Phase2 Report

Mahmoud Abdel Latif	#64
Karim Nasr	#48
Shady Abdel Aziz	#27
Hassan Khalil	#22

Data Structures:

Non terminal:

the structure of non-terminal term is called NonTerminal:

```
struct NonTerminal{
        string symbol
        vector<vector<int>> rule
        vector<vector<int*>> rule_ptr
        vector< set<int> > mappedFirst
        set<int> first
        set<int> follow
        vector<int> adjList
    }
```

It's used to represent rules of the grammar in numbers.

Parsing Grammer:

vector<NonTerminal> : this contains the grammar rules in numbers

Eliminating left recursion:

vector<vector<int*>>: This contain a pointer to each symbol in map to be updated automatically when updating map

Eliminating left factoring:

map<int, vector<vector<int>>> : To partition every non terminal with respect to first terminal

Calculating First & Follow:

First:

Set data structure to collect the first for each nonterminal and prevent duplicates as terminals accumulate while traversing new productions for the same non-terminal.

Follow:

List of List is used many times for adjacency lists during different phases of the algorithm, like when traversing the formed graph representing dependencies among non-terminals or when applying topological sort on this graph.

Set data structure is used multiple times while computing follow, to prevent occurrence of duplicity.

Building Parsing table:

Parsing table itself is represented as a two-dimensional array of list. Each entry in the array represents a transition of a non-terminal given some terminal, this entry yields a list which is the list of symbols that should be generated.

Panic-Mode Error Recovery:

stack<int> : To hold the number of rules.
vector<string> : To hold words of current line which will be printed in the file.

Algorithms and Techniques:

Parsing Grammar:

```
while(there is a line in the file){
    temp = input_line
    if(input_line starts with #){
        if(temp is not empty){
            begin to parse and fill map
        }
    }else{
        temp += input_line
    }
}
if(temp is not empty){
    parse and fill map
}
```

Eliminating left recursion:

```
- Arrange non-terminals in some order:  $A_1 \dots A_n$
- for  i  from  1  to  n  do  {
     - for  j from 1 to i-1 do  {
          replace each production
               $A_i \to A_j \gamma$
                    by
               $A_i \to \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$
               where $A_j \to \alpha_1 \mid \dots \mid \alpha_k$
       }
     - eliminate immediate left-recursions among $A_i$ productions
}
```

Eliminating left factoring:

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

  $$A \to \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

  convert it into

  $$A \to \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m$$
  $$A' \to \beta_1 \mid \dots \mid \beta_n$$

Calculating First & Follow:

First:
Dynamic Programming: Algorithm is loop for each production of a non-terminal, if it starts with a terminal, this production only adds this terminal to the set of first of that non-terminal. And if it is a non-terminal, it tests if its firsts are already calculated, if so it adds them to its own list of firsts directly, if not it calls itself recursively for our new non-terminal. If all symbol of production are non-terminals and produce the empty string, then our non-terminal adds the empty string to its set of firsts.

Follow:
Each nonterminal is represented by a node in a directed graph, a node has an edge to another if it needs its follow to computer its own. Strongly connected components in the graph represent a group of non-terminals having same follow set, so they are represented as single nodes in another graph. Graph is then sorted and follow sets are calculated for each node in the order of dependencies.

Building Parsing table:

Each entry in the table represents a production or a transition on a given terminal. The algorithm loops for each non-terminal, on each terminal in its first-set, the production from which it is found is put into this entry of table. If this non-terminal has the empty string in its first-set, the production producing this empty string is put in the table for the non-terminal for each terminal in its follow-set, if not for each terminal in the follow-set and in the first-set of the non-terminal, this entry is marked as a synchronizing token.

Panic-Mode Error Recovery:

- For each nonterminal A, mark the entries M[A,a] as ***synch*** if a is in the follow set of A. So, for an empty entry, the input symbol is discarded. This should continue until either:

1) an entry with a production is encountered. In the case, parsing is continued as usual.
2) an entry marked as ***synch*** is encountered. In this case, the parser will pop that non-terminal A from the stack. The parsing continues from that state.

- To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Parsing Table:

It’s printed in file called table in project folder.

Assumptions:

The input source file must not contain spaces, tabs or empty lines in the end of file.