University *of* Alexandria

Faculty of Engineering

Computer and Systems Engineering Department

Compilers

Phase 2

Predictive Parser

Names:

Ahmed Ibrahim Hafez	(04)
Ahmed El-Sayed Mahmoud	(05)
Amr Gamal Mohamed	(45)
Mohamed El-Sayed Mohamed	(54)

• ***Data Structures***

The used build-in data structures are: Vector, Map

The implemented data structures are: Rule, Symbol

1- Symbol.cpp

Every grammar rule consists of symbols and the symbol object holds the symbol type and string.

- enum Type {
      Terminal,
      NonTerminal
  };

  The type of the symbol either terminal or non-terminal.
- string symbol_string: The symbol name.

2- Rule.cpp

It holds the rule name(non-terminal) and productions (ORing of Concatenating terminals and non terminals) e.g. X → A b | c D.

- vector<vector<Symbol*>> productions: Outer vector is OR tokens e.g. A | B. Inner vector is the concatenated tokens e.g. A = '(' 'id' ')'.
- struct RuleStringStruct { // e.g. X → a b | c d
  string lhs; // X
  string rhs; // a b
  };
  This struct holds the left hand side and right hand side of the rule as string.
- String rule_name: The rule name e.g. A, B,..

3- CFG_Parser.cpp

It is responsible for parsing the context free grammar into rules and each rule consist of name(non-terminal) and productions (ORing of Concatenating terminals and non terminals).

- vector<Rule*> rules: a vector of rules after parsing the contxt free grammar file.
- vector<Symbol*> terminals: vector of all terminal in the grammar.
- vector<Symbol*> non_terminals: vector of all non-terminals in the grammar.

4- Generate_LL_Grammar.cpp

This is responsible for generating LL(1) grammar after eliminating the left recursion and left factor if exists.

- vector <Rule*> rules: The rules before and after elimination of left recursion and left factor.

vector<Symbol*> non_terminals: The non-terminals before and after elimination of left recursion and left factor.

5- Parsing_Table.cpp

It is responsible for making the parse table after getting the first and follow of each nonterminal in the grammar.

- vector<Rule*> rules: a vector of rules after parsing the contxt free grammar file.
- vector<Symbol*> terminals: vector of all terminal in the grammar.
- vector<Symbol*> non_terminals: vector of all non-terminals in the grammar.
- unordered_map<string,set<string>> first: contain set of terminals which are the outcome first of the corresponding non-terminal.

- unordered_map<string,set<string>> follow: contain set of terminals which are the following of the corresponding non-terminal.
- unordered_map<string,unordered_map<string,vector<Production_Rule*>>> parse_table: contain all the entries(productions) that a non-terminal can go to under certain terminal.

6- Production_Rule.cpp

It considered as a production to be added to an entry in the parse table.

-vector<Symbol*> production: one of possible outcome of RHS of CFG rule.

7- Matcher.cpp

Here the passed tokens are being matched as input with the parsing table generated earlier to provide output derivations from the parser.

- Vector<vector<string>> table: representation of the parsing table entries to be used.
- Unordered_map (string, int) terminals: representation of terminals as string and number pairs having same order as the parsing table.
- Unordered_map (string, int) non_terminals: representation of non terminals as string and number pairs having same order as the parsing table.
- Stack <string> validation_stack: a stack used to hold the production rules while matching the input tokens.

• *Algorithms & techniques*

1- Parsing the context free grammar file:

- Read the file as one string and parse it.
- Parse the string by the '#' character as it precedes every grammar rule. And generate a new Rule of Symbols after parse it.
- For every line string of rules, clean the string from white spaces, adjust lambda format, remove left and right spaces.
- Split the rule about the '=' sign and set the left hand side (lhs) and right hand side (rhs).
- Split the lhs about spaces into non-terminals and terminals. The terminal is identified with '' and productions consist of OR and Concatenations (vector of vector).
- If the rules doesn't exist in the vector of rules, add it. Else, skip.

2- Detect and check if the grammar is LL(1):

- First, substitute every non-terminal with its productions to handle non-immediate left recursion or non-immediate left factor.
- Second, check if there is immediate left recursion or left factor.
- If there is a left recursion or left factor, return false. Else, return true (LL(1)).

3- Eliminate Left recursion and Left factor:

<u>For left recursion:</u>

- Substitute for every non-terminals with its productions.
- Eliminate immediate left recursion.
- If the first symbol in every OR productions is equal to the rule name $A \rightarrow A\ \alpha \mid \beta$, then make a new rule A' with productions

α A' | ε , add it to rules vector, take the remaining concat productions and add it to the current rule A → β A'

For left factor:

- Substitute for every non-terminals with its productions.
- Eliminate immediate left factor.
- If the first symbol in every OR productions is equal to the following OR production A → αβ1 | αβ2 , then make a new rule A' with productions β1 | β2, add it to rules vector, take the current symbol and add it to the current rule A → β A' .

4- Calculating First:

For each non-terminal search it in all rules' LHS till it found then we go and get its firsts by getting first of all its possible productions, if the first symbol in a production was terminal symbol then add it to LHS non-terminal firsts', but if it was non-terminal then we mark the LHS non-terminal as visited then check if the production symbol was visited or not, if it was visited then we get its firsts' else we calculate them, then add it to the LHS non-terminal firsts', if there was epsilon in it then we will repeat the non-terminal part for the next symbol if it was non-terminal but if next symbol was terminal then add it to firsts'. Then continue to next production or next rule.

5- Calculating Follow:

For each non-terminal search it in all rules' RHS (productions) till it found then we go and get firsts of next symbol, if it was terminal then add it to non-terminal's follow, but if it was non-terminal then add this non-terminals first to follow, if there was epsilon in its first it will not be added to follow but then we will repeat the non-terminal part for the next symbol if it was non-terminal but if next symbol was terminal then add it to follow, if we reached the end of the production then we will add the follow of the LHS if it was calculated before else we will mark the non-terminal as visited and check if the LHS was visited or not, if it was visited then add its follow to non-terminal's follow, else go and calculate LHS follow then add them to non-terminal's follow.

6- Making Parse Table:

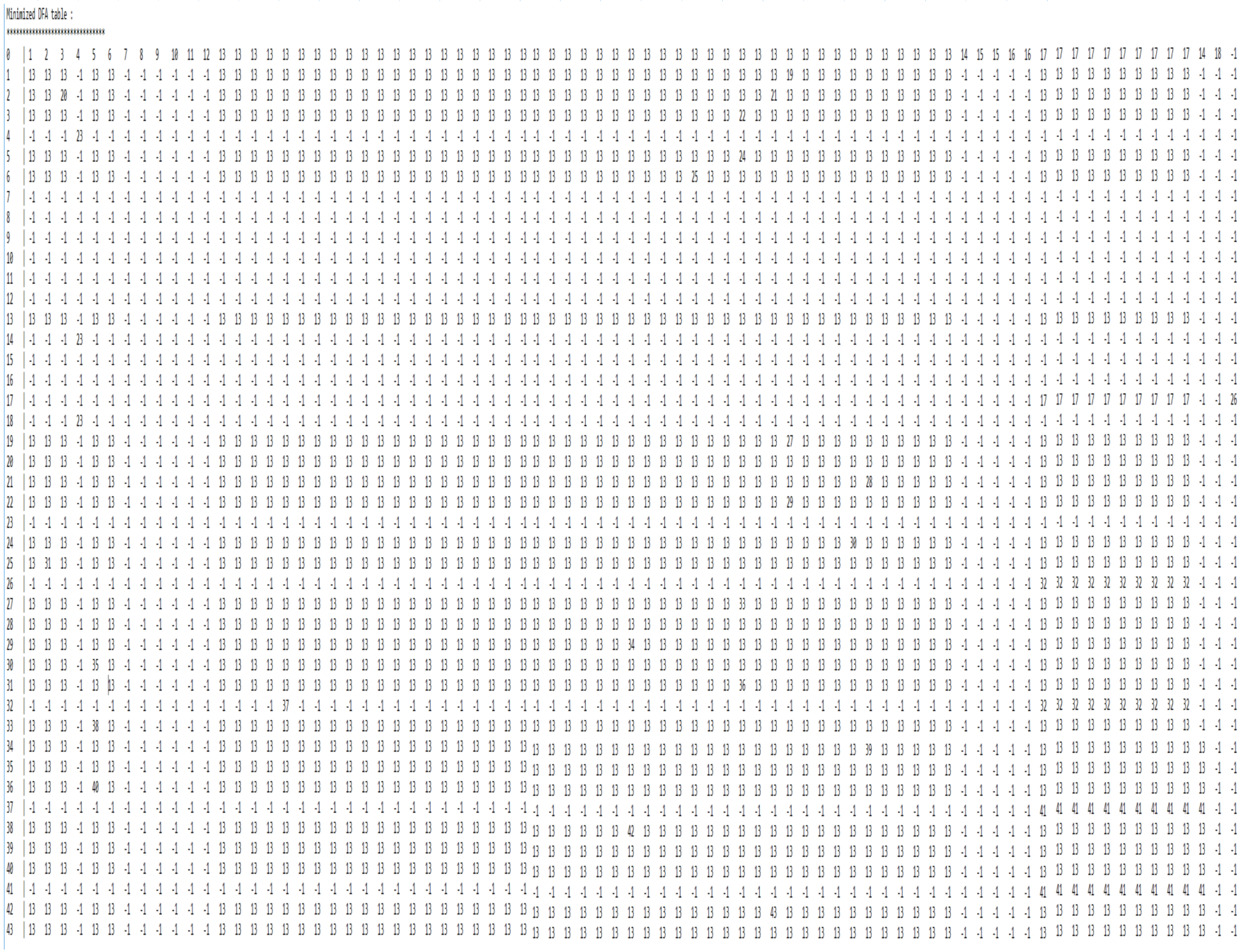
- First, Go to each production in a rule and get its first and add this production to the row of LHS under each terminal in the firsts got.
- Second, if there was epsilon in first of a production then go and add this production in the follow of the LHS.
- Third, after finishing all productions of the rule if there wasn't an epsilon in the first of LHS then put synchronize under the empty entries of the follow of the LHS.
- Repeat for all rules.

7- Parse the input tokens with the parsing table

- First, push the '$' on the validation stack then push the starting symbol on it.
- Second, iterate over the input tokens trying to match the top of the stack with the read token.
- If they don't match then try to find the suitable derivation from the parsing table and substitute it with the top of the stack.

Continue the process until either the validation stack is empty or the loop iterated over all input tokens, generate errors if necessary.

- **Transition Diagrams**
  - look at tables.txt file in lexical folder

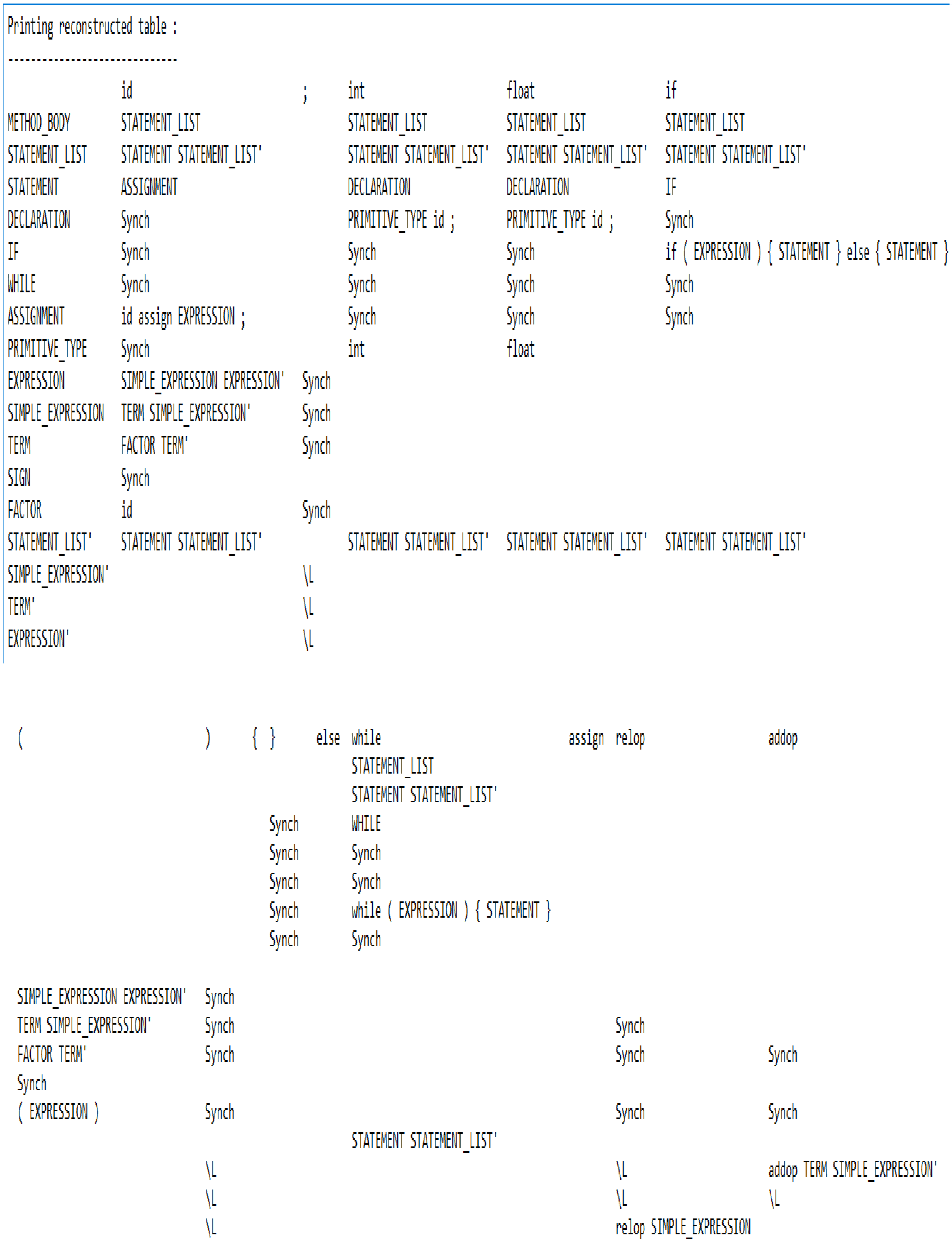


Minimized DFA table :

****

0	1	2	3	4	5	6	7	8	9	10	11	12	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	14	15	15	16	16	17	17	17	17	17	17	17	17	17	17	14	18	-1
1	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	19	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
2	13	13	20	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	21	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
3	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	22	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
4	-1	-1	-1	23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	24	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
6	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	25	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
14	-1	-1	-1	23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	17	17	17	17	17	17	17	17	17	17	-1	-1	26
18	-1	-1	-1	23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	27	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
20	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
21	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	28	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
22	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	29	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
24	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	30	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
25	13	31	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
26	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	32	32	32	32	32	32	32	32	32	32	-1	-1	-1
27	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	33	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
28	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
29	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	34	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
30	13	13	13	-1	35	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
31	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	36	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
32	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	37	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	32	32	32	32	32	32	32	32	32	32	-1	-1	-1
33	13	13	13	-1	38	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	-1	-1	-1
34	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	39	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
35	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
36	13	13	13	-1	40	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
37	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	41	41	41	41	41	41	41	41	41	41	41	-1	-1
38	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	42	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
39	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
40	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
41	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	41	41	41	41	41	41	41	41	41	41	41	-1	-1
42	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	43	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1
43	13	13	13	-1	13	13	-1	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	-1	-1	-1	-1	-1	13	13	13	13	13	13	13	13	13	13	13	-1	-1

- **Parsing Tables**
    - Reconstructed parse table (look at Reconstructed_Parse_Table.txt file)



```
Printing reconstructed table :
--------------------------------
                    id                              ;       int                         float                       if
METHOD_BODY         STATEMENT_LIST                          STATEMENT_LIST              STATEMENT_LIST              STATEMENT_LIST
STATEMENT_LIST      STATEMENT STATEMENT_LIST'               STATEMENT STATEMENT_LIST'   STATEMENT STATEMENT_LIST'   STATEMENT STATEMENT_LIST'
STATEMENT           ASSIGNMENT                              DECLARATION                 DECLARATION                 IF
DECLARATION         Synch                                   PRIMITIVE_TYPE id ;         PRIMITIVE_TYPE id ;         Synch
IF                  Synch                                   Synch                       Synch                       if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
WHILE               Synch                                   Synch                       Synch                       Synch
ASSIGNMENT          id assign EXPRESSION ;                  Synch                       Synch                       Synch
PRIMITIVE_TYPE      Synch                                   int                         float
EXPRESSION          SIMPLE_EXPRESSION EXPRESSION'   Synch
SIMPLE_EXPRESSION   TERM SIMPLE_EXPRESSION'         Synch
TERM                FACTOR TERM'                    Synch
SIGN                Synch
FACTOR              id                              Synch
STATEMENT_LIST'     STATEMENT STATEMENT_LIST'               STATEMENT STATEMENT_LIST'   STATEMENT STATEMENT_LIST'   STATEMENT STATEMENT_LIST'
SIMPLE_EXPRESSION'                                  \L
TERM'                                               \L
EXPRESSION'                                         \L


(                                ))       { }         else  while                                 assign  relop                  addop
                                                           STATEMENT_LIST
                                                           STATEMENT STATEMENT_LIST'
                                           Synch           WHILE
                                           Synch           Synch
                                           Synch           Synch
                                           Synch           while ( EXPRESSION ) { STATEMENT }
                                           Synch           Synch

SIMPLE_EXPRESSION EXPRESSION'    Synch
TERM SIMPLE_EXPRESSION'          Synch                                                                   Synch
FACTOR TERM'                     Synch                                                                   Synch                  Synch
Synch
( EXPRESSION )                   Synch                                                                   Synch                  Synch
                                                           STATEMENT STATEMENT_LIST'
                                 \L                                                                      \L                     addop TERM SIMPLE_EXPRESSION'
                                 \L                                                                      \L                     \L
                                 \L                                                                      relop SIMPLE_EXPRESSION
```

```
mulop          num                              +                              -                              $
                                                                                                              Synch
                                                                                                              Synch
                                                                                                              Synch
                                                                                                              Synch
                                                                                                              Synch
                                                                                                              Synch
                                                                                                              Synch

               SIMPLE_EXPRESSION EXPRESSION'    SIMPLE_EXPRESSION EXPRESSION'  SIMPLE_EXPRESSION EXPRESSION'
               TERM SIMPLE_EXPRESSION'          SIGN TERM SIMPLE_EXPRESSION'   SIGN TERM SIMPLE_EXPRESSION'
               FACTOR TERM'
               Synch                            +                              -
Synch          num
                                                                                                              \L

mulop FACTOR TERM'
```

- Parse Table (separated on rows) (look at Parse_Table.txt file)

```
..............................
METHOD_BODY
id :  STATEMENT_LIST
; :  Empty
int :  STATEMENT_LIST
float :  STATEMENT_LIST
if :  STATEMENT_LIST
( :  Empty
) :  Empty
{ :  Empty
} :  Empty
else :  Empty
while :  STATEMENT_LIST
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Synch
```

```
....................................
STATEMENT_LIST
id :  STATEMENT STATEMENT_LIST'
; :  Empty
int :  STATEMENT STATEMENT_LIST'
float :  STATEMENT STATEMENT_LIST'
if :  STATEMENT STATEMENT_LIST'
( :  Empty
) :  Empty
{ :  Empty
} :  Empty
else :  Empty
while :  STATEMENT STATEMENT_LIST'
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Synch
```

```
...............................
STATEMENT
id :   ASSIGNMENT
; :   Empty
int :   DECLARATION
float :   DECLARATION
if :   IF
( :   Empty
) :   Empty
{ :   Empty
} :   Synch
else :   Empty
while :   WHILE
assign :   Empty
relop :   Empty
addop :   Empty
mulop :   Empty
num :   Empty
+ :   Empty
- :   Empty
$ :   Synch
```

```
..............................
DECLARATION
id :   Synch
; :   Empty
int :   PRIMITIVE_TYPE id ;
float :   PRIMITIVE_TYPE id ;
if :   Synch
( :   Empty
) :   Empty
{ :   Empty
} :   Synch
else :   Empty
while :   Synch
assign :   Empty
relop :   Empty
addop :   Empty
mulop :   Empty
num :   Empty
+ :   Empty
- :   Empty
$ :   Synch
```

```
.............................................
IF
id :   Synch
; :   Empty
int :   Synch
float :   Synch
if :   if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
( :   Empty
) :   Empty
{ :   Empty
} :   Synch
else :   Empty
while :   Synch
assign :   Empty
relop :   Empty
addop :   Empty
mulop :   Empty
num :   Empty
+ :   Empty
- :   Empty
$ :   Synch
```

```
............................................
WHILE
id :  Synch
; :  Empty
int :  Synch
float :  Synch
if :  Synch
( :  Empty
) :  Empty
{ :  Empty
} :  Synch
else :  Empty
while :  while ( EXPRESSION ) { STATEMENT }
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Synch
```

```
..............................
ASSIGNMENT
id :  id assign EXPRESSION ;
; :  Empty
int :  Synch
float :  Synch
if :  Synch
( :  Empty
) :  Empty
{ :  Empty
} :  Synch
else :  Empty
while :  Synch
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Synch
```

```
.....................
PRIMITIVE_TYPE
id :  Synch
; :  Empty
int :  int
float :  float
if :  Empty
( :  Empty
) :  Empty
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Empty
```

```
.......................................
EXPRESSION
id :  SIMPLE_EXPRESSION EXPRESSION'
; :  Synch
int :  Empty
float :  Empty
if :  Empty
( :  SIMPLE_EXPRESSION EXPRESSION'
) :  Synch
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  SIMPLE_EXPRESSION EXPRESSION'
+ :  SIMPLE_EXPRESSION EXPRESSION'
- :  SIMPLE_EXPRESSION EXPRESSION'
$ :  Empty
```

```
.................................
SIMPLE_EXPRESSION
id :  TERM SIMPLE_EXPRESSION'
; :  Synch
int :  Empty
float :  Empty
if :  Empty
( :  TERM SIMPLE_EXPRESSION'
) :  Synch
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Synch
addop :  Empty
mulop :  Empty
num :  TERM SIMPLE_EXPRESSION'
+ :  SIGN TERM SIMPLE_EXPRESSION'
- :  SIGN TERM SIMPLE_EXPRESSION'
$ :  Empty
```

```
........................
TERM
id :  FACTOR TERM'
; :  Synch
int :  Empty
float :  Empty
if :  Empty
( :  FACTOR TERM'
) :  Synch
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Synch
addop :  Synch
mulop :  Empty
num :  FACTOR TERM'
+ :  Empty
- :  Empty
$ :  Empty
```

```
.......................
SIGN
id :  Synch
; :  Empty
int :  Empty
float :  Empty
if :  Empty
( :  Synch
) :  Empty
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Synch
+ :  +
- :  -
$ :  Empty
```

```
.......................
FACTOR
id :  id
; :  Synch
int :  Empty
float :  Empty
if :  Empty
( :  ( EXPRESSION )
) :  Synch
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  Synch
addop :  Synch
mulop :  Synch
num :  num
+ :  Empty
- :  Empty
$ :  Empty
```

```
....................................
STATEMENT_LIST'
id :  STATEMENT STATEMENT_LIST'
; :  Empty
int :  STATEMENT STATEMENT_LIST'
float :  STATEMENT STATEMENT_LIST'
if :  STATEMENT STATEMENT_LIST'
( :  Empty
) :  Empty
{ :  Empty
} :  Empty
else :  Empty
while :  STATEMENT STATEMENT_LIST'
assign :  Empty
relop :  Empty
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  \L
```

```
...................................
SIMPLE_EXPRESSION'
id :  Empty
; :  \L
int :  Empty
float :  Empty
if :  Empty
( :  Empty
) :  \L
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  \L
addop :  addop TERM SIMPLE_EXPRESSION'
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Empty
```

```
...............................................................
TERM'
id :  Empty
; :  \L
int :  Empty
float :  Empty
if :  Empty
( :  Empty
) :  \L
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  \L
addop :  \L
mulop :  mulop FACTOR TERM'
num :  Empty
+ :  Empty
- :  Empty
$ :  Empty
```

```
EXPRESSION'
id :  Empty
; :  \L
int :  Empty
float :  Empty
if :  Empty
( :  Empty
) :  \L
{ :  Empty
} :  Empty
else :  Empty
while :  Empty
assign :  Empty
relop :  relop SIMPLE_EXPRESSION
addop :  Empty
mulop :  Empty
num :  Empty
+ :  Empty
- :  Empty
$ :  Empty
```

- **Comments about tools**
  - Code Blocks: To write our C++ program on.
  - Notepad++: To read our output files from.

- **Functions Explanations**

1- Parsing Context Free Grammar:

bool is_LL_grammar(): CFG_Parser;

- check if the grammar is LL(1).
- First, substitute every non-terminal with its productions to handle non-immediate left recursion or non-immediate left factor.
- Second, check if there is immediate left recursion or left factor.
- If there is a left recursion or left factor, return false.
  Else, return true (LL(1)).

void parse_rules_string(string rules) : CFG_Parser;

- Take all the rules in the file as a string (argument).
- Split it about '#' character and put the result in a vector of string.
- for(string rule : rules_tokens) {
      add_new_rule(rule);

  }

void add_new_rule(string rule_string) : CFG_Parser;

- Take every rule in the file as string to parse it.
- Call Rule. split_rule_string(rule_string).
- If not exist in the vector of Rules, make a new rule and parse productions.
- Else parse its productions.

Rule::RuleStringStruct split_rule_string(string rule) : Rule;

- Find the assignment operator "=" index .

- Split the rule about the operator.
- Substring the lhs and rhs.
- check valid RHS, LHS for a valid grammar rules.

Parse productions about '|' to get OR of concatenations.

2- Generate LL(1) Grammar:

void eliminate_left_recursion();

- Substitute every non-terminal with its productions to handle non-immediate left recursion.
- Eliminate immediate left recursion by applying $A \rightarrow A\ \alpha \mid \beta$

  ➔ $A \rightarrow \beta\ A'$

  $A' \rightarrow \alpha\ A' \mid \varepsilon$

void eliminate_left_factor();

- Substitute every non-terminal with its productions to handle non-immediate left factor.

- Eliminate immediate left factor by applying $A \rightarrow \alpha\beta1 \mid \alpha\beta2$

  ➔

  $A \rightarrow \alpha A$

  $A' \rightarrow \beta1 \mid \beta2$

3- Creating Parse Table:

void make_parse_table();

- It build the parse table by calling get_first then get_follow for all non_terminals then it call create_parse_table.

void get_first(Symbol* s);

- It fills the first map with the first of symbol according to algorithm illustrated above.

void get_follow(Symbol* s);

- It fills the follow map with the follow of symbol according to algorithm illustrated above.

void create_parse_table();

- It fills the table map with the rows full of productions according to algorithm illustrated above.

vector<vector<string>> get_reconstructed_table();

- It converts the table from map to 2D vector array but it should only be called if Grammar wasn't ambiguous as it will only take the first production and add it to new table.

4- Matcher:

start_matcher(vector <string> tokens)

- Push '$' sign then the starting symbol on the validation stack.
- Start iterating over the stack and the input tokens.
- Try to match the left most input token with the top of stack, if there is a match then remove that input token and pop the stack.
- If there was no match then look up the parsing table for suitable production to the top of stack and the input token, if no row found then there was a terminal on the top of the validation stack that was not matched with the input so a 'missing error' is declared and the stack is popped, if no column found that means that the input token is not in the set of terminals in the grammar so a 'not accepted error' is declared and parsing is terminated.
- If an entry found then check if the production string returned is empty then a 'discard error' is declared to discard that input token and procced with the next one, if it was a sync or \L then the stack is popped and execution is continued, else pop the stack and push the production found in reversed order.

After finishing iteration check the size of both stack and input tokens, if stack is not empty then a ‘missing error’ is declared for each string on stack, if the input tokens are not empty then a ‘discard error’ is declared for every input token left.

- **Assumptions & Justification**

  1- Max levels of substitution left recursion and left factor is 1.