of the student codes did not compile and a few gave segmentation fault, which are awarded 0.

A penalty of 0.5 mark per suggested line change was imposed on 4 changes while counting two lines comment/uncomment change as one. More than two lines (in continuation), required to comment in the team's code was penalized with 1 mark per set. Any changes more than 4 in total were accommodated with 1 mark penalty each additional change.

**Rechecks:** Genuine recheck requests will be accepted during April 5, 2023 (10:00 AM) to April 6, 2023 (7:30 PM). A separate link will be opened on Nalanda tomorrow at 10:00 AM. 

**Marks list**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Group No. | Lexer(20 M) | | | | | Parser(25 M) | | | | | | | | | total (45 M) [A] | Correction penalty [B] | Final Marks              (45M)                  [A-B] |
| Code logic (5) | | Lexer output (15) | | | Code logic (12) | | | | | Output (13) | | | |
| paper (1) | DFA code logic (4) | comment removal (2) | Correct tokens (8) | Error reporting  (5) | paper (1) | First and Follow (5) | Parse table creation (2) | parsing (2) | parse tree (2) | syntactically correct  tetstcase (2) | parse tree printing (2) | syntactically incorrect  tetstcase (6) | error recovery (3) |
| 1 | 1 | 4 | 2 | 8 | 2 | 1 | 5 | 2 | 2 | 2 | 2 | 0.5 | 6 | 3 | 40.5 | 0 | 40.5 |
| 2 | 1 | 4 | 2 | 8 | 4 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 38 | 0 | 38 |
| 3 | 1 | 4 | 2 | 8 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 17 | 0 | 17 |
| 4 | 1 | 4 | 2 | 8 | 3 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 43 | 0 | 43 |
| 5 | 1 | 2 | 2 | 0 | 0 | 1 | 3 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 12 | 0 | 12 |
| 6 | 1 | 2 | 2 | 0.5 | 0 | 1 | 3 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 12.5 | 2 | 10.5 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 1 | 1.5 | 0 | 0 | 0 | 1 | 0 | 0 | 0.5 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 4 |
| 9 | 1 | 4 | 2 | 8 | 5 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 42 | 0 | 42 |
| 10 | 1 | 2 | 2 | 0 | 0 | 1 | 0 | 1 | 0.5 | 1 | 0 | 0 | 0 | 0 | 8.5 | 0 | 8.5 |
| 11 | 1 | 2 | 2 | 0 | 0 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 30 | 0 | 30 |
| 12 | 1 | 4 | 2 | 8 | 5 | 1 | 0 | 0.5 | 0 | 0 | 0 | 0 | 0 | 0 | 21.5 | 1 | 20.5 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 1 | 1.5 | 2 | 0 | 0 | 1 | 3.5 | 1.5 | 1.5 | 2 | 2 | 2 | 2.5 | 1 | 21.5 | 0 | 21.5 |
| 15 | 1 | 4 | 2 | 8 | 5 | 1 | 4 | 1.5 | 1.5 | 2 | 2 | 2 | 2 | 0.5 | 36.5 | 0 | 36.5 |
| 16 | 1 | 4 | 2 | 8 | 5 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 45 | 0 | 45 |
| 17 | 1 | 4 | 2 | 8 | 5 | 1 | 4 | 1.5 | 1 | 2 | 2 | 2 | 1.5 | 1 | 36 | 0 | 36 |
| 18 | 1 | 1.5 | 0 | 0 | 0 | 1 | 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 8.5 | 0 | 8.5 |
| 19 | 1 | 1.5 | 0 | 0 | 0 | 1 | 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 8.5 | 1.5 | 7 |
| 20 | 1 | 4 | 2 | 8 | 5 | 1 | 4 | 1.5 | 1 | 1.5 | 1 | 2 | 6 | 3 | 41 | 6 | 35 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22 | 1 | 1.5 | 2 | 0 | 0 | 1 | 0 | 1.5 | 1 | 1.5 | 2 | 2 | 0.5 | 0.5 | 14.5 | 0 | 14.5 |
| 23 | 1 | 4 | 2 | 8 | 5 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 45 | 0 | 45 |
| 24 | 1 | 4 | 2 | 8 | 5 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 45 | 2 | 43 |
| 25 | 1 | 3 | 1 | 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 10 |
| 26 | 1 | 3 | 0 | 8 | 3 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 19 | 3 | 16 |
| 27 | 1 | 3 | 2 | 6 | 0 | 1 | 3 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 19 | 3.5 | 15.5 |
| 28 | 1 | 4 | 2 | 6.5 | 4.5 | 1 | 0 | 2 | 2 | 1 | 2 | 0 | 6 | 0 | 32 | 2 | 30 |
| 29 | 1 | 4 | 2 | 8 | 5 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 45 | 0 | 45 |
| 30 | 1 | 4 | 2 | 8 | 5 | 1 | 2.5 | 1 | 1 | 1 | 0 | 0.5 | 0 | 0 | 27 | 0 | 27 |
| 31 | 1 | 1 | 0 | 0 | 0 | 1 | 2 | 0.5 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 6.5 | 2 | 4.5 |
| 32 | 1 | 4 | 2 | 8 | 5 | 1 | 0 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 22 |
| 33 | 1 | 2.5 | 2 | 8 | 0 | 1 | 3.5 | 1 | 1 | 1 | 0 | 0.5 | 0 | 0 | 21.5 | 0 | 21.5 |
| 34 | 1 | 3 | 2 | 6 | 2.5 | 1 | 4 | 1 | 1 | 1 | 2 | 2 | 0 | 0 | 26.5 | 0 | 26.5 |
| 35 | 1 | 2 | 0 | 0 | 0 | 1 | 2 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 7 |
| 36 | 1 | 4 | 2 | 8 | 4 | 1 | 2 | 0.5 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 23.5 | 0 | 23.5 |
| 37 | 1 | 4 | 2 | 8 | 5 | 1 | 4 | 2 | 2 | 2 | 2 | 2 | 4 | 2 | 41 | 4 | 37 |
| 38 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 3 |
| 39 | 1 | 4 | 2 | 8 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 25 | 3 | 22 |
| 40 | 1 | 4 | 2 | 8 | 2 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 4.5 | 2.5 | 40 | 0 | 40 |
| 41 | 1 | 4 | 2 | 8 | 3.5 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 43.5 | 0 | 43.5 |
| 42 | 1 | 4 | 2 | 8 | 3.5 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 21.5 | 0 | 21.5 |
| 43 | 1 | 3 | 2 | 2 | 0 | 1 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 15 |
| 44 | 1 | 4 | 2 | 7 | 2 | 1 | 3 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 23 | 0 | 23 |
| 45 | 1 | 4 | 2 | 7 | 0.5 | 1 | 2.5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 37 | 0 | 37 |
| 46 | 1 | 4 | 2 | 8 | 4 | 1 | 5 | 2 | 2 | 2 | 1 | 2 | 6 | 3 | 43 | 0 | 43 |
| 47 | 1 | 4 | 2 | 8 | 5 | 1 | 5 | 1.5 | 1.5 | 0 | 0 | 0 | 4 | 2 | 35 | 0 | 35 |
| 48 | 1 | 2 | 2 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 9 | 0 | 9 |
| 49 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 50 | 1 | 3 | 2 | 4 | 2 | 1 | 0 | 0.5 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 14.5 | 0 | 14.5 |
| 51 | 1 | 4 | 2 | 8 | 3 | 1 | 5 | 2 | 2 | 2 | 2 | 2 | 6 | 3 | 43 | 0 | 43 |
| 52 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 0.5 | 0.5 | 0.5 | 0 | 0 | 0 | 0 | 5.5 | 0 | 5.5 |
| 53 | 1 | 4 | 2 | 8 | 0 | 1 | 4 | 2 | 2 | 1 | 2 | 0 | 4 | 2 | 33 | 0 | 33 |

### **Stage 2: Semantic analysis test cases**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Saturday, 8 April 2023, 9:04 AM

The test cases are uploaded for your reference. There are ten different test cases with increasing complexity handling variety of semantic checks.

Please inform me if

* there is any lexical or syntactic error in any of these test cases.
* any semantic or type error is not reported by me.
* any error is reported incorrectly.
* or you observe any discrepancy in the test case.

If a variable is used but not initialized need not be reported as an error in these test cases. This will be taken care of in the code generation test cases. You are expected to report all errors with line numbers in the following format.

Line #: Error message

where # is the line number.

The code generation test cases will be uploaded soon.

**Symbol Table format**

**Symbol Table:** For printing the Symbol Table giving following information (ten in number) for each variable identifier at each line using formatted output. **[ Use width of variables of type integer as 2, of real as 4 and of boolean as 1 for printing the symbol table]**

1. Variable name
2. Scope - module name
3. scope - line number pairs of start and end of the scope
4. type of element
5. is array
6. if array, whether static or dynamic
7. if array, range variables or number lexemes (e.g. [m, n], [p,q], [10, 20] etc.)
8. width (if a variable is of array type, then add 1 to total requirement for all elements of an array for holding address of the first element)
9. offset
10. nesting level (for an input or output parameter, level= 0, local variable in function definition, level = 1, any variable inside a nested scope should get its level incremented appropriately)

For example, consider module var\_demo\_array in the test case t4.txt whose symbol table is displayed below. Similarly other modules also become part of this.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| variable name | scope (module name) | scope (line numbers) | type of element | is\_array | Static/dynamic | array range | width | offset | nesting level |
| x | var\_demo\_array | [5-29] | integer | no | \*\* | \*\* | 2 | 0 | 0 |
| m | var\_demo\_array | [5-29] | integer | yes | static | [2,20] | 39 | 2 | 0 |
| p | var\_demo\_array | [5-29] | real | no | \*\* | \*\* | 4 | 41 | 0 |
| n | var\_demo\_array | [5-29] | integer | no | \*\* | \*\* | 2 | 45 | 0 |
| k | var\_demo\_array | [5-29] | boolean | no | \*\* | \*\* | 1 | 47 | 0 |
| a | var\_demo\_array | [5-29] | integer | no | \*\* | \*\* | 2 | 48 | 1 |
| b | var\_demo\_array | [5-29] | integer | no | \*\* | \*\* | 2 | 50 | 1 |
| p | var\_demo\_array | [5-29] | integer | yes | static | [2,20] | 39 | 52 | 1 |
| p | var\_demo\_array | [16-23] | integer | no | \*\* | \*\* | 2 | 91 | 2 |
| b | var\_demo\_array | [16-23] | integer | yes | static | [2..20] | 39 | 93 | 2 |

and so on. If an entry is not applicable, then mark as \*\*. The activation record size of this module should also be calculated based on the use of space for parameters, local variables, return addresses etc.

Once the symbol table is completely populated, use **printSymbolTable()** to print the contents of that by traversing the hierarchical hash tables and print appropriately. It is absolutely acceptable that the variables may not appear in the same order as they appeared in the test case code, provided their computed offsets genuinely verify the ordering of variables in the test case. However, variable sequence from one scope should not be overlapped with that of another scope of the same nesting level (say sibling). One scope's (say S) variables along with all its children nested scopes variables should be printed before you print the variables from a sibling scope of S.

### **Stage 2: Code generation test cases**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Sunday, 9 April 2023, 12:09 PM

Please find attached herewith eleven test cases for your reference. These test cases handle different constructs with increasing complexity.

The formats for reading values from the console are as follows (template for get\_value(a) should be created according to type of variable a).

For primitive data types - integer, real and boolean

$> Input: Enter an integer value

$> 5

For array type variable (for example for declare A: array[6..10] of integer; get\_value(A);, see the format for reading array element values - read element values from different lines)

$> Input: Enter 5 array elements of integer type for range 6 to 10

$>2

$>-6

$>4

$>10

$>-8

**Printing on the console:** The formats for printing values on the console are as follows (template for print(a) should be created according to type of variable a).

For primitive data types - integer, real and boolean

$> Output: 10

$> Output: true

For array type variable - print the array elements all in one line separated by a blank.

$> Output: 2 -6 4 10 -8

Please inform me if you notice any discrepancy in the test cases c1-c11.

Please inform me, if any test case has

* any lexical or syntax errors.
* any type errors except the ones of dynamic types like dynamic arrays or array element access using id or expressions
* any output wrongly reported by me in each test case
* not reported any valid output or the reported output is wrong
* or any other discrepancy observed by you

Driver, compilation, execution details will be uploaded soon.

### **Stage 2: Errata (Semantic analysis test cases)**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Monday, 10 April 2023, 9:51 AM

* **t1.txt**
* Line 36: Type mismatch error
* Line 38: variable res\_2\_b needs to be corrected to res\_2\_\_b
* Line 56: Input parameter type mismatch
* **t3.txt**
* Line 24: w, m and x are found in the static parent and r and a are local.
* Line 51: type mismatch error
* Line 72: Comment closes with \*\*
* Line 81: undeclared variable res\_2
* Line 82: Variables d and p are not declared
* Line 95: variable k is not declared and number of input parameters mismatch
* **t4.txt**
* Lines 16-23: none of the while loop condition variables is assigned a value in the loop body
* Line 66: Type mismatch
* Line 68: C[9] is out of bound
* **t5.txt**
* Line 21: Type mismatch error as the arrays b and m are not structurally equivalent
* Line 66: Variable E is not declared
* **t6.txt**
* Line 44: correct the data type beeolean to boolean
* Line 52: the type mismatch for output parameter v3 and for input parameter b7
* **t10.txt**
* Line 55: Variable Q is not declared
* Line 60: Variable E is not declared
* Line 65: Variable E is not declared

Please check all the constructs for

* At least one of the conditional variables in while loop gets assigned a value
* Each for loop, while loop or case statement has its own scope.
* The individual case statement has its own scope, therefore verify the variables not declared errors in the case scopes, if not reported by me.
* For loop variable is not assigned any value in its scope.
* structurally equivalent arrays can be assigned to each other. For example declare A: array[-3..12] of real; B: array[-20..-5] of real; then A and B are structurally equivalent and A:=B is not an error. If C: array[-20..-4] of real; then A:=C; is a type mismatch.

Inform me if any other discrepancies are observed by you in these testcases or in code generation testcases.

### **Stage 2: Some updates and explanations**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Monday, 10 April 2023, 10:46 AM

1. Please use the 64-bit representation in NASM. The latest version of NASM is not stable as reported by the teams. Therefore, you are advised to use 2.14.02 which works well for the UBUNTU version used by us.

2. You can use standard printf and scanf functions in your generated code. Make sure that complete code as generated by your compiler should be in code.asm file. It is expected that the templates for printing of data of variables or reading data for variables of different data types, such as integer, real, boolean or array type, should be part of the assembly code of the function. Separate C files for such implementations will not be accepted.

3. Include in your test cases the semantic of verifying that at least one of the conditional variables used in while loop construct is assigned a value. If none of the conditional variables is assigned any value within the nested scope of while should be treated as error.

4. You can use one complete word for representing a boolean value. Use 1 for reading true value and 0 for reading false value. However, printing of the value of a boolean variable should be strings true and false, and not 1 and 0. Use messages comprising of strings appropriately.

5. All compile time errors should be reported till the end of the user source code. You should not exit the code after reporting few errors. Rather, keep reporting the errors till the end of user source code.

6. Scope information must be collected in three different forms - name of the function, pairs of line numbers of start and end for a scope, and nesting level. These should be reflected in the symbol table output as per the format given earlier.

7. A switch statement construct should be handled with extra care. If the type of the switch variable is an array or a real number, you should not visit its corresponding sub-tree for reporting of additional errors such as case value and the presence or absence of the default statement.

8. Implementing code generation for function calls using static or dynamic array variables is complex and students are advised to first complete other requirements and take up this only if they have sufficient code confidence and available time.

9. Error reporting should be line-wise if the error is present at that line. Any error due to the absence of the construct is needed to be reported using the scope lines of that block e.g. lines 22-45, none of the variables of while loop is assigned a value, or a default is missing in a switch case statement, and so on.

10. Placing of the input parameters of array type by the caller: The caller places ***three*** values implicitly for the array parameter (say A)

* the address of the first element of A (say B), and the
* values of lower and higher subranges (say m,n)

**Static array:** An input parameter of array type with integer range values (say 10 and 20, in an array variable declared as declare A: array[10..20] of integer) is required to pass information (by the caller to the callee) about the base address of the first element of the array along with the range values (low and high ) for future bound checking by the callee. In total the information passed comprises of one address (base of A = base address of caller + C + offset of A) and two integer constants (low = 10 and high = 20 - available at compile time in the symbol table). Remember that the array elements data remains in the caller's memory and only its base reference is passed to the callee. This imitates pass by reference parameter passing method.

**Dynamic array:** Consider a dynamic array - declare A: array [low..high] of integer. Procedure almost remains same except that the range values (low and high) need to be copied by the caller from their associated memory locations (low and high values - accessed from locations of low and high, populated at run time) to the memory designated for the input parameters.

11. Calculation of width of array input parameters and local variables of array type (for symbol table as well as for code generation): This is calculated different from width of local variable of array type.

* Width of **input parameter** of array type - static or dynamic: 1(base address )+ 2\*sizeof(int)  [refer B, m and n above]
* Width of **local variable** of array type - **static**: 1(base address) + (high-low +1) \* sizeof(array element)
* Width of **local variable** of array type- **dynamic:** 1(base address)  [Note: we allow the elements to be populated after all fixed sized variables. Assumption: any array to be passed as a parameter to the callee should have been populated before the call].

Remember that the widths of local variables of array type are different for static and dynamic arrays.

### **Stage 2: Driver, makefile and execution details**

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) - Monday, 10 April 2023, 11:49 AM

**[A]. Driver**

Your driver must have the following TEN choices

Press option for the defined task

**0.** To exit from the loop (Ask for the choices in a loop)

**1. Lexer:** For printing the token list generated by the lexer (on the console)

**2. Parser:** For parsing to verify the syntactic correctness of the input source code and to produce parse tree (On Console)

**3. AST:** For printing the Abstract Syntax Tree in appropriate format. Also specify the traversal order at the beginning. (On Console)

**4. Memory:** For displaying the amount of allocated memory and number of nodes to each of parse tree and abstract syntax tree for the test case used. The format should be as per the example given below

Parse tree Number of nodes = 150, Allocated Memory = 1024 Bytes

AST Number of nodes = 30, Allocated Memory = 200 Bytes

Compression percentage = ((1024‐200)/1024)\*100

(use sizeof() to compute size of allocated memory while allocate memory during construction of these trees)

**5. Symbol Table:**For printing the Symbol Table giving following information (ten in number) for each variable identifier at each line using formatted output as was specified earlier. **[ Use width of variables of type integer as 2, of real as 4 and of boolean as 1 for printing the symbol table]**

1. Variable name
2. Scope - module name
3. scope - line number pairs of start and end of the scope
4. type of element
5. is array
6. if array, whether static or dynamic
7. if array, range variables or number lexemes (e.g. [m, n], [p,q], [10, 20] etc.)
8. width (if a variable is of array type, then add 1 to total requirement for all elements of an array for holding address of the first element)
9. offset
10. nesting level (for an input or output parameter, level= 0, local variable in function definition, level = 1, any variable inside a nested scope should get its level incremented appropriately)

**6. Activation record size (fixed, excluding system related):** For printing the total memory requirement (sum total of widths of all variables in the function scope) for each function. The format is as follows

* function1 18
* compute\_sum 34

 .... and so on

**7. Static and dynamic arrays:** For printing the type expressions and width of array variables in a line for a test case for the following information. Separate entries using formatted output (e.g.  %10d, %15.4f and so on )

* Scope - module name
* scope - line number pairs of start and end of the scope
* Name of array variable
* whether static or dynamic
* range variables or number lexemes
* type of element

 Example format

 switch\_var\_demo1           36- 56          B         static array      [10, 30]          integer

 switch\_var\_demo1            36- 56         E         static array       [4, 10]             integer

 var\_demo\_array               178-200        b4      static array       [100, 150]        boolean

...........................

..................

 module name                  67- 119            A        dynamic array    [m, n]           real

  and so on......

**8. Errors reporting and total compiling time:** For compiling to verify the syntactic and semantic correctness of the input source code If the code is syntactically incorrect, report all syntax errors only. If the code is syntactically correct, then report all type checking and semantic errors. Also print (on the console) the total time taken by your integrated compiler. Print both total\_CPU\_time and total\_CPU\_time\_in\_seconds (as mentioned earlier)

**9. Code generation:** For producing assembly code (Linux based NASM will be used for execution) (assuming that there is no lexical, syntactic, semantic or type mismatch error in the test cases).

Perform actions appropriately by invoking appropriate functions as above All lexical, syntax and semantic errors including type mismatch errors must be reported appropriately on the console (Standard output) ONLY and not in any file unless otherwise specified.

The very first line on the console, as your compiler code executes, should contain a message regarding the status of your work such as

**LEVEL #: Message**

where # is any one index in {1,2,3,4} and the message is Symbol table/type Checking/ Semantic rules module(s) work(s)/ handled static and dynamic arrays in type checking and code generation (specify one or many as applicable). The level specification is according to the total number of semantic rules you could successfully implement. The measure for the total number of semantic rules (type checking and semantic ) is as per the number of errors you could target. However, This level of message does not count on the syntax errors (which however are to be reported if exist).

Specify LEVEL as

1 , if less than or equal to 5 ERRORs out of all errors could be successfully handled

2 , if 6‐10 ERRORs could be implemented

3 , if 11‐15 ERRORs could be implemented

4 , if 16 or more ERRORs could be implemented

Note: If you print a message,

LEVEL 2: Symbol table/ AST/ Semantic Rules modules work.

This means that you were able to implement the symbol table, AST and semantic rules handling 6‐10 semantic errors in total, but you could not implement type checking successfully (subject to verification).

The complete ERROR list (with line numbers first) should be printed on the console if the code is syntactically or semantically incorrect, else print a message [Essential when there is no error]

**Code compiles successfully..........**

**[B]. 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). The evaluation of your code will be done using the GCC and Ubuntu version as specified earlier.  Please ensure compatibility.

NOTE: The test cases t#.txt (1-10)and c#.txt (# :1-11 ) were provided only as a support and they may not be complete with respect to all aspects of the implementation. Students are advised to create own test cases for verifying the correctness of their compiler code.

**[C]. Execution:**

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

$./compiler testcase.txt code.asm

where compiler is the executable generated after linking all the files (your makefile should be absolutely correct). Also, the testcase.txt is the input source code (in the language you are implementing) file to be compiled. code.asm is the output file containing the assembly language code (NASM ‐64 bits compatible) equivalent to the input source code. All errors including lexical errors, syntax errors and semantic errors should be displayed on the console with line numbers. Testing of the Resultant code in assembly language will be done on a simulator NASM 2.14.02. The Instruction Set Architecture of NASM should be used for the target code.

### Stage 2: Submission guidelines

by [Vandana Agarwal .](https://nalanda-aws.bits-pilani.ac.in/user/view.php?id=5637&course=5761) -

1. Stage 2 code submission is due on April 12, 2023 (Wednesday), i.e. tomorrow from 6:30 p.m. to 8:00 p.m.

2. Teams, whose stage 1 submission of code was not plagiarized, can submit their stage 2 code.

3. Ensure that you are not submitting any piece of plagiarized code. Avoid last minute submission to avoid any problems due to network issues.

4. All team members should be present at the time of submission together to ensure correct files are included in the folder and that the code is not plagiarized.

5. Stage‐2 submission includes ALL modules of stage‐1 and stage‐2 as integrated compiler.

6. A submission with wrong group number will not be considered for evaluation.

7. Any one member of the team will be allowed to submit the code.

8. The Group folder (refer earlier details) must contain the code of front and back end both (stage1 properly integrated with the stage2).

(a) All definition and prototype files (names and number of files depends on your implementation)- should be available in your group folder directly

* lexerDef.h
* parserDef.h
* astDef.h
* symbolTableDef.h
* lexer.h
* parser.h
* ast.h
* typechecker.h
* symbolTable.h
* semanticAnalyzer.h
* codegen.h
* grammar file
* and all helper files.

(b) All Implementation files (names and number of files depends on your implementation)- should be available in your group folder directly

* lexer.c
* parser.c
* ast.c
* typechecker.c
* symbolTable.c
* semanticAnalyzer.c
* codegen.c
* driver.c
* and makefile etc.

(c) All test cases (should be available in your group folder directly)

(d) Support files (should be available in your group folder directly)

* + pdf of scan of your DFA design
  + pdf of your FIRST and FOLLOW sets computation
  + Text file of encoding scheme of your grammar
  + pdf of your semantic rules for AST

(e) Coding details proforma (Download the attached docx file, type in the details and submit)

You should submit all relevant files comprising of your STAGE 1 and STAGE 2 working code (with appropriate naming of the files)

9. Group folder should not have any subfolder in it.

10. Do not include \*.o and the executable file compiler in the above folder for uploading.

11. Each testcase file must have a prefix testcase followed by the number, followed by the extension 'txt', e.g. t1.txt, t2.txt etc. Submit only those testcases with which your code works well.

12. The testcases meant for code generation are named differently as c1.txt, c2.txt, etc.

13. Check the file dependencies to ensure correct working of the makefile. If the makefile does not work, it may carry a penalty in terms of deduction of marks. Ensure that the makefile is named as makefile (and no other name is used for it).

14. Ensure that the driver supports TEN options for execution as was mentioned earlier.

15. Verify that each file (except any input file you use in your code) has in its first few lines your Group number, Ids and names of all team members in comment. Do not forget to comment these. Ensure the compatibility of your code with GCC version as specified earlier. The code will not be evaluated if the \*.c files carrying your compiler code are without group number, Ids and names of all team members.

16. Refresh the course webpage to verify that you have not missed out any modifications/ errata uploaded till submission time. Go through all the details and ensure that nothing is left out which you would have preferred to incorporate in your code.

17. Compress the group folder using zip ( of type Group\_#.zip, where # represents your group number) to upload. [Do not use rar or tar, use only zip)

18. All team members must be present at the time of submission of their stage 2 code .

19. Fill up the given coding details pro forma with necessary details and submit along with other files.

20. Ensure that the code you are intending to submit is not plagiarized. Teams submitting plagiarized code will be penalized as per the rules.

21. Use any one of the team members Nalanda account to upload.

22. Submit your file Group\_#.zip (# represents your group number) through Nalanda. The link will be made available with effect from 6:30 p.m. tomorrow, April 12, 2023 and will remain open till 8:00 p.m on the same day.

23. Avoid last minute submission. If you find any problem in uploading the files through Nalanda, please do not panic and do not send any code through email. Code sent through email will not be accepted . It is advised that you submit your code at least 15 minutes before the closure time of submission portal so as to avoid any network congestion.

24. In case of any problem in uploading your file, you can contact me over an email.

25. If you choose to avail the life line ( if not used up during stage-1 submission), you can submit the code on April 13, 2023 (Thursday) without any loss of marks. Those who are availing lifeline, should submit the coding details pro forma on the day of submission. The link for code submission (With Lifeline) will be open during 6:30 p.m. to 8:00 p.m on April 13, 2023(Thursday).

26. In case of doubts, please contact me through email before 1:00 p.m. tomorrow.