Course Instructor: Dr. Shounak Chakraborty, Asst. Professor, Dept. of CSE, IIIT Guwahati

Course Objectives. Starting with some prime file handling programs in C in addition with some basic concepts in programming, this lab course will mostly focus on the construction of a compiler with considering C as a host language. However, the traditional automatic compiler construction tools like Lex, YaCC etc. will also be the part of this course with in-depth elaboration of the individual concepts. The associated theory course CS320 will be the basis for this lab course. The implementation of theoretical knowledge discussed in the class-room will be implemented in this laboratory. Overall through this course, you will learn how to engineer a Compiler from an implementation point of view with deep understanding of its associated pros-n-cons.
By end of this course, you will learn the following-

- necessity of module based design of a Compiler
- implementation nitty gritties of several compilation steps
- step-wise and as a whole performance analysis of a Compiler
- optimisation of compilation steps wherever required
- sound understanding of interface between Computer Architectures, Operating Systems and Compiler
- data structures associated with Compiler Design

Course Policies. Attending the lab session (*CSE Lab 1, 2-5 PM, Thu*) is mandatory. The lab session will be as follows:

- There will be a small session for the overview of the problems those you are going to implement in the respective sessions
- Either instructor or any student (selected randomly!!) can be asked to discuss the prime concepts those are necessary for implementation
- The session will be more interactive than our theory classes
- However, the implementation will take place after that, and on completion you have to show the same and be evaluated by your TA
- It is expected that, the lab-experiments will be completed and evaluated in the same session, but, there may have some special cases where some extra time will be provided to complete the same
- There will be mid-sem as well as end-sem exams but the assignments will carry the principal amount of points
- Late-submission policy:
  (1) delay $<$ 24 hours from the deadline will deduct 10% (of the obtained marks)
  (2) delay $\geq$ 24 hours & $<$ 48 hours from the deadline will deduct 20% (of the obtained marks)
  (3) delay $\geq$ 48 hours & $<$ 72 hours from the deadline will deduct 30% (of the obtained marks)
  (4) delay $\geq$ 72 hours will deduct 100% (of the obtained marks)

For medical emergency or any other unavoidable circumstances, students have to inform the instructor before the commencement of examinations/tests/assignment deadlines. The justification for the same will be needed with proper documentations for taking make-up steps (if possible). In that case, make-up tests, assignments or examinations will have a bit different format than the usual ones and will be conducted as per instructor's convenient time-slots.

Please go through Table 1 for grading policy of this course.

Table 1. Grading Policy Details

Assignment I	6%
Assignment II	10%
Assignment III	10%
Assignment IV	10%
Mid-Sem	16%
Assignment V	6%
Assignment VI	7%
Assignment VII	11%
End-Sem	24%

Please note that, there will be zero tolerance in case of cheating. The institute's policy for academic dishonesty will strictly be enforced.

Course Overview. The practical practices/implementations will be done in respective lab sessions subject to completion of the topic in the prior lectures(s).

Course Prerequisites.

(1) Formal Language & Automata Theory
(2) Basics of Computer Programming (Preferably C/C++)
(3) Computer Architecture
(4) Operating Systems & Familiarisation with Linux environment(s)

Course Materials. There is no particular book(s) that can be recommended as a text for this course, but you may use the following book, that may be useful for you to learn Lex and YaCC:

- **[OR]** J. R. Levin, T. Mason, & D. Brown, "Lex & YaCC", O'Reilly, 2nd Edition.

The links for the others on-line materials (e.g. some relevant conference/journal papers) will be provided subject to their requirements.

Topics will be covered in this course.

(1) File I/O: A practice session with an implementation of DFA to detect some predefined patterns, symbols, comments, etc. A realisation of scanner/lexical analyser.
(2) Hands-on data structures: Implementation of symbol tables.
(3) Lex Tool for implementing lexical analyser.
(4) Token to Parse Tree formation in C, with the help of a particular CFG.
(5) Generate YaCC specification for a few syntactic categories.
    - Program to recognize a valid arithmetic expression that uses operator +, - , * and / along with operator precedence.
    - Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
    - Implementation of Calculator using Lex and YaCC.
(6) Familiarisation with BNF & implement the BNF rules in YaCC. Generate an Abstract Parse Tree. Inclusion of S-attributed grammar.
(7) Implementation of Type Checking in with LL(1) parser along with L-attributed grammar.
(8) Control & Data Flow Analysis.
(9) Implementation of storage allocation and deallocation strategies (including garbage collection).
(10) DAG construction.
(11) Machine Independent Optimisation of 3-address code and translation of the same to a specific assembly level code, and Machine Dependent Code Optimisation.

(12) Register allocation along with an approximation of the graph colouring problem.
(13) An introduction to LLVM.

Teaching Assistants.

- KHANJAN CHANGAI BARUAH (email: khanjan099@yahoo.com)
- MRIDUL HAQUE (email: mridulh7@gmail.com)

They will evaluate your lab assignments/assessments according to the instructions given to them for the respective assignments/assessments.

****

Lab 1 (Practice Session). You will be given a file that contains a C program, where we will have a set of key-words (*if, else, case, switch, int, char, float, struct*), a set of relational operators ($>$, $<$, $==$, $>=$, $<=$, $!=$), a set of numbers (*10, 109.8987, 0.776, 1110561*), and a set of user-defined variables (*a1, x, yy, z59*). Build up a source code in C to implement the DFA that can detect the above mentioned items and can throw errors in mismatch.
[**You can get help from the TAs for developing the codes, but, it is expected and strongly encouraged that, you will develop the logic for implementing DFA.**]

Lab 2 (Assignment I [Hard Deadline 22:30 hrs, 8-Aug-2018, Time Zone: GMT + 5:30 hrs]). This assignment will give you the flavour, that how a Lexical Analyser works in traditional compilation process. Suppose you have a file that contains C code. Now, write a program in C/C++ which can read the contents of the file and will generate tokens, the prime outcome of Lexical Analysis Phase. While implementing this, you have to follow the design patterns as given below:

- Remove single & multi-line comments.
- Identify the keywords individually.
- Recognise the variables/identifiers and put them in a table. The format will be as follows: < name, type, size>.
- Identify both integers & floating point numbers.
- The literals in the printf() and scanf() need to be detected judiciously.
- Once tokens are classified (as discussed in the lecture of CS320) and ready with the respective table entries for your identifiers, put all the tokens in a single queue and print them in a file. Each line of your output file will contain a single token. At the end, also print the contents of the Table.

Some rules that your have to follow:

(1) Your implementation must contain a single or a set of DFAs, implementation of a Table as asked above, and a queue that will contain the tokens.
(2) The recommended language for implementation will be C/C++, but you may implement in some other languages without violating the implementation guidelines.
(3) Do not use recursion in your code. Use of at least one recursion will award you *ZERO* for the whole assignment.
(4) Any optimisation that you will make will give you an extra credit. Any assumptions that you have done while implementing must be mentioned in your report as well as in your code (use comments).
(5) Prepare a note/report (in Latex) and submit it with your assignment. This note will describe your code in brief and justify the optimisation (if any), and also discuss to the points in terms of robustness, correctness and asymptotic complexity (both space &

time). Your report should not be more than 2 pages. Marks will be deducted for extra & irrelevant writing.

(6) You can ignore the headers and any pre-processive directives in the input file. For your implementation, you can assume that, all of the words are separated by at least one white-space. If more white spaces are there, keep in mind, your analyser must detect it as a single white space, and tokens will be formed accordingly.

(7) You have to implement this in a group of two. You are free to choose your partner as you wish. The credits will be given to both equally, without any biasness. While submitting, your report must contain the names and roll numbers for both of you, and there will be a single directory which will contain both the files, and directory should be named strictly in the following manner: < Roll1_Roll2_AS1 >.

Send your codes, and report (in pdf) at *cs321iiitg2018fall@gmail.com*. In the subject line, write as follows: "CS321_AS1_Submission". Note that, TA will not help you in this assignment. The evaluation will have a strategy for partial marks as per your works done.

Lab 3 (Practice Session). In this practice session we are going to work with Lex tool, an automatic scanner genrator. You are hereby hence asked further to go through the Chapter 1 of the book [OR] mentioned in this document. Try to develop a scanner for detecting English words and categorise them in proper Parts of Speech. Next, enhance the same towards building a scanner with an entry in the symbol table for the unknown words. You can now try to develop the idea that, how can we make it work for lexically analysing any C code with an implementation of the symbol table. However, your TA will help you for understanding the Lex codes, but it is expected that, you will understand the basic Lex syntax by yourself or together with your peers.

Lab 4 (Practice Session). Build-up a code in Lex to read a C program from a file and count the number of characters, words and lines. Print all of these parameters as output. Now, define rules/patterns in your Lex code to detect the keywords, identifiers, delimeters, brackets etc. You can ignore the pre-processive directives. Also, try to detect and eliminate the comments from your input C code. You are advised to use the book [OR] as mentioned above for your reference. It is recommended/sugested that, before attending this lab session you may go through the contents of Chapter 2 (from [OR]) for quick implementation of your Lex code. In case of any difficulties please ask TAs or peers for help.

Lab 5 (Assignment II [Hard Deadline 11:45 hrs, 28-Aug-2018, Time Zone: GMT + 5:30 hrs]). Consider the C program given in Figure 1. This code contains the following types of sentences, that we need to parse:

- Declarations & Function Definition
- Expressions
- "for" loop
- Conditional Statement
- Return Statement

For each of these types of sentences, we need to now generate a grammar, which we will be using towards engineering a parser (top-down). Firstly, the parsing table needs to be built, and then we need to check whether this underlying grammar is a strong-LL(1) one. If it is not, then a few assumptions may be included to make it a strong-LL(1). The respective grammar's production rules are shown in Figure 2. Note that, words start with an upper-case letter are Non-terminals,

```
int main ( ) {

 int a , b , c , i ;

 a = 45 ;
 b = 8 ;
 c = a + b * c ;

 for ( i = 0 ; i < 50 ; i ++ ) {
  if ( i % 2 == 0 ) {
    a = 0 ;
  } else {
    a = 1 ;
  }
 }

 return 0 ;
}
```

FIGURE 1. A sample C code to be parsed after Lexical Analysis.

and start symbol is "MD". Feel free to change the grammar in case you need, and note down all of your changes/assumptions in your report, guidelines of which is given below.
Now, do the following:

- Prepare a parsing table for your LL(1) grammar. [Get it ready manually.]
- Implement the parsing table in C. [Put it in some 2D array kind of structure so that your code can able to parse. Note that, if you are able to do this table generation automatically (by implementing it in C) from the underlying grammar, you will be awarded extra credits.]
- Now, implement the top-down predictive parsing algorithm in C to parse the input program, which must be written as a collection of tokens. [Must be implemented completely in C.]
- On successful parsing of a sentence print "Success", else print "Error" with a brief message. But do parse whole program, not a particular sentence.

```
MD -> type id [ F | [[, id]* ;] ]

F -> () { Stmts }

Stmts -> MD | Ret | Cond | For_loop | Job

Job -> id [ Assg | Expr | C_E ] ;

Assg -> [ lit | Expr ]

Expr -> op [Expr | lit | id ]

C_E -> cond_op [ lit | id ]

For_loop -> for ( id Assg ; C_E ; id Inr) { Stmts }

Inr -> ++ | --

Cond -> if ( [ id | Expr ] cond_op [ id | lit ] ) { Stmts } C'

C' -> NULL | else { Stmts }

Ret -> return [ lit | id | Expr ] ;
```

FIGURE 2. Production rules for the CFG.

In your report, show the parsing table and write briefly about the implementation. Note that, compute FIRST() & FOLLOW() functions yourself and do not implement those in C (do not

include in your report also). Just prepare the parsing table and implement the parsing algorithm with a stack. Use the program of Figure 1 to show your output.
Some rules that your have to follow:

(1) Input must be a set of tokens.
(2) The recommended language for implementation will be C/C++, but you may implement in some other languages without violating the implementation guidelines.
(3) Do not use recursion in your code. Use of at least one recursion will award you *ZERO* for the whole assignment.
(4) Any optimisation that you will make will give you an extra credit. Any assumptions that you have done while implementing must be mentioned in your report as well as in your code (use comments).
(5) Prepare a note/report (in Latex) and submit it with your assignment. This note will describe your code in brief and justify the optimisation (if any), and also discuss to the points in terms of robustness, correctness and asymptotic complexity (both space & time). Your report should not be more than 3 pages. Marks will be deducted for extra & irrelevant writing.
(6) You can ignore the headers and any pre-processive directives in the input file. For your implementation, you can assume that, all of the words are separated by at least one white-space as given in the sample C code.
(7) You have to implement this in a group of two and *strictly NOT with your partner of Assignment I.* The credits will be given to both equally, without any biasness. While submitting, your report must contain the names and roll numbers for both of you, and there will be a single directory which will contain both the files, and directory should be named strictly in the following manner: < Roll1_Roll2_AS2 >.

Send your codes, and report (in pdf) at *cs321iiitg2018fall@gmail.com*. In the subject line, write as follows: "CS321_AS2_Submission". Note that, TA will not help you in this assignment. The evaluation will have a strategy for partial marks as per your works done. Submissions, those will be sent to my email-id(s), will not be evaluated, so do not send anything to me.

Lab 6 (Practice Session). This is an introductory session for learning YACC. Towards this, you have to build-up a code in YACC for an infix primitive calculator. Before that, a small discussion will take place on YACC tool and Lex (we have already seen in our last couple of lab sessions). The corresponding CFG along with other inputs to YACC will be discussed briefly, and you are advised to go through the book (Chapter 3) [OR], for details. You can also visits the available on-line contents (so many!!) for further learning of this tool. Note that, Lex is necessary for today's lab session as tokens generated by this will have to be sent to the parser generated through YACC.

Lab 7 (Practice Session). In this lab session, you will be given a lex-yacc code which if executes can generate a simple parser for assignment operations. Your job will be to develop a grammar (specifically the production rules) towards inclusion of following operations to your parser:

(1) Segrerate between keywords and identifiers.
(2) Check the balanced parenthesis.
(3) Declarations of the variables, and put them in symbol table on their first occurance.
(4) Throw error messages properly, like it is implemented in the given code.

The production rules those have already been used in your given code are as follows:

- Stmt_list − > Stmt | Stmt_list ; Stmt

- Stmt − > Variable assignop Expression
- Variable − > id | id [Expression]
- Expression − > Simple_expression | Simple_expression relop Simple_expression
- Simple_expression − > Term | Simple_expression addop Term
- Term − > Factor | Term mulop Factor
- Factor − > id | num | (Expression) | id [Expression] | not Factor

Table 2 contains all about the token specifications.

Table 2. Token Specification

Token Name	**Pattern/Description**
ID	- Starts with a leter followed by letters, digits or underscore. - Max length will be of 5 characters. - Cannot be any of the keywords of C.
Num	- Integers - real numbers that include a decimal point and at least one digit to the left and right of the point.
Relop	"<", ">", ">=", "<=", "==", "! ="
Addop	"+", "−", "||"
Mulop	"∗", "/", "%", "&&"
Assignop	"="
Not	"!"

Lab 8 (Assignment III) [Hard Deadline 06:30 hrs, 06-Sep-2018, Time Zone: GMT]. On top of the (Lex-Yacc) code, that has been given to you for Lab 7, you have to add the functionalities of an Operator Precedence Parser. Specifically, this parser will only work for any kinds of mathematical expressions/equations, with proper implementation of the classical precedence relations. Inserts the brackets after compilation, wherever it is required, and print the outputs/modified code. So, finally, your code will be working as follows:

- The lexer will generate the tokens (Try to print them in a file).
- Variables will be in the symbol tables along with their respective types. On their first occurance they will be placed in the symbol table.
- Write a grammar for the operator precedence relations and the rules of your grammar have to be implemented in your parser.
- Your parsing algorithm/program should able to do the precedence relation checking and do throw error wherever needed.
- The balanced parenthesis checking should be done (Do not include *for, while* etc. loops, but *if-else* structures has to be implemented along with the (function) scopes).
- Check if the declaration statements are written according to the syntax or not.

You can also print your outputs on the console but it is recommended to print them in files. In case of parsers, print "success" on successful parsing of the statements, else throw error messages. Note that, assignment grammar that has already been there must be present in your implementation.
For this assignment, no restriction on the choice of your partner with a group size of 2. Prepare either a make-file or a shellscript to run your whole code. You have to submit your lex and yacc code along with a soft copy of your report. The report should not be more than 2 pages and must contain brief descriptions about the grammars and implementations.

Lab 9 (Practice Session). We have already seen the attributes and Syntax Directed Translations in our theory classes. Moreover, inclusion of these attributes in Yacc has also been discussed in our lecture sessions. Now, in Lab 9, we will see how can we implement a part of type systems (for C) in Yacc on top of the code distributed in the last lab session (Lab 7). As Yacc supports bottom-up parsing technique, hence, inclusion of synthesized attributes will be very trivial and can also be implemented along with the parser.
Consider the grammar given in Figure 3. Now implement the part of this grammar that just supports the statements like this: " *int a, k, c; float b, m45;* ". Now, identify the following-

- if same variable is declared multiple times or not
- modify your expression grammar also to check the type mismatch [note that, for this one, actions of SDT have to be included in your parser code]

```
DeclarationList     →  DeclarationList Declaration
                    |  Declaration
Declaration         →  SpecifierList InitDeclaratorList ;
SpecifierList       →  Specifier SpecifierList
                    |  Specifier
Specifier           →  StorageClass
                    |  TypeSpecifier
StorageClass        →  auto
                    |  static
                    |  extern
                    |  register
TypeSpecifier       →  void
                    |  char
                    |  short
                    |  int
                    |  long
                    |  signed
                    |  unsigned
                    |  float
                    |  double
InitDeclaratorList  →  InitDeclaratorList , InitDeclarator
                    |  InitDeclarator
InitDeclarator      →  Declarator = Initializer
                    |  Declarator
Declarator          →  Pointer DirectDeclarator
                    |  DirectDeclarator
Pointer             →  *
                    |  * Pointer
DirectDeclarator    →  ident
                    |  ( Declarator )
                    |  DirectDeclarator ( )
                    |  DirectDeclarator ( ParameterTypeList )
                    |  DirectDeclarator ( IdentifierList )
                    |  DirectDeclarator [ ]
                    |  DirectDeclarator [ ConstantExpr ]
```

FIGURE 3. A subset of C's Declaration Syntax.

The input file will be given with this code to check the results of your implementation.

Lab 10 (Assignment IV) [Hard Deadline 12:30 hrs, 14-Sep-2018, Time Zone: GMT].
In this assignment, you have to extend the practice session of Lab 9 in the following manner:

- Consider the whole code written in Figure 3, and implement it in Yacc on top of your code written for Assignment III (Lab 8).
- Next, write actions for the following-
  (1) Check the types for each element of the expressions (or assignments, if any) and throw messages on errors.
  (2) Include cost of evaluation of each expressions. Assume, both addition and subtraction have cost of 5 individually, where multiplication and division have costs of 9 and 12, respectively. Do it as simple as possible. Also if you have any load or store operations, assume average costs of 3 and 5 for each of them, respectively.
- Include avoidance of multiple declaration of variables by implementing symbol table where each entry includes name and type of the respective identifier.

You may keep your group same as you had for Assignment III. Also submit a report of one page, which will just highlight the bullet points of your implementation.

Lab 11 (Practice Session). In this lab session you will be familiar with the implementation of *Symbol Table*. A source code written in C will be provided to you (Exp.cpp), where a set of variables are declared along some functions. Segrerate the functions, arguments & variable names by some flags and make their entry in the symbol table as it was discussded in the class. On first occurance of each and every identifiers, you have to make an entry in your symbol table properly with apt distinctions. Moreover, your symbol table must be implemented in some hierarchical manner, with a set of pointers by which connections will be established across the hierarchies. You may consider Figure 4 as a reference for your implementation. It is also advised to implement the whole code in C/C++. If you are interested to implement in any other language, note that, you will have to integrate the same code with the semantic analyser that you will implement in your next assignment.

Name	Result_Type	Par_List_Ptr	Var_List_Ptr	#Parameters	GVar_Flag

Global Symbol Tables: for Functions and Global Variables

Name	Type	Par_Tag	Decl_Level

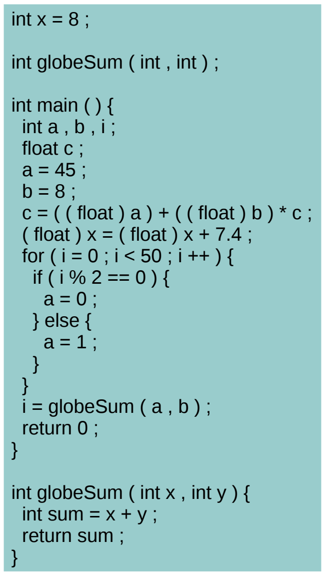
Local Symbol Tables: for Local Variables and Function Parameters

FIGURE 4. Symbol Table Structure.

Lab 12 (Assignment V) [Hard Deadline 12:30 hrs, 09-Oct-2018, Time Zone: GMT + 5:30hrs]. The *Symbol Table* implemented in *Lab 11* will have to be integrated with the code that you have developed upto your Assignment II. This indicates that, your developed Scanner and Parser will have this new symbol table, that you can easily use for the semantic analysis. The semantic operations have to be now implemented in your top-down parser in the following manner:

- Add rules for type checking by using inherited attribute and associate them with your parser. (Just attribute you have to use, no other parameters.)
- Design the following for checking of types, once your rules have been integrated with your parser:
  (1) Access all of your variables and functions (you may include main()) from your symbol table.
  (2) Check the types of all of your variables (both local & global) and throw errors, if found anything in their usages.
  (3) Check the functions' return types along with its other associated parameters (like, arguments etc.) according to their declarations, else throw errors.
  (4) Check for *type-cast* cases and throw errors in respective cases.

In this assignment, size of each group will be of 3. So, there will be total 9 groups, as your class-strength is 27. No restriction on choice of your group members. Along with your code you will have to submit a report of 2 pages (strict page limit), in which you have to write the brief about your code, implementation details, and optimisation that exclusively you have done. You will be awarded extra credit for small, simple and complete implementation. Consider the input program given in Figure 5.



```
int x = 8 ;

int globeSum ( int , int ) ;

int main ( ) {
  int a , b , i ;
  float c ;
  a = 45 ;
  b = 8 ;
  c = ( ( float ) a ) + ( ( float ) b ) * c ;
  ( float ) x = ( float ) x + 7.4 ;
  for ( i = 0 ; i < 50 ; i ++ ) {
    if ( i % 2 == 0 ) {
      a = 0 ;
    } else {
      a = 1 ;
    }
  }
  i = globeSum ( a , b ) ;
  return 0 ;
}

int globeSum ( int x , int y ) {
  int sum = x + y ;
  return sum ;
}
```

FIGURE 5. A sample C code.

Lab 13 (Practice Session). This lab session onwards, we will be focusing on the designing of back-end of the compiler. Towards this, in this first session of back-end generation, we are going to hand-on the formation of DAG from a given mathematical expression. Construction of DAG will be done by using couple of functions: *Leaf(id, [entry of id])* and *Node('op', l, r)*. Former one is used to generate leaf nodes while encountering an identifier in the expression, and it immediately points to the corresponding symbol table entry (Note that, in our previous lab session we have already created symbol table). The latter one frames the mathematical operation which will be performed by involving a pair of operands pointed by *l* and *r*. While doing this lab experiment, assume that, all of the identifiers are already placed in your symbol table. Now write a code to generate a DAG (and associated temporary structures, if needed) for the following expression:

$$a + a * (b - c) + (b - c) * d$$

Lab 14 (Practice Session). Consider the C-code given in Figure 1. Prepare a syntax tree for the whole program, excluding instructions for function definition and, declarations of variables. Once your tree is ready for the set of instructions, try to write down the semantic rules that can help you in generating 3-Address Code for your program. Towards this, *gen(), newTemp(), etc.* functions need to be used wherever required. Successful implementation of this whole process can generate a 3-address code generator, but, this implementation will incur a lot of designing/engineering issues and hence, will be time consuming undoubtedly. Therefore, a small part of the same can be implemented which can give you a flavour of designing a 3-address code generator.
Hope all of you have outputs of the lexical & syntax analyser for the same C-code. From the outputs of these analysis phase, can you now write a code in C to detect the C-instructions individually and can also trigger an output on requirement of temporaries in 3-address code? Note that, for individual identifiers you have to collect their lexemes from the respective symbol table entries (developed in Lab 11). Also, try to detect the control/branch intructions and their defined spans/scopes in your input C-code. If possible, detect and put *goto* statements wherever required.

Lab 15 (Assignment VI) [Hard Deadline 23:59 hrs, 27-Oct-2018, Time Zone: GMT + 5:30hrs]. Consider the code snippet given in Figure 1. Particularly, for this code, already you have engineered all of the three analysers individually and this code is now error free, as expected. According to our theoretical study, we have seen that, bottom-up parsers along with a set of semantic rules are needed to generate the three address code at *Intermediate Code Generation*. In our design, a top-down predictive parser has been implemented, hence, realistic implementation along with the backtracking can not be possible here as we studied. Hence, in this assignment, you have to consider either the original source code (from your input file) or the output of lexer to generate the three address code. Towards that, you have to implement the following in C/C++:

- Analyse the whole source code and determine number of temporaries required prior preparing the three address code.
- Identify the instructions (may be line number), for which you need a temporary.
- For each line where you need a temporary, write the three address code for the same and write them in a temporary file.
- Now consider the branch/control instructions. Write algorithms for individual controls to generate a three address code. Use proper data structures for this, because this is practically the heart of your whole implementation. Implement the algorithm in C/C++ now.
- Now, combine the above two steps to generate the final three address code, and print them in a file.

Make any assumptions that will be needed for your implementation, and mention the same in your report. Report must contain the algorithm written for branch instructions. Properly justify the cause(s) for using one or more data structures for your implementation. Analyse the time complexity of your algorithm and include it in your report. Special bonus marks will be given if you can state any solution for further improvement of your algorithm in terms of time complexity. Your report should not be more than 2 pages. Note that, you have to work in a group of three and this time your group should be same with Assignment V.

Lab 16 (Practice Session). In this lab session you have to be familiarised with the implementation of garbage collector. By using random number generator of C/C++ along with a hash function (make sure your random number varies between 0 and 127), generate a set of random numbers and prepare a table with four columns. Each row of your table will contain four numbers, where first column denotes the node which refers the other three. All these numbers are some arbitrary memory locations requested by some program(s) during execution. Lets assume that, the numbers which are present only in the first column at most once are considered to be the root node. Now, identify the *root set* to fill-up the *Unscanned* list and prepare the reachability graph by implementing it through linked-list. Note that, for each node you must have to implement a *reached_bit* for implementing *Mark-n-Sweep* algorithm. Firstly, implement the Mark Phase and show the output; make sure *reached_bit* is set for all the nodes as per its applicability. Now, by implementing Sweep Phase, scan the whole heap and identify the possible unreachable nodes, if any, which can be thrown into the free pool of heap memory. Print the *Free* list finally.

Lab 17 (Practice Session). This session will make you familiarised with the generation of basic blocks & flow graph. Towards that, you will be given a three-address code for a source program, and you have to write an algorithm to generate basic blocks & flow graph from the three-address code. Implement the same in C/C++ and show the output. Just mention in your output about the predecessor and successor of the nodes in flow graph.

Lab 18 (Practice Session). Code generation and optimisation are the final steps of a traditional compilation process. In this session, you will have to use the output that you have generated in your last lab session (Lab 17), and one additional code library for *RISC-V* architecture will also be provided. Now, it is your job to map your 3-address code written inside the basic blocks to the proper instructions from the ISA so that an optimised code can be excogitated. Towards that a brief about the code generation will be discussed in your lab session, and accordingly you will have to complete the task assigned for today.

Lab 19 (Assignment VII) [Hard Deadline 23:59 hrs, 14-Nov-2018, Time Zone: GMT + 5:30hrs]. Consider the ISA of *RISC V* which should be your target architecture, and your mini-compiler should generate code for the same. Towards that, you need to consider the code snippet given in Figure 1, and generate the target code for the same. In this assignment, you have to build-up a code generator, as we discussed in the last lab, which should take 3-address code as input and produces target code as per *RISC V*. As it is a RISC architecture, the size of ISA is small enough and complex arithmetic/logical operations are not included in this, hence, in your code generator implement the algorithm which can generate codes for these missing operations. Note that, implementation of the same has to be optimised. In a nutshell, you have to do the following:

- Take 3-address code as input and make the basic blocks.
- For each of these basic blocks generate the target code and print the same in a file. (Assume you have infinite number of registers.)
- Implement inside your code generator which can help you in generating target code for some complex operations.
- Count the usage of individual registers inside the basic block and print it.
- For each basic block, sort the registers in non-increasing order and print them.
- Find out and print any false dependency that can be solved by register renaming. Do not write the code for renaming, but just identify and print the same.

In your report of 2-pages, write briefly about your implementation and optimisation, if any. Make sure, starting from the lexical analysis upto code generator everything is running sequentially as a part of a complete compilation process. For that write a “makefile” or “shellscript” to run all of them in proper manner. Keep your group unchanged that you had for your last assignment.
