COMPILERS
PARSER GENERATOR

22/4/2020

DATA STRUCTURES

1. Syntactic Term

Syntactic term is an inheritance class used to represent the non-terminal terms and it consists of:

- Vector of Production Rules for this term.
- Unordered set of strings for the first
- Unordered set of strings for the follow

2. Production term

Production term is a class used to represent terminal, non-terminal terms and it consists of:

- String to save the name of this term.
- Production term type either terminal or non-terminal.

3. Production Rule

Production rule is a class used to represent rules and it consists of:

- Vector of pointers on Production terms to save all terms in RHS of the rule.
- Pointer on Syntactic term which is in the LHS of the rule.

4. Parsing Table

It is a class to calculate first, follow and parsing table:

- First: A map its key is pointer on the Syntactic term and its value is unordered set of strings which contains the terminals of the first set.
- First: A map its key is pointer on the Syntactic term and its value is unordered set of strings which contains the terminals of the follow set.
- A map its key is pointer on the Syntactic term and its value is another map, this map of key string “the terminal” and the value is a production rule.

5. We defined the following helping enums:

- ProductionTermType: ***Terminal***, ***NonTerminal***.

6. We used the following data structures:

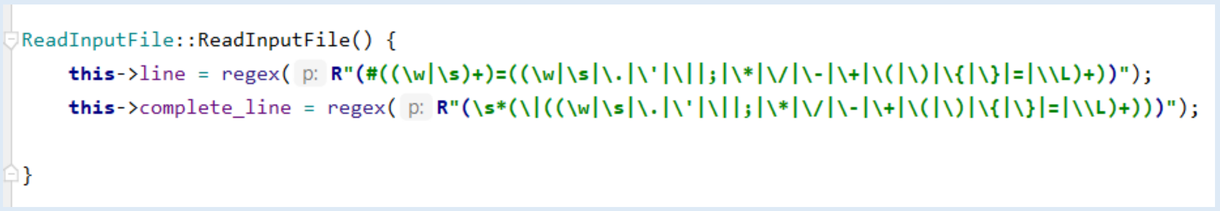
1. **Stack:** Used in parsing input in the syntactical analyzer to match the input token with the grammar table.
2. **Map:** As the data structure to hold the grammar table.
3. **Unordered sets**: were used to hold the strings of the first or follow sets of expressions or elements since order was not significant.

ALGORITHMS

1. REGEX for grammar parsing

We used regex to check the validity of the grammar file and extract the values of *Non-Terminals and Terminals to build our parser.*

*So, there some main procedures to do this parsing them using **ReadInputFile** class.*



```
ReadInputFile::ReadInputFile() {
    this->line = regex( p: R"(#((\w|\s)+)=((\w|\s|\.|\'|\||;|\*|\/|\-|\+|\(|\)|\{|\}|=|\\L)+))");
    this->complete_line = regex( p: R"(\s*(\|((\w|\s|\.|\'|\||;|\*|\/|\-|\+|\(|\)|\{|\}|=|\\L)+)))");

}
```

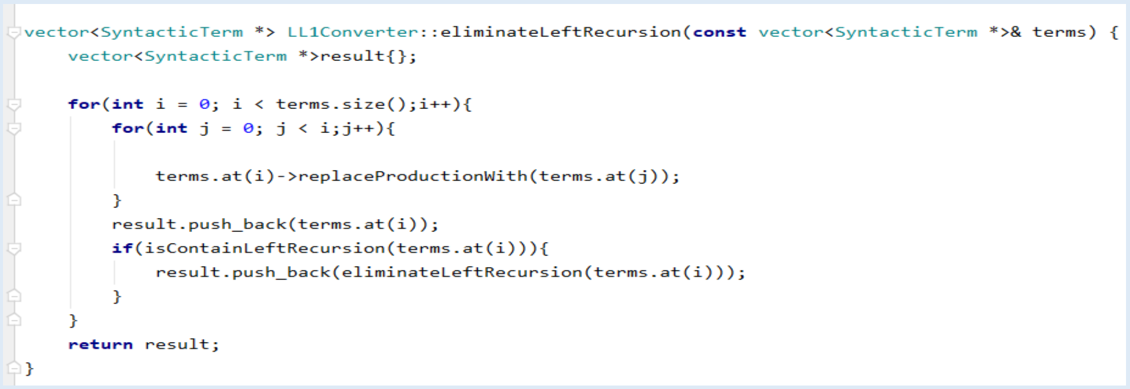
2. Eliminate left recursion and left factoring of CFG (BOUNS)

To convert context free grammar rules to LL1; We used the algorithm discussed in lecture.

So, there some main procedures to do this construction using ***LL1Converter*** *class.*

1. Elimination of Left recursion:

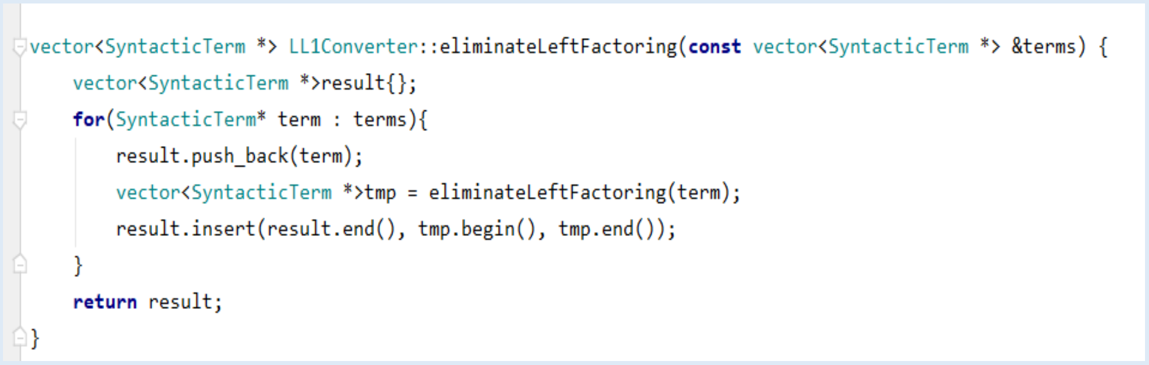
we check first for the presence of direct recursion in the grammar rule, If yes we eliminate it be removing first element in the expressions which cause this left recursion and form a new rule and put this expressions in, all expressions we add the new rule at the end of each expression, If not direct Then we check for indirect left recursion and put all expression that lead to indirect left recursion in a list, then we substitute them to form a direct left recursion rule and solve it as we mentioned before.



```
vector<SyntacticTerm *> LL1Converter::eliminateLeftRecursion(const vector<SyntacticTerm *>& terms) {
    vector<SyntacticTerm *>result{};

    for(int i = 0; i < terms.size();i++){
        for(int j = 0; j < i;j++){

            terms.at(i)->replaceProductionWith(terms.at(j));
        }
        result.push_back(terms.at(i));
        if(isContainLeftRecursion(terms.at(i))){
            result.push_back(eliminateLeftRecursion(terms.at(i)));
        }
    }
    return result;
}
```

2. Left Factoring:

We first check for a direct left factoring and start to have a factor of elements then we put them in an expression and add a new grammar element to this expression, then we take the expression involved in this factorization and put them in this new grammar element, we also check the nested left factoring then by substituting the same as we have done in left recursion we get a direct left factoring then apply the same technique of direct left factoring.



```
vector<SyntacticTerm *> LL1Converter::eliminateLeftFactoring(const vector<SyntacticTerm *> &terms) {
    vector<SyntacticTerm *>result{};
    for(SyntacticTerm* term : terms){
        result.push_back(term);
        vector<SyntacticTerm *>tmp = eliminateLeftFactoring(term);
        result.insert(result.end(), tmp.begin(), tmp.end());
    }
    return result;
}
```

3. Compute First and Follow

To Calculate first of a non-terminal term we do these steps:

- *Loop on all productions for this term*
- *Cheek if this production rule is epsilon or not i.e: term → EPS if yes add eps to answer and continue to the next production rule.*
- *Get the first term in the left side there is to cases*
  - term is terminal: then add it to the set of the result
  - term is non-terminal: add the first of this term to the result set, if it still didn't computed then we call recursively the function.
- *If there is an epsilon in the first of the first non-terminal, then we check the term after it and so on.*
- *Each string we get and put in the result set, it's added instantaneously to the table with the production rule.*

```
unordered_set<string> ParsingTable::getFirst(SyntacticTerm* non_terminal) {
    unordered_set<string> res;
    if (first.find(non_terminal) == first.end()) {
        vector<ProductionRule *> x = non_terminal->getProductions();
        /// loop on each production rule and get from it first
        for (auto productionRule:x) {
            unordered_set<string> onePR;
            if (productionRule->isEpsilon()) {
                res.insert("EPS");
                continue;
            }
            int index = 0;
            auto *firstTerm = (SyntacticTerm *) productionRule->getTerms().at(index);
            /// if symbol is terminal then add to the list

            if (firstTerm->getType() == Terminal) {
                if (table.find(non_terminal) == table.end()){
                    map<std::string, struct ProductionRule> newchar;
                    newchar.insert(pair<std::string, struct ProductionRule>(firstTerm->getName(),
*productionRule));
                    table.insert(pair<SyntacticTerm *, map<std::string, struct
ProductionRule>>(non_terminal,newchar));
                } else {
                    if (table.find(non_terminal)->second.find(firstTerm->getName()) != table.find(non_terminal)-
>second.end()){
                        table.find(non_terminal)->second.find(firstTerm->getName())->second = *productionRule;
                    } else {
                        table.find(non_terminal)->second.insert(
                                pair<std::string, struct ProductionRule>(firstTerm->getName(), *productionRule));
                    }
                }
                res.insert(firstTerm->getName());
                ///else  if symbol is non-terminal then compute its first then add it
            } else {
                unordered_set<string> temp;
                /// if the first is already computed add it
                if (first.find(firstTerm) != first.end()) {
                    temp = first.at(firstTerm);
                    ///else  if it's not computed before then recursive call the function
                } else {
                    temp = getFirst(firstTerm);
                }
                /// handling special case for having epsilon at first of the first non-terminal terms.
                while (temp.find("EPS") != temp.end()) {
                    temp.erase("EPS");
                    res.insert(temp.begin(), temp.end());
                    onePR.insert(temp.begin(), temp.end());
                    index++;
                    if (index < productionRule->getTerms().size()) {
                        auto *nextTerm = (SyntacticTerm *) productionRule->getTerms().at(index);
                        temp = getFirst(nextTerm);
                    } else {
                        temp.insert("EPS");
                        break;
                    }
                }
                res.insert(temp.begin(), temp.end());
                onePR.insert(temp.begin(), temp.end());
            }
            for (const auto& c:onePR){
                if (table.find(non_terminal) != table.end()){
                    if (table.find(non_terminal)->second.find(c) != table.find(non_terminal)->second.end()){
                        table.find(non_terminal)->second.find(c)->second = *productionRule;
                    } else {
                        table.find(non_terminal)->second.insert(pair <std::string, struct ProductionRule> (c,
*productionRule));
                    }
                } else {
                    map<std::string, struct ProductionRule> newchar;
                    newchar.insert(pair<std::string, struct ProductionRule>(c, *productionRule));
                    table.insert(pair<SyntacticTerm *, map<std::string, struct
ProductionRule>>(non_terminal,newchar));
                }
            }
        }
        first.insert(pair<SyntacticTerm *, unordered_set<string>>(non_terminal, res));
        non_terminal->setFirst(res);
        return res;
    } else {
        non_terminal->setFirst(first.at(non_terminal));
        return first.at(non_terminal);
    }
}
```

To Calculate follow of all non-terminal term we do these steps:

- *Loop on all non-terminal term we have.*

- *Loop on all production rules for each term.*
- *Loop on all terms in each production rule twice*
  -forward: to know each term id followed by what, saving terminals in a set and the non-terminals in another set till the time it will be calculated
  - backward: to get cases for the epsilon in the last element.
- *At the end all non-terminal sets are cleared and but instead it the follow of this terms.*

```
void ParsingTable::setFollowTable(vector<SyntacticTerm*> non_terminal) {
    map<SyntacticTerm*, unordered_set<SyntacticTerm*>> nonterm_follow;
    /// setting the sets foe follow results
    for (auto item:non_terminal) {
        unordered_set<string> newSet;
        unordered_set<SyntacticTerm*> newSetT;
        if (non_terminal.at(0) == item){ /// If S is the start symbol ⎝ $ is in FOLLOW(S)
            newSet.insert("$");
        }
        follow.insert(pair<SyntacticTerm*, unordered_set<string>>(item,newSet));
        nonterm_follow.insert(pair<SyntacticTerm*, unordered_set<SyntacticTerm*>>(item,newSetT));
    }
    /// loop on all non terminals
    for (auto item:non_terminal){
        /// loop on all productions of each non-terminal item
        for (auto p:item->getProductions()) {
            vector<ProductionTerm *> terms =p->getTerms();
            /// loop on terms of each production rule twice: forward and backward
            for (int i = 1;i <terms.size();i++){

                if (terms.at(i-1)->getType() == NonTerminal) {
                    if (terms.at(i)->getType() == Terminal) {
                        if (follow.find((SyntacticTerm *) terms.at(i - 1)) != follow.end()) {
                            follow.find((SyntacticTerm *) terms.at(i - 1))->second.insert(terms.at(i)->getName());
                        } else {
                            unordered_set<string> newSet;
                            newSet.insert(terms.at(i)->getName());
                            follow.insert(pair<SyntacticTerm*, unordered_set<string>>((SyntacticTerm *) terms.at(i -
1),newSet));
                        }
                    } else { ///  if  A-> aBb  is a production rule ⎝ everything in FIRST(b) is FOLLOW(B) except
EPS
                        unordered_set<string> temp ;
                        if (first.find((SyntacticTerm *)terms.at(i)) != first.end()) {
                            temp = first.find((SyntacticTerm *)terms.at(i))->second;
                        } else {
                            temp = getFirst((SyntacticTerm *)terms.at(i));
                        }
                        if (temp.find("EPS") != temp.end()) {
                            temp.erase(temp.find("EPS"));
                        }
                        if (follow.find((SyntacticTerm *) terms.at(i - 1)) != follow.end()) {
                            follow.find((SyntacticTerm *) terms.at(i - 1))->second.insert(temp.begin(), temp.end());
                        } else {
                            unordered_set<string> newSet;
                            newSet.insert(temp.begin(), temp.end());
                            follow.insert(pair<SyntacticTerm*, unordered_set<string>>((SyntacticTerm *) terms.at(i -
1),newSet));
                        }
                    }
                }
            }
            /// loop backward to get cases for the epsilon in last elements
            if (terms.at(terms.size()-1)->getType() == NonTerminal && terms.at(terms.size()-1) != item){
                nonterm_follow.find((SyntacticTerm *) terms.at(terms.size()-1))->second.insert((SyntacticTerm *)
item);
            }
            int n = terms.size();
            n--;
            for (int i = n;i > 0;i--){
                if (terms.at(i)->getType() == NonTerminal && terms.at(i-1)->getType() == NonTerminal){
                    if (((SyntacticTerm *) terms.at(i))->isDerivingToEpsilon() && ( terms.at(i - 1) != item)){
                        nonterm_follow.find((SyntacticTerm *) terms.at(i - 1))->second.insert(item);
                    } else {
                        break;
                    }
                } else {
                    break;
                }
            }
        }
    }
    /// to finalize the follow results and remove non-terminal from it
    finalizingfollow(nonterm_follow);
}

void ParsingTable::finalizingfollow(map<SyntacticTerm *, unordered_set<SyntacticTerm *>>
nonterm_follow) {
    int i = 0;
    int times = 0;
    do {
        times ++;
        i = 0;
        for (auto item:nonterm_follow) {
            if (!item.second.empty()) {
                i++;
                /// eliminate non-terminal from other elements
                for (auto item2:item.second) {
                    if (nonterm_follow.find(item2)->second.empty() && item.second.find(item2) !=
item.second.end()){
                        follow.find(item.first)->second.insert(
                                follow.find(item2)->second.begin(), follow.find(item2)->second.end());
                        item.second.erase(item.second.find(item2));
                        nonterm_follow.find(item.first)->second = item.second;
                    }
                }
            }
        }
    } while (i != 0 && times <= nonterm_follow.size());
}
```

```
void ParsingTable::finalizingfollow(map<SyntacticTerm *, unordered_set<SyntacticTerm *>>
nonterm_follow) {
    int i = 0;
    int times = 0;
    do {
        times ++;
        i = 0;
        for (auto item:nonterm_follow) {
            if (!item.second.empty()) {
                i++;
                /// eliminate non-terminal from other elements
                for (auto item2:item.second) {
                    if (nonterm_follow.find(item2)->second.empty() && item.second.find(item2) !=
item.second.end()){
                        follow.find(item.first)->second.insert(
                                follow.find(item2)->second.begin(), follow.find(item2)->second.end());
                        item.second.erase(item.second.find(item2));
                        nonterm_follow.find(item.first)->second = item.second;
                    }
                }
            }
        }
    } while (i != 0 && times <= nonterm_follow.size());
}
```

4. Constructing Parsing Table

Steps:

- *Loop on all non-terminals to get calculate all first values and add it to the parsing table.*
- *Calculate follow for all non-terminals.*
- *Loop on follows we have*
  -If the term is driving to Epsilon then add the production term → EPS to cells from the set of follow in the table.
  --- if this cell is already have a production rule set ambiguity to true.
  -If it's not driving to Epsilon then add synch to the cells.
- *Then return table.*

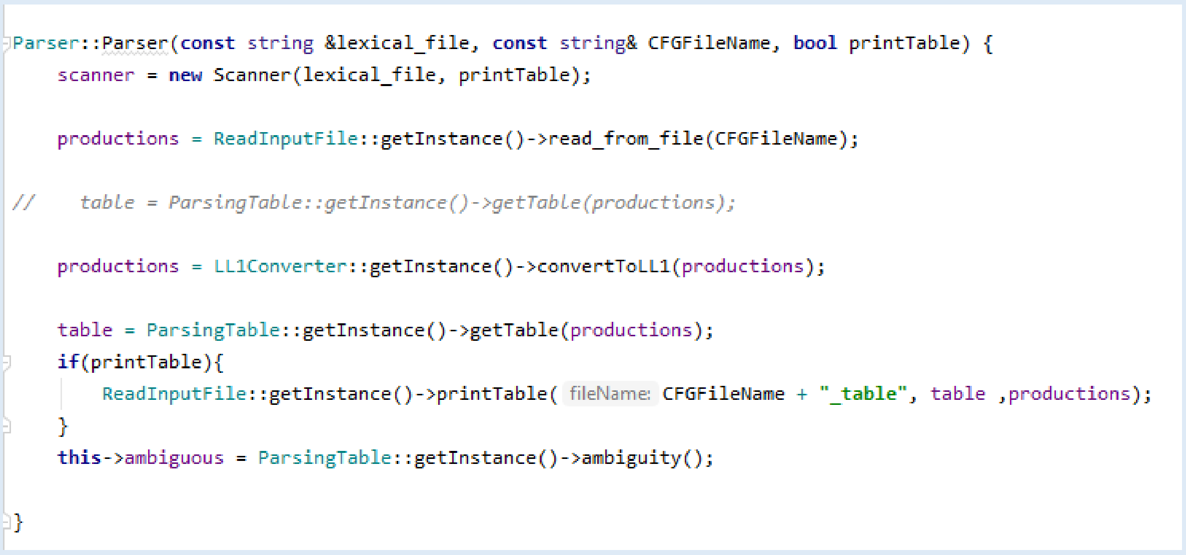
```
void ParsingTable::settingFirstANDFollow(const vector<SyntacticTerm *>& non_terminal) {
    auto* synch = new ProductionRule();
    if (!cons) {
        for (auto i:non_terminal) {
            getFirst(i);
        }
        setFollowTable(non_terminal);
        for (auto i:non_terminal) {
            i->setFollow(follow.find(i)->second);
            /// add sync to table
            if (i->isDerivingToEpsilon()){
                for (auto *pr:i->getProductions()){
                    if (pr->isEpsilon()){
                        for (const auto& c:i->getFollow()){
                            if (table.find(i)->second.find(c) != table.find(i)->second.end()){
//                                table.find(i)->second.find(c)->second = *pr;
                                amb = true;
                            } else {
                                table.find(i)->second.insert(pair <std::string, struct ProductionRule>(c,*pr));
                            }
                        }
                        break;
                    }
                }
            } else {
                for (const auto& c:i->getFollow()) {
                    if (!(table.find(i)->second.find(c) != table.find(i)->second.end())){
                        table.find(i)->second.insert(pair <std::string, struct ProductionRule>(c,*synch));
                    }
                }
            }
        }
        cons = true;
    }
}


map<SyntacticTerm *, map<std::string, struct ProductionRule>> ParsingTable::getTable(const vector
<SyntacticTerm*>& non_terminal) {
    if(non_terminal.empty()){
        return table;
    }
    settingFirstANDFollow(non_terminal);
    return table;
}
```

5. Left Most Derivation

We used Backtracking technique in Scanner that is the Scanner starts moving with the input in the graph until reaching an invalid state or the input ends then it backtracks until *reaching the last acceptance state according to Maximal Munch.*

Constructor of parser:

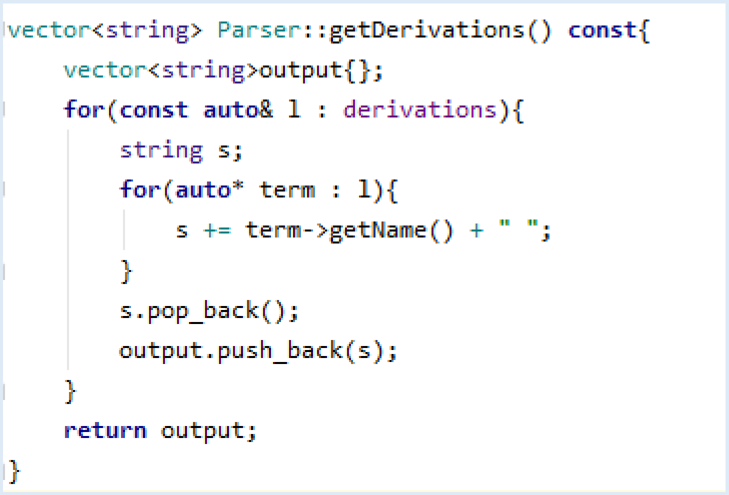
- *Read the lexical file.*
- *Get the productions and remove left recursion and left factoring.*
- *Printing table if needed. Tables of file lexical table and parsing table*
- *Put the ambiguity in ambiguous.*



```
Parser::Parser(const string &lexical_file, const string& CFGFileName, bool printTable) {
    scanner = new Scanner(lexical_file, printTable);

    productions = ReadInputFile::getInstance()->read_from_file(CFGFileName);

//    table = ParsingTable::getInstance()->getTable(productions);

    productions = LL1Converter::getInstance()->convertToLL1(productions);

    table = ParsingTable::getInstance()->getTable(productions);
    if(printTable){
        ReadInputFile::getInstance()->printTable( fileName: CFGFileName + "_table", table ,productions);
    }
    this->ambiguous = ParsingTable::getInstance()->ambiguity();

}
```

- There is vector of strings called to have errors.
- There is vector of vector of production term called derivations to have derivations.

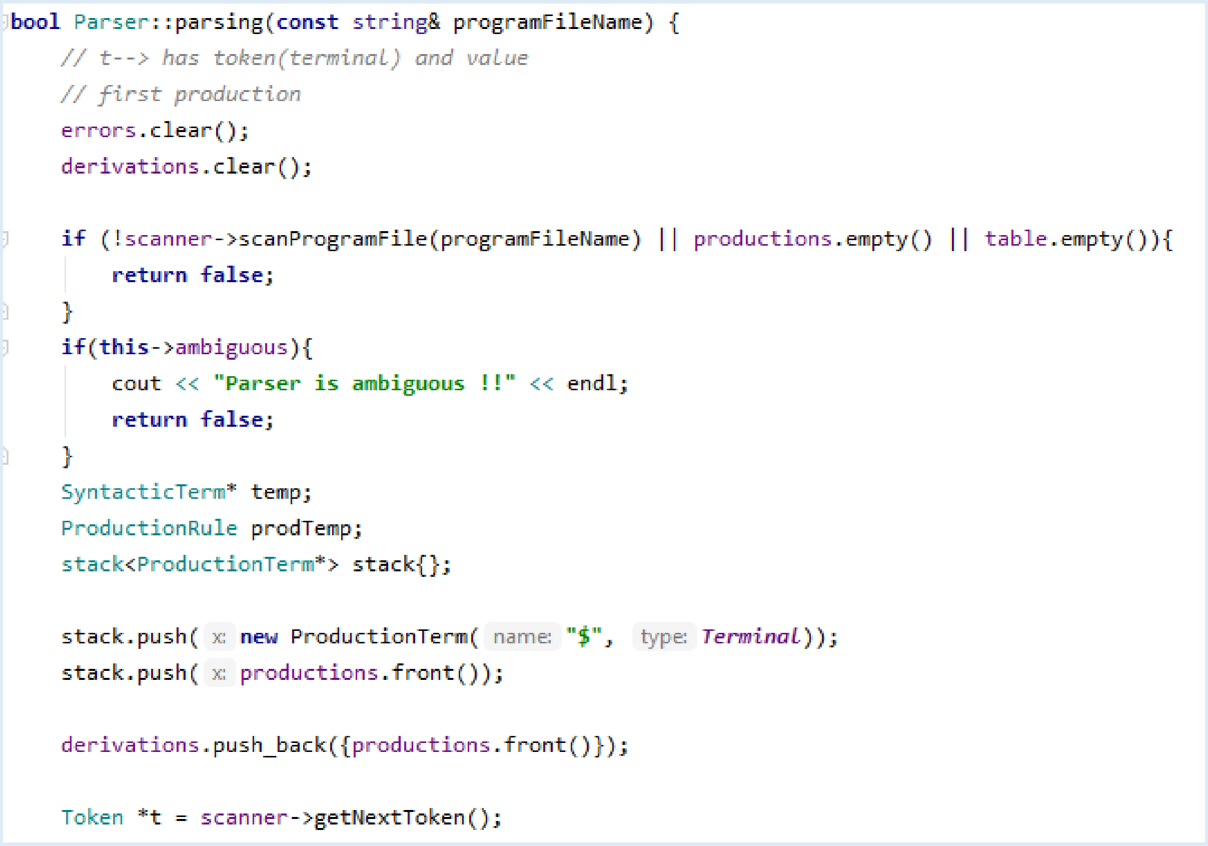
Loop on all the left most derivations to save it in output which is vector of strings.



```
vector<string> Parser::getDerivations() const{
    vector<string>output{};
    for(const auto& l : derivations){
        string s;
        for(auto* term : l){
            s += term->getName() + " ";
        }
        s.pop_back();
        output.push_back(s);
    }
    return output;
}
```

Bool function that return false if the grammar is ambiguous or if there is an error in the file or the productions or any problem in phase one while reading file, return true otherwise.

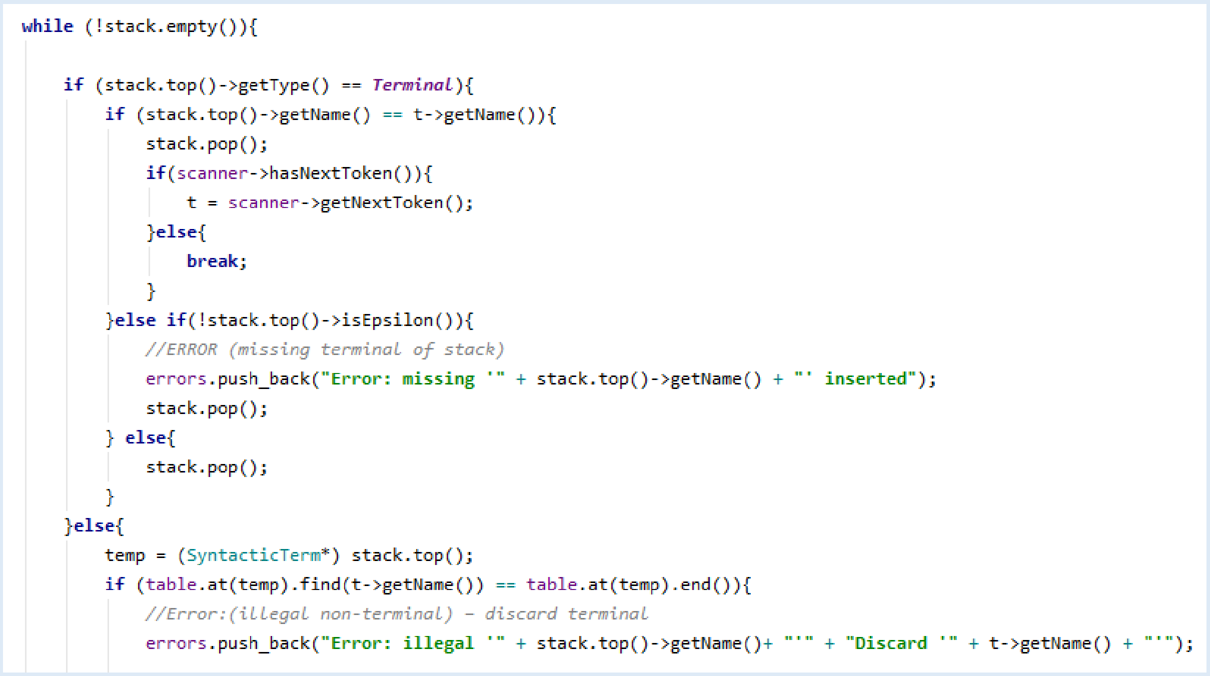
- Push in stack the dollar sign then the start production.
- Token t has the input tokens.



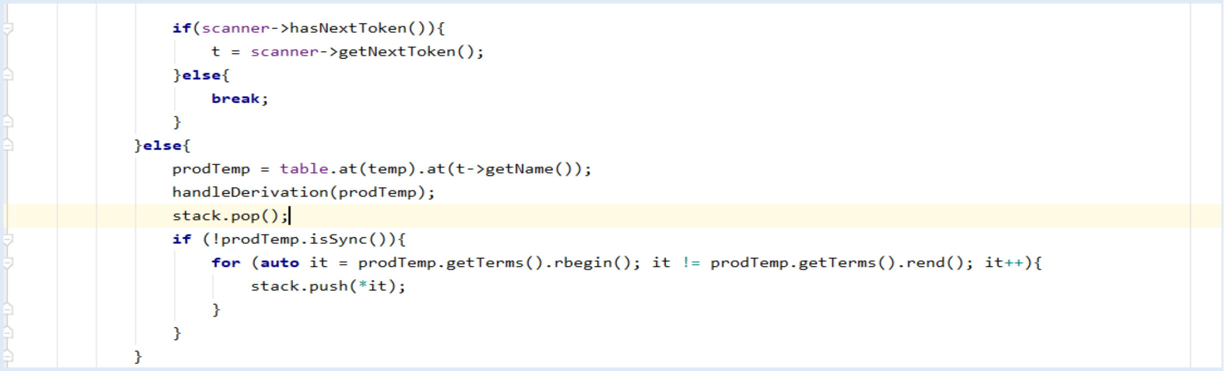
```
bool Parser::parsing(const string& programFileName) {
    // t--> has token(terminal) and value
    // first production
    errors.clear();
    derivations.clear();

    if (!scanner->scanProgramFile(programFileName) || productions.empty() || table.empty()){
        return false;
    }
    if(this->ambiguous){
        cout << "Parser is ambiguous !!" << endl;
        return false;
    }
    SyntacticTerm* temp;
    ProductionRule prodTemp;
    stack<ProductionTerm*> stack{};

    stack.push( x: new ProductionTerm( name: "$",  type: Terminal));
    stack.push( x: productions.front());

    derivations.push_back({productions.front()});

    Token *t = scanner->getNextToken();
```

- Check if the stack is empty then loop and do the algorithm or steps of left most derivation.
- Check if the stack top is terminal:
    - If so, check if the input token equals the stack top, pop the stack then get the next token.
    - If the token is not equal the stack, check if it is epsilon, if so, then pop.

        - Else, save this error as missing error.
    - If the stack top is not terminal.
        - If the stack top under the token goes to nothing, then pop the stack and save the error discard and get the next token.
        - If the stack top under the token goes to production, check if it goes to sync, calls handle derivation then pop the stack.
        - Else, pop the stack then calls handle derivation

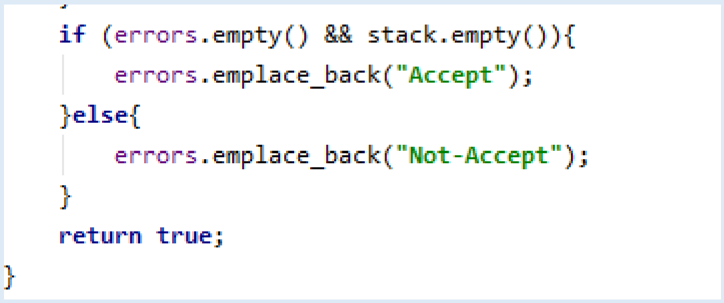


```
while (!stack.empty()){

    if (stack.top()->getType() == Terminal){
        if (stack.top()->getName() == t->getName()){
            stack.pop();
            if(scanner->hasNextToken()){
                t = scanner->getNextToken();
            }else{
                break;
            }
        }else if(!stack.top()->isEpsilon()){
            //ERROR (missing terminal of stack)
            errors.push_back("Error: missing '" + stack.top()->getName() + "' inserted");
            stack.pop();
        } else{
            stack.pop();
        }
    }else{
        temp = (SyntacticTerm*) stack.top();
        if (table.at(temp).find(t->getName()) == table.at(temp).end()){
            //Error:(illegal non-terminal) - discard terminal
            errors.push_back("Error: illegal '" + stack.top()->getName()+ "'" + "Discard '" + t->getName() + "'");
```



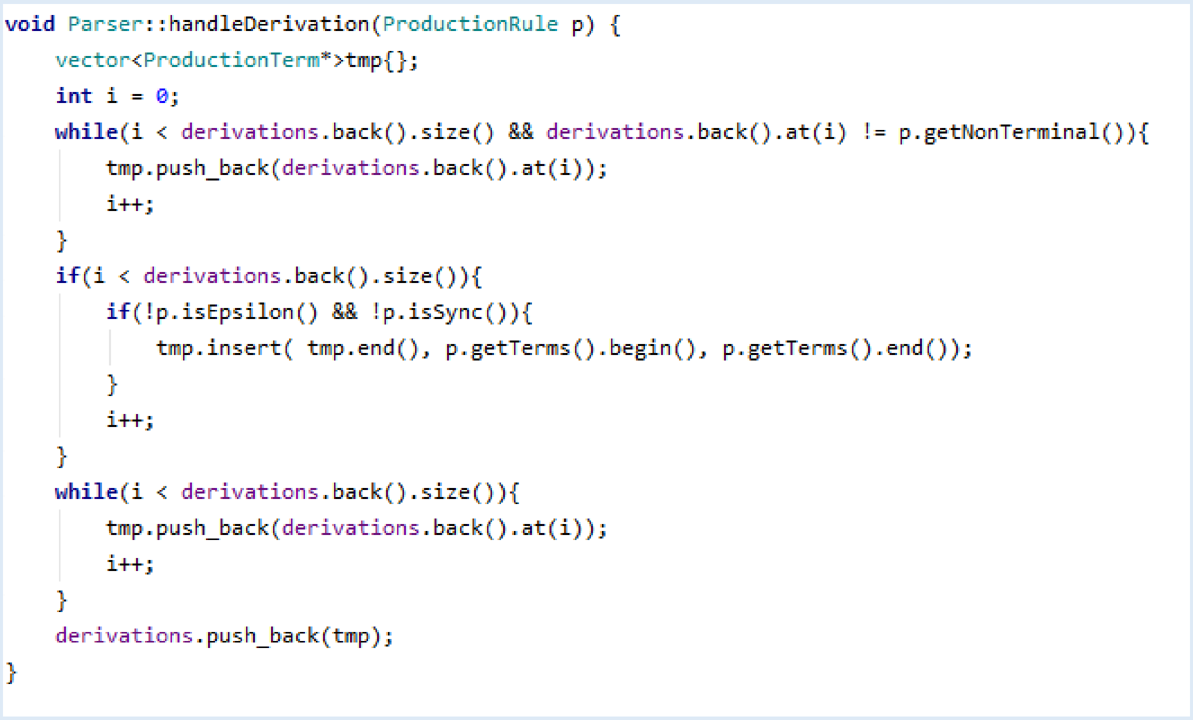
```
            if(scanner->hasNextToken()){
                t = scanner->getNextToken();
            }else{
                break;
            }
        }else{
            prodTemp = table.at(temp).at(t->getName());
            handleDerivation(prodTemp);
            stack.pop();
            if (!prodTemp.isSync()){
                for (auto it = prodTemp.getTerms().rbegin(); it != prodTemp.getTerms().rend(); it++){
                    stack.push(*it);
                }
            }
        }
```

Check if the stack and errors are empty, then accept the grammar, else not accept the grammar.



```
    if (errors.empty() && stack.empty()){
        errors.emplace_back("Accept");
    }else{
        errors.emplace_back("Not-Accept");
    }
    return true;
}
```

Handle derivation that replace one non terminal in the stack with its production.

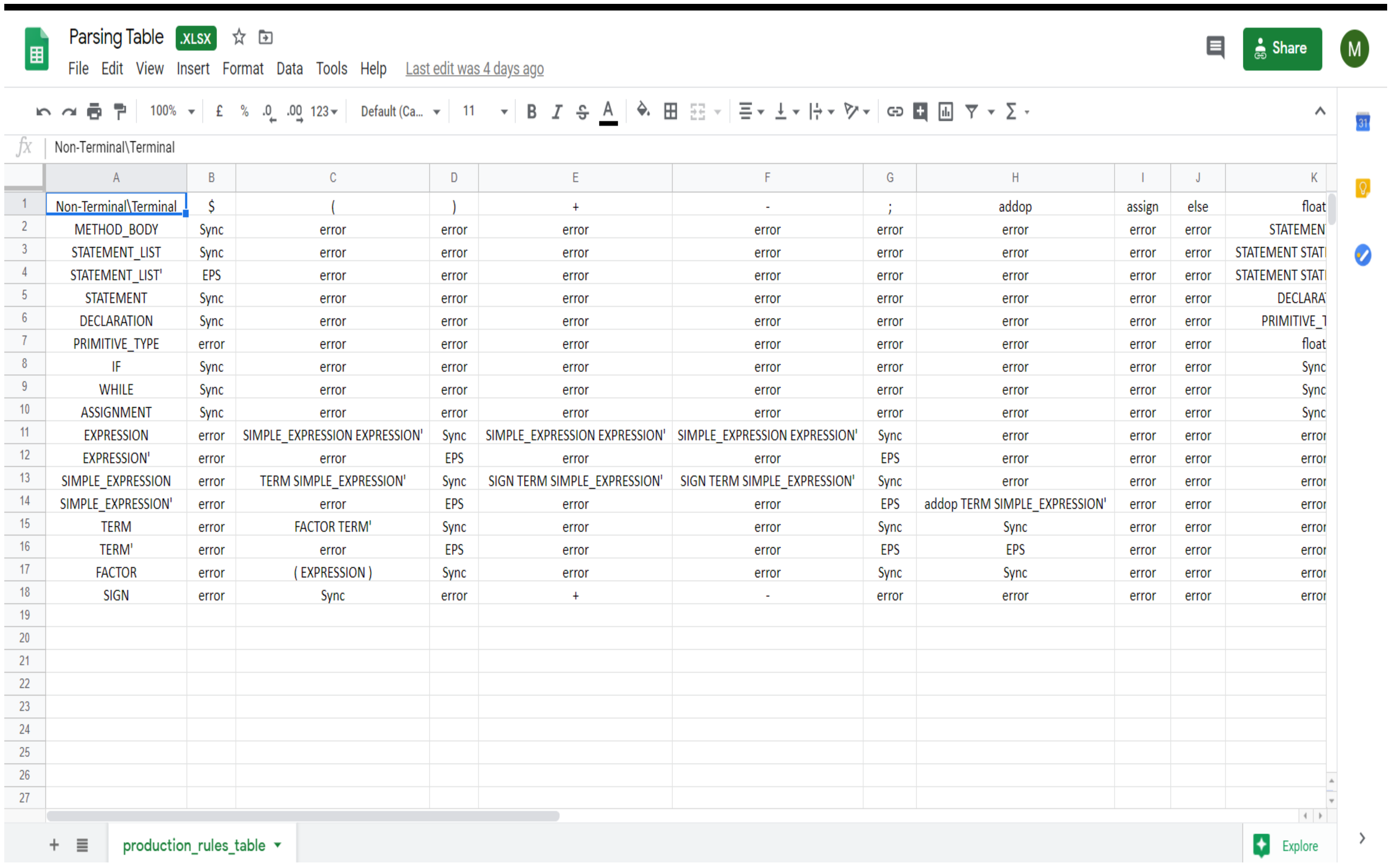


```
void Parser::handleDerivation(ProductionRule p) {
    vector<ProductionTerm*>tmp{};
    int i = 0;
    while(i < derivations.back().size() && derivations.back().at(i) != p.getNonTerminal()){
        tmp.push_back(derivations.back().at(i));
        i++;
    }
    if(i < derivations.back().size()){
        if(!p.isEpsilon() && !p.isSync()){
            tmp.insert( tmp.end(), p.getTerms().begin(), p.getTerms().end());
        }
        i++;
    }
    while(i < derivations.back().size()){
        tmp.push_back(derivations.back().at(i));
        i++;
    }
    derivations.push_back(tmp);
}
```

TRANSITION DIAGRAMS AND PARSING TABLES

Fully spreadsheet is found here for the resultant grammar table:

https://drive.google.com/file/d/1Cbf5-nn8W5NPp9cnPAML1PzFLI5SqDzj/view?usp=sharing



Non-Terminal\Terminal

Non-Terminal\Terminal	$	(	)	+	-	;	addop	assign	else	float
METHOD_BODY	Sync	error	error	error	error	error	error	error	error	STATEMEN
STATEMENT_LIST	Sync	error	error	error	error	error	error	error	error	STATEMENT STATI
STATEMENT_LIST'	EPS	error	error	error	error	error	error	error	error	STATEMENT STATI
STATEMENT	Sync	error	error	error	error	error	error	error	error	DECLARA
DECLARATION	Sync	error	error	error	error	error	error	error	error	PRIMITIVE_1
PRIMITIVE_TYPE	error	error	error	error	error	error	error	error	error	float
IF	Sync	error	error	error	error	error	error	error	error	Sync
WHILE	Sync	error	error	error	error	error	error	error	error	Sync
ASSIGNMENT	Sync	error	error	error	error	error	error	error	error	Sync
EXPRESSION	error	SIMPLE_EXPRESSION EXPRESSION'	Sync	SIMPLE_EXPRESSION EXPRESSION'	SIMPLE_EXPRESSION EXPRESSION'	Sync	error	error	error	error
EXPRESSION'	error	error	EPS	error	error	EPS	error	error	error	error
SIMPLE_EXPRESSION	error	TERM SIMPLE_EXPRESSION'	Sync	SIGN TERM SIMPLE_EXPRESSION'	SIGN TERM SIMPLE_EXPRESSION'	Sync	error	error	error	error
SIMPLE_EXPRESSION'	error	error	EPS	error	error	EPS	addop TERM SIMPLE_EXPRESSION'	error	error	error
TERM	error	FACTOR TERM'	Sync	error	error	Sync	Sync	error	error	error
TERM'	error	error	EPS	error	error	EPS	EPS	error	error	error
FACTOR	error	(EXPRESSION)	Sync	error	error	Sync	Sync	error	error	error
SIGN	error	Sync	error	+	-	error	error	error	error	error

SAMPLE RUN

INPUT FILES

Lexical Rules Input File

```
letter = a-z|A-Z
digit = 0-9
{ boolean int float}
id: letter (letter|digit) *
digits = digit+
num: digit+ | digit+. digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
{ if else while true false}
assign: =
[; , \( \) { }]
addop: \+ | -
mulop: \* | /
logop: \|\| | &&
```

Grammar Rules Input File

```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST =  STATEMENT |  STATEMENT_LIST  STATEMENT
# STATEMENT =   DECLARATION
 | IF
 | WHILE
 | ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{'
STATEMENT '}'
# WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT = 'id' 'assign' EXPRESSION ';'
# EXPRESSION =  SIMPLE_EXPRESSION | SIMPLE_EXPRESSION 'relop'
SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION =  TERM | SIGN TERM | SIMPLE_EXPRESSION
'addop' TERM
# TERM = FACTOR | TERM 'mulop' FACTOR
# FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
# SIGN = '+' | '-'
```

AFTER ELIMINATING LEFT RECURSION AND FACTORING

```
METHOD_BODY ----> STATEMENT_LIST

STATEMENT_LIST ----> STATEMENT STATEMENT_LIST'

STATEMENT_LIST' ----> STATEMENT STATEMENT_LIST' | EPS

STATEMENT ----> DECLARATION | IF | WHILE | ASSIGNMENT

DECLARATION ----> PRIMITIVE_TYPE id ;

PRIMITIVE_TYPE ----> int | float

IF ----> if ( EXPRESSION ) { STATEMENT } else { STATEMENT }

WHILE ----> while ( EXPRESSION ) { STATEMENT }

ASSIGNMENT ----> id assign EXPRESSION ;

EXPRESSION ----> SIMPLE_EXPRESSION EXPRESSION'

EXPRESSION' ----> EPS | relop SIMPLE_EXPRESSION

SIMPLE_EXPRESSION ----> TERM SIMPLE_EXPRESSION' | SIGN TERM
SIMPLE_EXPRESSION'

SIMPLE_EXPRESSION' ----> addop TERM SIMPLE_EXPRESSION' | EPS

TERM ----> FACTOR TERM'

TERM' ----> mulop FACTOR TERM' | EPS

FACTOR ----> id | num | ( EXPRESSION )

SIGN ----> + | -
```

TEST PROGRAM

```
int x;
x = 5;
if (x > 2)
{
x = 0;
}else {
  x = 45;
}
```

Analyzer Output:

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

```
STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( SIMPLE_EXPRESSION EXPRESSION' ) {
STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( TERM SIMPLE_EXPRESSION' EXPRESSION'
) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( FACTOR TERM' SIMPLE_EXPRESSION'
EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id TERM' SIMPLE_EXPRESSION'
EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id SIMPLE_EXPRESSION' EXPRESSION' )
{ STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id EXPRESSION' ) { STATEMENT } else
{ STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop SIMPLE_EXPRESSION ) {
STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop TERM SIMPLE_EXPRESSION' )
{ STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop FACTOR TERM'
SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num TERM'
SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num SIMPLE_EXPRESSION' ) {
STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { STATEMENT } else {
STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { ASSIGNMENT } else
{ STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign
EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign
SIMPLE_EXPRESSION EXPRESSION' ; } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign TERM
SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign FACTOR
TERM' SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num
TERM' SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT }
```

```
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num
SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num
EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { ASSIGNMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign EXPRESSION ; } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign SIMPLE_EXPRESSION EXPRESSION' ; }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign TERM SIMPLE_EXPRESSION' EXPRESSION' ; }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION' ; }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION' ; }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign num SIMPLE_EXPRESSION' EXPRESSION' ; }
STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign num EXPRESSION' ; } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign num ; } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; }
else { id assign num ; }
```

ASSUMPTIONS

- Grammar rules are written LHS = RHS.
- EPS is a terminal and is represented as ‘\L’ in the grammar rules file.
- We add (‘) to new productions that are made from left recursion or left factoring of the original one.

TEAM MEMBERS

Name	ID
Arsany Atef Abdo	**10**
Kirellos Malak Habib	**35**
Michael Said Beshara	**38**
Yomna Gamal El-Din Mahmoud	**63**