Phase 2 report

Islam Mohamed Mohamed	12
Bahaa khalaf	21
Abdelrahman Aly Abdallah	38
Makrm William	63

Data Structure :

- ## ReadGrammers

  It is a class used to read grammer from input file and parse it using regex to find nonterminals and their terminals.
  it consist of :
  - Vectors :
    - Vector of production to read from input file .
    - Vector of vector of production to split the input file into terminal and non terminal.
  - Maps :
    - Map of terminals: to get terminals and we write the string of each terminal and its production in the map as each nonterminal or terminal has an instance of the class production to have its attribute
    - Map of nonterminals : the same as the map of terminals but for nonterminals

```
static vector< vector< production *> >  split(const string &str,production *t,map<string, production *> &nonterminals
                                        ,map<string, production *> &terminals);
```

```
vector<production *> ReadGrammarFile(const string &grammerfile);
```

```
map<string, production *> nonterminals;
map<string, production *> terminals;
vector<production *> vec;
```

- ## production

  It is a class used to represent nonterminals and terminals and differ between them using enum productionType .
  it consist of :
  - Bool eps : if the non terminal has epsilon or not
  - String value : the value of terminal or nonterminal
  - Vectors :
    - Vector of vector to connect each nonterminal with its production in RHS after it is splitted in class **ReadGrammers.**

- Vector of appearance shows the appearance of each nonterminal in the RHS which is connected to the nonterminal of LHS.

- Maps :
  - Map to represent the first of production :
    - it maps the terminal string to the vector of vector of production (terminal or non terminal)

      E→TF | ‘c’

      T→ ‘k’

      First of (E) :

      [[“k” map to [T, F],

      ‘c’ map to [c]].

  - Map to represent the follow of production :
    - find the follow for each nonterminal .

```
class production {
public:
    string value;
    string temp;
    vector< vector< production *>  > RHS;
    vector<production*> appearance;
    bool eps;
    map<string,vector<production*>> PrFirst;
    map<string,vector<production*>> follow;
    production(string val,productionType type);
    void SetFirst(map<string,production *> nonTerminal);
    productionType type;
```

• ParserTable

It is a class used to represent the parser table and it consist of :

- Map called table to map each pair of (production and string) to vector of production e.g map[E, ’+’] = [T, F]
- Un_ordered set<string> for terminals .
- Stack that has first the “$” sign and the start nonterminal

```
    ParserTable();
    map<pair<production *, string>, vector<production *>> table;
    bool Ambiguity;
```

Algorithms :

1- To Read from input :

- Use Regex to parse the input
- **ReadGrammarFile :** read the input according to regex then call **removeSpaces** then call **findNonTerminalLHS** finally we call **split**
- **removeSpaces :** A function to remove any space in non terminal production
- **findNonTerminalLHS** : check if the nonterminal is in the nonterminal map else create new production and put it into map
- **Split :** take the nonterminals map and terminals then split it according to rules in the phase 2 pdf then call **findTerminal** and finally call **findNonTerminalRHS**
- **findTerminal** : check if the term in the terminal map else create new production and puts it into the map.
- **findNonTerminalRHS :** check if the term in the RHS of each non terminal after splitted else create new production and put it into map.then, put LHS Nonterminal in its (appearance) vector .

2- To Get first and follow

- **SetFirst :** A function takes the nonterminal vector of production and used to get the first of production and fill it into a map that we used in the production class which belongs to first using the **DFS algorithm**.
- **SetFollow :** A function takes the nonterminal vector of production and used to find the first nonterminal and put the “$” sign in it’s follow and then call calcFollow to find the follow of each non terminal.
- **calcFollow:** A function takes the nonterminal vector of production and is used to calculate the follow of nonterminal according to the three rules which were discussed in the lecture.

3- To Build Parser table :

- ### getTable :
  - takes nonterminal vector and go through the map of first (Prfirst) and make pair of (nonterminal with the terminal string of prfirst) and map it to the vector of production
    E.g pair (E , ‘id’) map to E->TF
  - Go through the follow map and put “Sync” in the nonterminal production which does not have epsilon.

- If the production nonterminal has epsilon and the entry of the table is not empty it results in ambiguity and makes ambiguity true

4- To match the input and get the output

- getOutput :

- take token from phase 1 when needed until test program end then make input =”$”
- Take the start of nonterminal production
- Make stack of production and first pushes “$” and start
- Search in the parser table with the top of stack and the input string.
- If not found in the map then report error and get next token from input and continue
- If the top stack was terminal, compare with input string if match pop it and get the next token , else report error , insert correct token and match.
- If pareser table entry is “Sync” report error and pop top of stack.

- Print all stack productions in every loop in output file
- Print the action output in every loop in the terminal output

Parser Table :

NonTerminal/Terminal	mulop	relop
METHOD_BODY	Error	Error
STATEMENT_LIST	Error	Error
STATEMENT_LISTDA	Error	Error
STATEMENT	Error	Error
DECLARATION	Error	Error
PRIMITIVE_TYPE	Error	Error
IF	Error	Error
WHILE	Error	Error
ASSIGNMENT	Error	Error
EXPRESSION	Error	Error
EXPRESSIONDA	Error	relop SIMPLE_EXPRESSION
SIMPLE_EXPRESSION	Error	Sync
SIMPLE_EXPRESSIONDA	Error	eps
TERM	Error	Sync
TERMDA	mulop FACTOR TERMDA	eps
FACTOR	Sync	Sync
SIGN	Error	Error

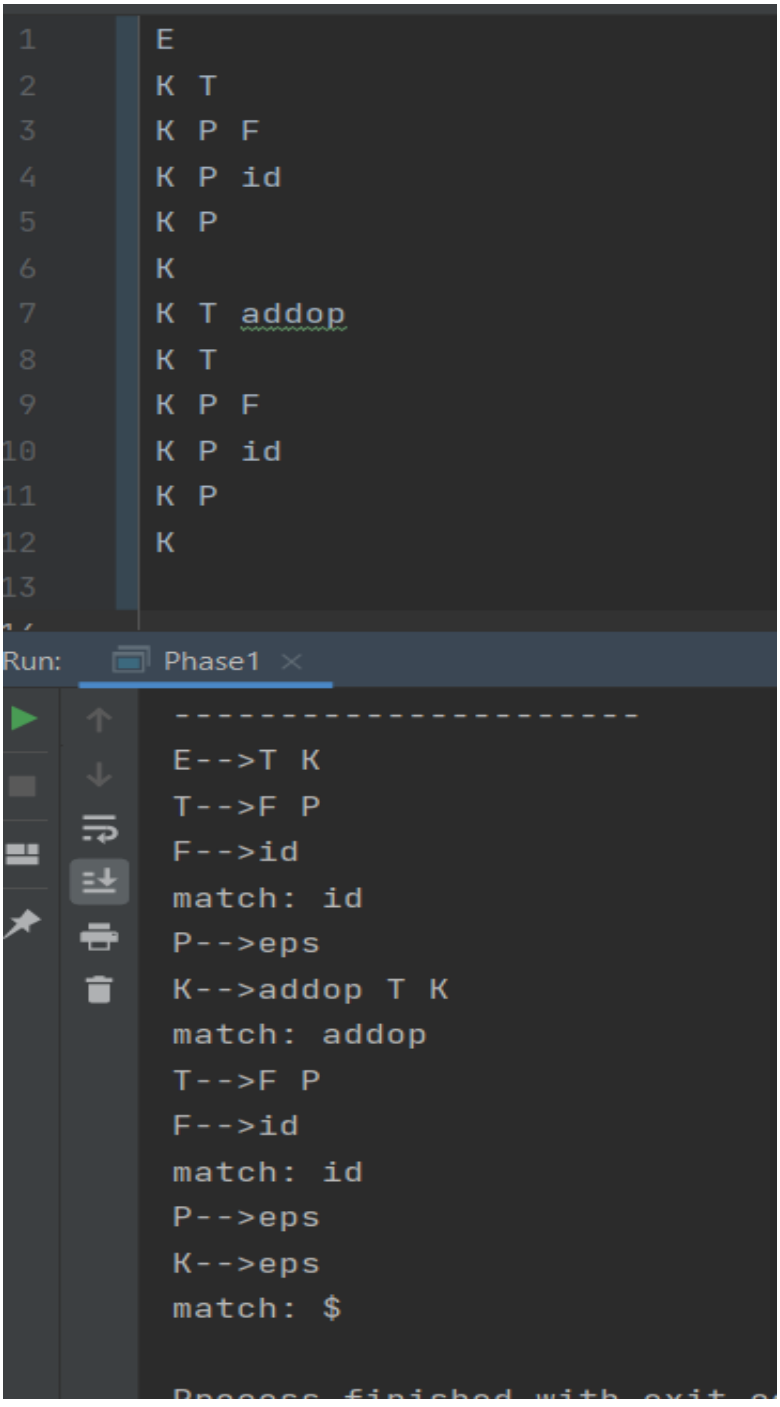
	if	int	$
	STATEMENT_LIST	STATEMENT_LIST	Sync
	STATEMENT STATEMENT_LIST"	STATEMENT STATEMENT_LIST"	Sync
	STATEMENT STATEMENT_LIST"	STATEMENT STATEMENT_LIST"	eps
	IF	DECLARATION	Sync
	Sync	PRIMITIVE_TYPE id ;	Sync
	Error	int	Error
	if (EXPRESSION) { STATEMENT } else { STATEMENT }	Sync	Sync
	Sync	Sync	Sync
	Sync	Sync	Sync
PRESSION EXPRESSION"	Error	Error	Error
	Error	Error	Error
SIMPLE_EXPRESSION"	Error	Error	Error
	Error	Error	Error
	Error	Error	Error
	Error	Error	Error
	Error	Error	Error
	Error	Error	Error

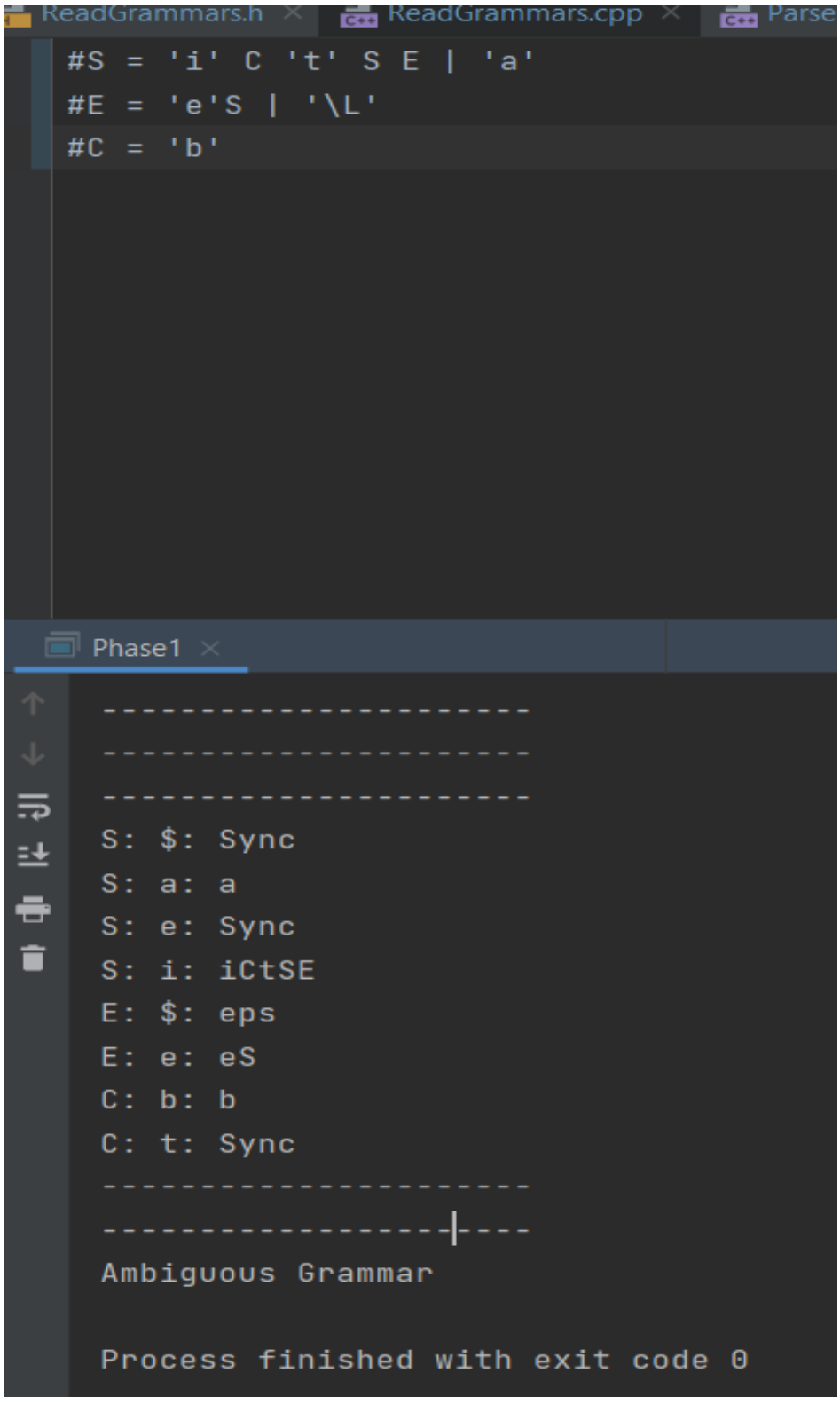
Result Of Example:

- Lab example:

```
METHOD_BODY
STATEMENT_LIST
STATEMENT_LIST" STATEMENT
STATEMENT_LIST" DECLARATION
STATEMENT_LIST" ; id PRIMITIVE_TYPE
STATEMENT_LIST" ; id int
STATEMENT_LIST" ; id
STATEMENT_LIST" ;
STATEMENT_LIST"
STATEMENT_LIST" STATEMENT
STATEMENT_LIST" ASSIGNMENT
STATEMENT_LIST" ; EXPRESSION assign id
STATEMENT_LIST" ; EXPRESSION assign
STATEMENT_LIST" ; EXPRESSION
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION" TERM
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION" TERM" FACTOR
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION" TERM" num
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION" TERM"
STATEMENT_LIST" ; EXPRESSION" SIMPLE_EXPRESSION"
STATEMENT_LIST" ; EXPRESSION"
STATEMENT_LIST" ;
STATEMENT_LIST"
STATEMENT_LIST" STATEMENT
STATEMENT_LIST" IF
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION ( if
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION (
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION" TERM
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION" TERM" FACTOR
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION" TERM" id
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION" TERM"
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION" SIMPLE_EXPRESSION"
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) EXPRESSION"
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION relop
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION" TERM
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION" TERM" FACTOR
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION" TERM" num
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION" TERM"
STATEMENT_LIST" } STATEMENT { else } STATEMENT { ) SIMPLE_EXPRESSION"
STATEMENT_LIST" } STATEMENT { else } STATEMENT { )
STATEMENT_LIST" } STATEMENT { else } STATEMENT {
STATEMENT_LIST" } STATEMENT { else } ASSIGNMENT
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION assign id
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION assign
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION" TERM
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION" TERM" FACTOR
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION" TERM" num
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION" TERM"
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION" SIMPLE_EXPRESSION"
STATEMENT_LIST" } STATEMENT { else } ; EXPRESSION"
STATEMENT_LIST" } STATEMENT { else } ;
STATEMENT_LIST" } STATEMENT { else }
STATEMENT_LIST" } STATEMENT { else
STATEMENT_LIST" } STATEMENT {
STATEMENT_LIST" } STATEMENT
STATEMENT_LIST" }
STATEMENT_LIST"
```

- Lecture examples:




ReadGrammars.h
ReadGrammars.cpp
Parse
#S = 'i' C 't' S E | 'a'
#E = 'eS | '\L'
#C = 'b'
Phase1

S: $: Sync
S: a: a
S: e: Sync
S: i: iCtSE
E: $: eps
E: e: eS
C: b: b
C: t: Sync

Ambiguous Grammar
Process finished with exit code 0

Assumption :

- The input grammar does not contain left recursion or left factoring.
- We replace ‘ in the end of the new non-terminal with “ for regex parsing .