Karim Mohamed Ali (34) *Mazen Ahmed Khater (36)*
Michael Ramy Ramsis (37) *Youssef Abdallah Youssef (65)*

Programming Languages and Compilers
Assignment 1
Phase2: Parser Generator

Problem Statement:

- Your task in this phase of the assignment is to design and implement an LL (1) parser generator tool.
- The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar.
- The table is to be used to drive a predictive top-down parser. If the input grammar is not LL (1), an appropriate error message should be produced.
- The generated parser is required to produce some representation of the leftmost derivation for a correct input. If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing.
- The parser generator is required to be tested using the given context free grammar of a small subset of Java. Of course, you have to modify the grammar to allow predictive parsing.
- Combine the lexical analyzer generated in phase1 and parser such that the lexical analyzer is to be called by the parser to find the next token. Use the simple program given in phase 1 to test the combined lexical analyzer and parser.
- *BONUS Task: Automatically eliminating grammar left recursion and performing left factoring before generating the parser will be considered a bonus work.*

Modules of the Parser Generator:

1. Scanner Module: (Read Input File)

ReadFile: A class that read input file and add productions in data structure that used to compute first and follow sets, by doing the following steps:

1. Read the input file then remove all spaces between strings and put strings of each production in vector.
2. ***Eliminate Left Recursion of the grammar.***
3. ***Eliminate Left Factoring of the grammar.***
4. Add each rule in production to the data structure used to compute first and follow sets.

Main Functions:

- *ExtractStrings():* return vector contains strings of the production without any spaces.
- *AddProductions():* for each rule for each production → add in the data structure which is used for the computation of first and follow sets of the LHS "Non-terminal of production" and vector contains the Terminals and Nonterminals of the rule.
- For Eliminate left factoring we split each production to rules by split around "|" then get common strings between rules and at the end add new productions for common strings which were found.
- *SplitProduction()*: return all rules for specific production by splitting around "|".
- *GetCommon():* return the common string if found between two rules.
- *InsertNode()*: insert new node in struct "FactorNode".
- *AddLeftFactoring()*: add new production for each common string.
- *FactorNode:* A struct used for elimination left factoring by carrying the size of common string ,the common string itself and which rules share the same common string.

2. First and Follow Computation Module:

1. Structures that hold the terminals and non terminals:
   - Base class Token → private string type
   - Terminal extends Token
   - NonTerminal extends Token

   - Class that holds the Grammar's productions and operations performed on it
     Grammar private:
   - multimap<shared_ptr<Token>, vector<shared_ptr<Token>>> productions; // grammar productions
   - map<shared_ptr<Token>, set<string>> follow; // follow of each nonterminal
   - map<shared_ptr<Token>, set<string>> first; // first of each token
   - set<shared_ptr<Token>> epsilonTokens; // terminals that have a production rule going to epsilon
   - shared_ptr<NonTerminal> startingSymbol; // starting symbols of the grammar.
   - bool unify(shared_ptr<Token>, shared_ptr<Token>); // unifying the first of two Tokens

- bool updateFollow(shared_ptr<Token> token, set<string> followSoFar); // updating the follow by a token

public:

- void addProduction(shared_ptr<NonTerminal> &nonTerminal, vector<shared_ptr<Token>> &rightHandSide) {
    productions.insert({nonTerminal, rightHandSide});
  }

  // adding a production to the grammar.
- void setStartingSymbol(shared_ptr<NonTerminal> _startingSymbol) {
    this->startingSymbol = _startingSymbol;
  }

  // sets the starting symbol of the grammar.
- shared_ptr<NonTerminal> getStartingSymbol(){
    return this->startingSymbol;
  }

  // getter for the starting symbol
- map<shared_ptr<Token>, set<string>> getFirst() {
    return this->first;
  }

  // getter for the first of the grammar
- map<shared_ptr<Token>, set<string>> getFollow() {
    return this->follow;
  }

  // method that computes the follow of the grammar.
- void computeFirst();

  // method that computes the follow
- void computeFollow();

  // getter for the productions of the grammar.
- multimap<shared_ptr<Token>, vector<shared_ptr<Token>>> getProductions();

In ReadFile class:

Two private functions: (mentioned before in the Scanner Module)

- void eliminateLeftRecursion(); // eliminates left recursion in the grammar, it calls eliminateImmediateLeftRecursion()
- void eliminateImmediateLeftRecursion(unsigned int); // eliminates immediate left recursion in the grammar.

3. Parsing Table Module:

This module uses the computation results of the First and Follow sets to generate the corresponding parsing table.

Data Structures needed:

- Vector of Terminals
- Vector of NonTerminals
- Multimap of all production rules
- First and Follow sets computed previously

Main Functions:

- Fill Parsing Table: A row of productions must be inserted for each NonTerminal in the grammar (one production corresponding to each Terminal).
  There are **4** different cases to fill a cell in the table.
  1. Terminal is "$":
     Check the follow set of the corresponding NonTerminal for the existence of Epsilon. If it was found, then put the production that leads to Epsilon using a private helper function special_handle() (direct production to Epsilon or indirect by other productions). Otherwise, put "Synch".
  2. Terminal is Found in the First set:
     Add the corresponding production rule which leads to this terminal using a helper private function add_terminal().
  3. Terminal is Found in the Follow set:
     Check for the existence of Epsilon in the corresponding First Set. If it was found, put the Epsilon production. Otherwise, put "Synch".
  4. Error Case:
     In case that all the previous cases were not satisfied, put error in this cell.

4. Derivation and Error handling Module:

The Derivator class takes the input and makes left most left to right derivation.

- ***Derivation***:

It holds a stack containing terminals and non terminals through every step of the derivation *(initially it contains the dollar sign "$" and the starting NonTerminal)* and according to the top of the stack and the current input token it uses the parsing table to continue the derivation such that:

- If the top of stack is a terminal token it matches it with the current input.
- If the top of stack is a non terminal it goes to the corresponding row in the parsing table and the corresponding column according to the current table and then it pops the current non terminal from the stack and pushes the production found from the table in the reverse order.
- If the production found in the table is epsilon then we pop the non terminal from the stack without pushing anything in it.
- If the top of stack is '$' (which is the first element inserted in the stack) and the current input is also '$' (which is the last token in the input) it matches them and then parsing has been terminated successfully.

- ***Error Handling:***

  - If the top of the stack is a terminal token but it doesn't match with the current input then we have an extra terminal in the stack and we pop it.
  - If the top of the stack is '$' and the input hasn't ended then parsing terminates without completing because the stack is empty and we have an error.
  - If the corresponding entry in the table is an empty cell then the error is that we have an excess token in the input and we get the next token and continue searching in the table according to the stack and the input until we reach a production or until reaching a synch entry then we pop the non terminal from the stack and continue derivation.

Assumptions:

1. Epsilon is represented internally by a zero string → “0”
2. The input program is received from the lexical analyzer as a vector of tokens to be easily treated during the derivation.

Combining Phase:

We combined the results of this phase with the lexical analyzer phase such that the input to the program is:

1. Lexical Rules
2. Grammar Rules
3. Input program

The final result will be the left most derivation of the input program.

Sample run: (for the given testing inputs in the 2 phases)

```
File  Edit  View  Search  Terminal  Help
Top of Stack: METHOD_BODY   Input: int
METHOD_BODY -> STATEMENT_LIST
Top of Stack: STATEMENT_LIST   Input: int
STATEMENT_LIST -> STATEMENT STATEMENT_LISTdash
Top of Stack: STATEMENT   Input: int
STATEMENT -> DECLARATION
Top of Stack: DECLARATION   Input: int
DECLARATION -> PRIMITIVE_TYPE id ;
Top of Stack: PRIMITIVE_TYPE   Input: int
PRIMITIVE_TYPE -> int
Top of Stack: int   Input: int
 matched int
Top of Stack: id   Input: id
 matched id
Top of Stack: ;   Input: ;
 matched ;
Top of Stack: STATEMENT_LISTdash   Input: id
STATEMENT_LISTdash -> STATEMENT STATEMENT_LISTdash
Top of Stack: STATEMENT   Input: id
STATEMENT -> ASSIGNMENT
Top of Stack: ASSIGNMENT   Input: id
ASSIGNMENT -> id assign EXPRESSION ;
Top of Stack: id   Input: id
 matched id
Top of Stack: assign   Input: assign
 matched assign
Top of Stack: EXPRESSION   Input: num
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION1
Top of Stack: SIMPLE_EXPRESSION   Input: num
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSIONdash
Top of Stack: TERM   Input: num
TERM -> FACTOR TERMdash
Top of Stack: FACTOR   Input: num
FACTOR -> num
Top of Stack: num   Input: num
 matched num
Top of Stack: TERMdash   Input: ;
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: ;
SIMPLE_EXPRESSIONdash -> epsilon
Top of Stack: EXPRESSION1   Input: ;
EXPRESSION1 -> epsilon
Top of Stack: ;   Input: ;
 matched ;
Top of Stack: STATEMENT_LISTdash   Input: if
STATEMENT_LISTdash -> STATEMENT STATEMENT_LISTdash
Top of Stack: STATEMENT   Input: if
STATEMENT -> IF
Top of Stack: IF   Input: if
IF -> if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
Top of Stack: if   Input: if
 matched if
Top of Stack: (   Input: (
 matched (
Top of Stack: EXPRESSION   Input: id
```

```
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION1
Top of Stack: SIMPLE_EXPRESSION   Input: id
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSIONdash
Top of Stack: TERM   Input: id
TERM -> FACTOR TERMdash
Top of Stack: FACTOR   Input: id
FACTOR -> id
Top of Stack: id   Input: id
 matched id
Top of Stack: TERMdash   Input: relop
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: relop
SIMPLE_EXPRESSIONdash -> epsilon
Top of Stack: EXPRESSION1   Input: relop
EXPRESSION1 -> relop SIMPLE_EXPRESSION
Top of Stack: relop   Input: relop
 matched relop
Top of Stack: SIMPLE_EXPRESSION   Input: num
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSIONdash
Top of Stack: TERM   Input: num
TERM -> FACTOR TERMdash
Top of Stack: FACTOR   Input: num
FACTOR -> num
Top of Stack: num   Input: num
 matched num
Top of Stack: TERMdash   Input: )
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: )
SIMPLE_EXPRESSIONdash -> epsilon
Top of Stack: )   Input: )
 matched )
Top of Stack: {   Input: {
 matched {
Top of Stack: STATEMENT   Input: id
STATEMENT -> ASSIGNMENT
Top of Stack: ASSIGNMENT   Input: id
ASSIGNMENT -> id assign EXPRESSION ;
Top of Stack: id   Input: id
 matched id
Top of Stack: assign   Input: assign
 matched assign
Top of Stack: EXPRESSION   Input: num
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION1
Top of Stack: SIMPLE_EXPRESSION   Input: num
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSIONdash
Top of Stack: TERM   Input: num
TERM -> FACTOR TERMdash
Top of Stack: FACTOR   Input: num
FACTOR -> num
Top of Stack: num   Input: num
 matched num
Top of Stack: TERMdash   Input: ;
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: ;
SIMPLE_EXPRESSIONdash -> epsilon
```

```
File  Edit  View  Search  Terminal  Help
FACTOR -> num
Top of Stack: num   Input: num
 matched num
Top of Stack: TERMdash   Input: )
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: )
SIMPLE_EXPRESSIONdash -> epsilon
Top of Stack: )   Input: )
 matched )
Top of Stack: {   Input: {
 matched {
Top of Stack: STATEMENT   Input: id
STATEMENT -> ASSIGNMENT
Top of Stack: ASSIGNMENT   Input: id
ASSIGNMENT -> id assign EXPRESSION ;
Top of Stack: id   Input: id
 matched id
Top of Stack: assign   Input: assign
 matched assign
Top of Stack: EXPRESSION   Input: num
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION1
Top of Stack: SIMPLE_EXPRESSION   Input: num
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSIONdash
Top of Stack: TERM   Input: num
TERM -> FACTOR TERMdash
Top of Stack: FACTOR   Input: num
FACTOR -> num
Top of Stack: num   Input: num
 matched num
Top of Stack: TERMdash   Input: ;
TERMdash -> epsilon
Top of Stack: SIMPLE_EXPRESSIONdash   Input: ;
SIMPLE_EXPRESSIONdash -> epsilon
Top of Stack: EXPRESSION1   Input: ;
EXPRESSION1 -> epsilon
Top of Stack: ;   Input: ;
 matched ;
Top of Stack: }   Input: }
 matched }
Top of Stack: else   Input: $
Error : matching terminals failed! Extra else in stack
Top of Stack: {   Input: $
Error : matching terminals failed! Extra { in stack
Top of Stack: STATEMENT   Input: $
STATEMENT -> synch
Top of Stack: }   Input: $
Error : matching terminals failed! Extra } in stack
Top of Stack: STATEMENT_LISTdash   Input: $
STATEMENT_LISTdash -> epsilon
Top of Stack: $   Input: $
Successfully Done
```