Programming Language & Compilers

Project Phase 2

Source Code: Syntax_analyzer

Ahmad Abdallah Waheeb (06)

Ahmed Mohamed El-Zeny (08)

Ziad Taha (20)

Mostafa Farrag (52)

Problem Statement:

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.

Objective:

This phase of the assignment aims to practice techniques for building an automatic LL (1) parser generator tool.

Design Approach:

We used concepts of OOP and Data Structure to facilitate designing this project as follows:

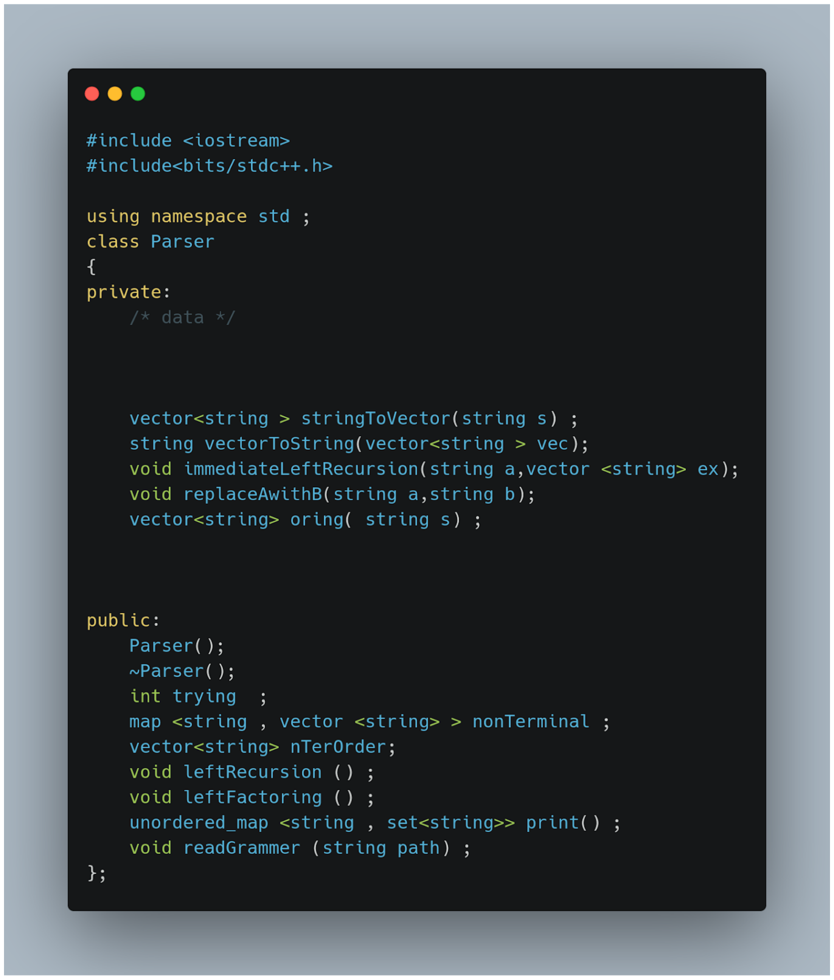
- ### Data Structures:

Class: Parser

It contains non terminals map whose key is the non terminal and values is the products of each non terminals

Non terminals vector for reserving the order of the non terminals

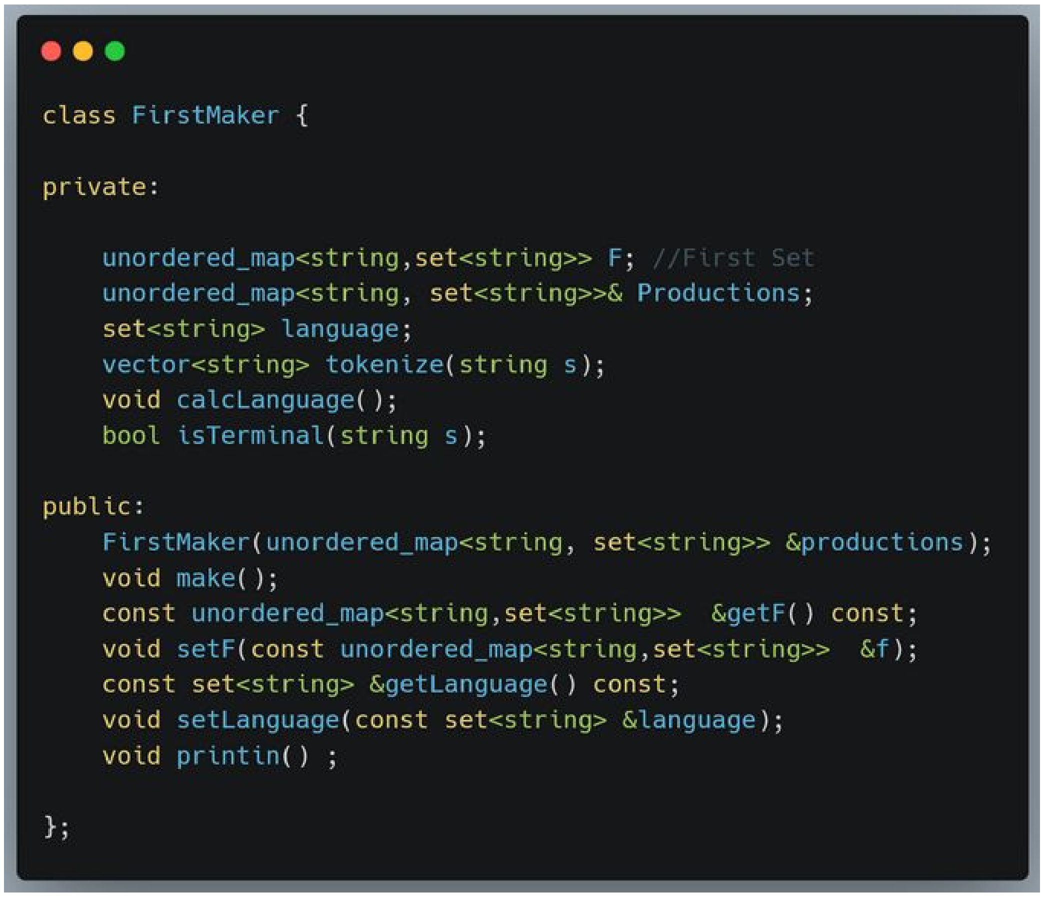
It also contains three main methods :ReadGrammer ,Left Recursion and Left Factoring.



```
#include <iostream>
#include<bits/stdc++.h>

using namespace std ;
class Parser
{
private:
    /* data */



    vector<string > stringToVector(string s) ;
    string vectorToString(vector<string > vec);
    void immediateLeftRecursion(string a,vector <string> ex);
    void replaceAwithB(string a,string b);
    vector<string> oring( string s) ;



public:
    Parser();
    ~Parser();
    int trying  ;
    map <string , vector <string> > nonTerminal ;
    vector<string> nTerOrder;
    void leftRecursion () ;
    void leftFactoring () ;
    unordered_map <string , set<string>> print() ;
    void readGrammer (string path) ;
};
```

Class: FirstMaker

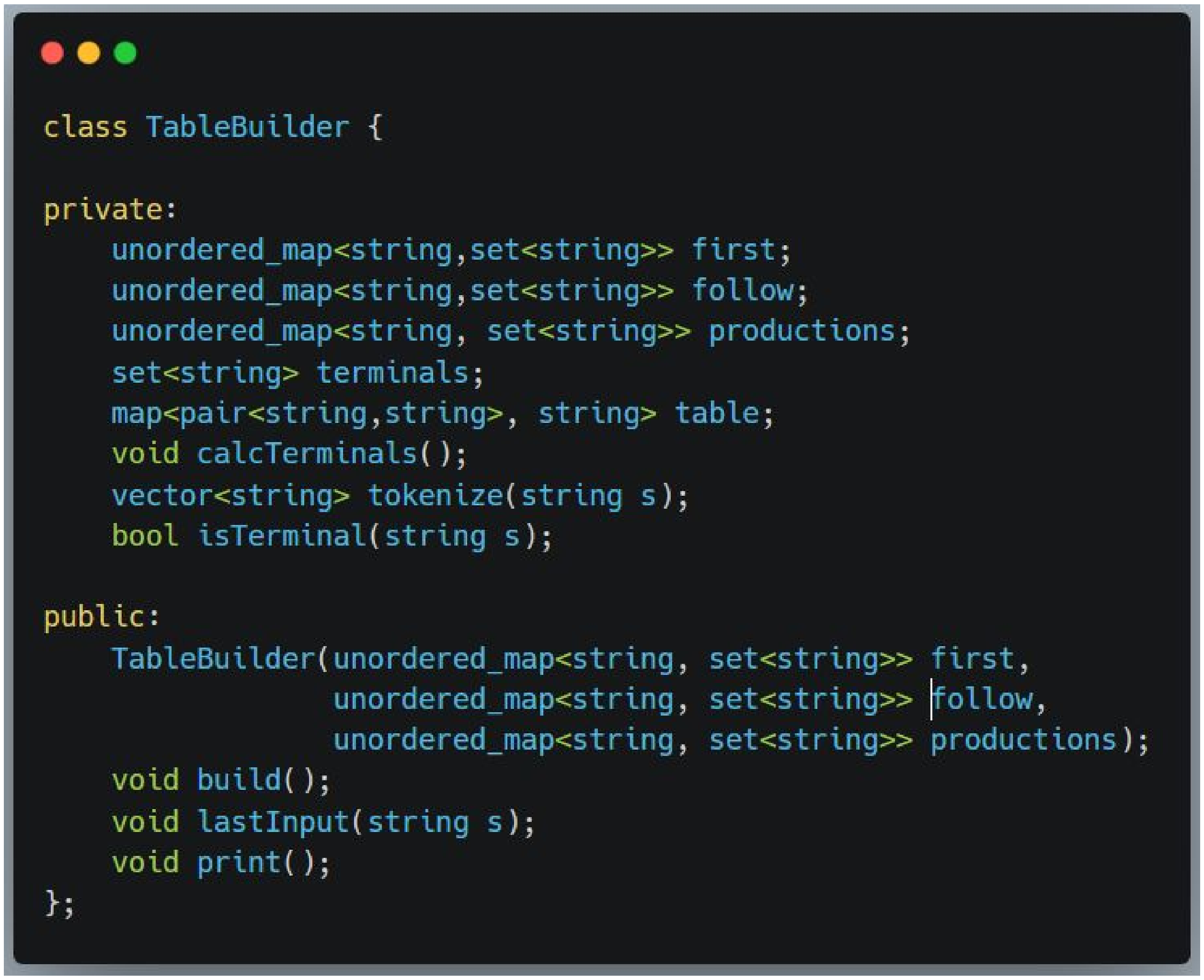
contains the productions sent by the parser object, it calculates the language stores it in a set of strings, the first set and stores it as an unordered map with a string as a key and a value of set of strings, the class contains some helper functions to do so like tokenize, and isTerminal the main function in this class is make that generates the first set in the object.



```cpp
class FirstMaker {

private:

    unordered_map<string,set<string>> F; //First Set
    unordered_map<string, set<string>>& Productions;
    set<string> language;
    vector<string> tokenize(string s);
    void calcLanguage();
    bool isTerminal(string s);

public:
    FirstMaker(unordered_map<string, set<string>> &productions);
    void make();
    const unordered_map<string,set<string>>  &getF() const;
    void setF(const unordered_map<string,set<string>>  &f);
    const set<string> &getLanguage() const;
    void setLanguage(const set<string> &language);
    void printin() ;

};
```

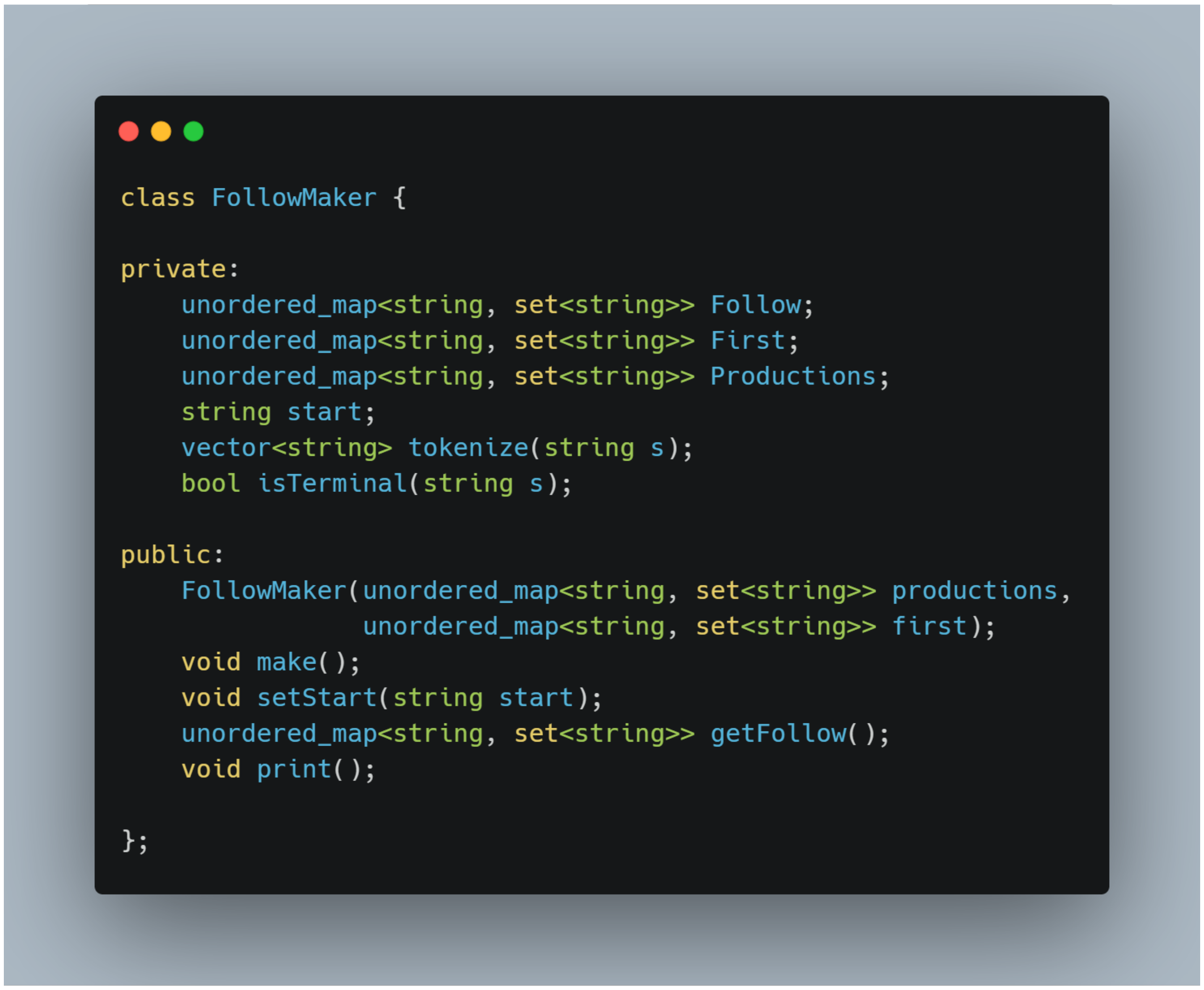
Class: TableBuilder

Contains the first set, follow set, productions, and calculated terminals. its main function “build” generates the table and stores it as a map with a pair of strings as a key <nonTerminal ,Terminal> and a value of a string.



```
class TableBuilder {

private:
    unordered_map<string,set<string>> first;
    unordered_map<string,set<string>> follow;
    unordered_map<string, set<string>> productions;
    set<string> terminals;
    map<pair<string,string>, string> table;
    void calcTerminals();
    vector<string> tokenize(string s);
    bool isTerminal(string s);

public:
    TableBuilder(unordered_map<string, set<string>> first,
                 unordered_map<string, set<string>> follow,
                 unordered_map<string, set<string>> productions);
    void build();
    void lastInput(string s);
    void print();
};
```

Class: FollowMaker



```
class FollowMaker {

private:
    unordered_map<string, set<string>> Follow;
    unordered_map<string, set<string>> First;
    unordered_map<string, set<string>> Productions;
    string start;
    vector<string> tokenize(string s);
    bool isTerminal(string s);

public:
    FollowMaker(unordered_map<string, set<string>> productions,
                unordered_map<string, set<string>> first);
    void make();
    void setStart(string start);
    unordered_map<string, set<string>> getFollow();
    void print();

};
```

- **Usage:** it's responsible for creating the Follow Set of given productions and first set.
- **Properties:**
    - Unordered_map **First**: holds the first set of the productions
    - Unordered_map **Follow**: holds the follow set of the productions
    - Unordered_map **Productions**: holds the productions that we want to get the follow set for.
    - String **start**: holds the first production LHS

- Main Algorithms Functions:

```
void readGrammer (string path) ;
```

Which read the input file and extract the nonterminals and its products and put it in the map of non terminals

```
void leftRecursion () ;
```

It removes the left recursion from the grammar

According to the following algorithm in the lectures :

```
-Arrange non-terminals in some order : A1 ......An
-for i from 1 to n do {
    -for j from 1 to i-1 do {
        Replace each production
        Ai → Aj $\beta$
        by
        Ai → $\gamma$i $\beta$ |.....| $\gamma$k $\beta$
        Where Aj → $\gamma$i | .....| $\gamma$k
    }
  -eliminate immediate left recursion among Ai products
}
```

The immediate left recursion removing algorithm

A → A α1 | ...| A αm | β1 | ......| βn Where b1 ... βn do not start with A

⬇

A →β1 A’ | ..... | βn A’

A’ → α1 A’ |..... | αm A’ | Ɛ

```
void leftFactoring () ;
```

for each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix , let say
A → Ba | Bb | Bc ... Bz
Convert it to
A → B A’
A’ → a | b | c ... z

`vector<string> FirstMaker::tokenize(string s)`

A helper function that returns a vector of string contains the tokens of a string separated by a space, we use it when calculating the language given the productions.

`void FirstMaker::calcLanguage()`

This function gets called when instantiating a first maker object, it basically uses the previous function to calculate the language of the program given all productions and stores them in the first maker’s object.

`void FirstMaker::make()`

Main function of the first maker, it basically uses the productions, the language and another helper function `bool FirstMaker::isTerminal(string s)` to make the first table.

The algorithm is based on the following pseudocode:

```
for each $\alpha \in (T \cup \text{eof} \cup \epsilon)$ do;
    $\text{FIRST}(\alpha) \leftarrow \alpha$;
end;
for each $A \in NT$ do;
    $\text{FIRST}(A) \leftarrow \emptyset$;
end;
while (FIRST sets are still changing) do;
   for each $p \in P$, where $p$ has the form $A \rightarrow \beta$ do;
       if $\beta$ is $\beta_1 \beta_2 \ldots \beta_k$, where $\beta_i \in T \cup NT$, then begin;
         $rhs \leftarrow \text{FIRST}(\beta_1) - \{\epsilon\}$;
         $i \leftarrow 1$;
         while ($\epsilon \in \text{FIRST}(\beta_i)$ and $i \leq k-1$) do;
             $rhs \leftarrow rhs \cup (\text{FIRST}(\beta_{i+1}) - \{\epsilon\})$;
             $i \leftarrow i + 1$;
         end;
       end;
      if $i = k$ and $\epsilon \in \text{FIRST}(\beta_k)$
          then $rhs \leftarrow rhs \cup \{\epsilon\}$;
      $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup rhs$;
   end;
end;
```

`void TableBuilder::calcTerminals()`

This function gets called when instantiating a table builder object, it basically separates the terminals from the production and stores them in the table builder's object.

`void TableBuilder::build()`

Main function of table builder, it uses all the class's attributes to build the parsing table.

It iterates each slot in the table inserting error, then goes product by product, each terminal by terminal, filling only the terminals of the product's first set, when done, filling every slot whose terminal in the follow set, with epson (if it was originally in it's first set), or synch if it wasn't, putting in mind that if a slot is filled with a product, we ignore putting synch.

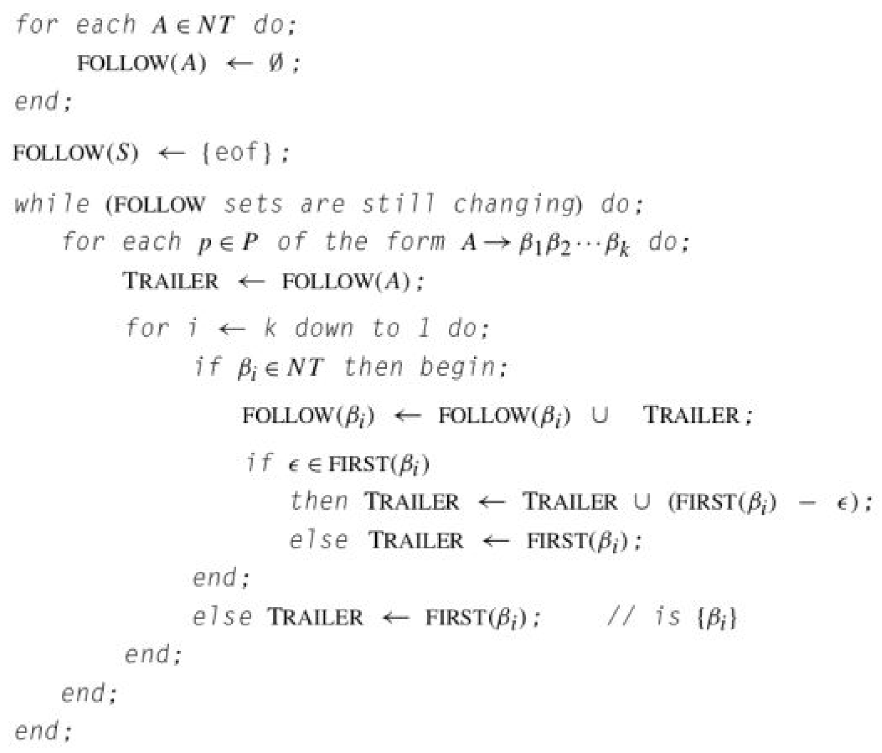
`void FollowMaker::make()`

This function holds the main algorithm responsible for creating the follow set.

At the beginning of it, it first pushes “$” to the first production LHS follow set. Then it loops over all productions until now new changes are made to any of production LHS follow sets.

Inside that loop, for each production LHS, we iterate over all RHS options, and iterate each token of that RHS in reverse, we check if that token is a Not Terminal using the IsTerminal() method. If it’s a terminal we add it to a trailer so that it’s the follow of the preceding token. If it’s not a terminal, we add the trailer to its follow set. Then we check if its first set contains **ε**. If it does we add that set minus the **ε** to the trailer. If not we reset the trailer to that first set.

Pseudocode:



```
for each $A \in NT$ do;
    FOLLOW($A$) ← $\emptyset$;
end;

FOLLOW($S$) ← {eof};

while (FOLLOW sets are still changing) do;
   for each $p \in P$ of the form $A \rightarrow \beta_1 \beta_2 \cdots \beta_k$ do;
       TRAILER ← FOLLOW($A$);
       for $i$ ← $k$ down to 1 do;
           if $\beta_i \in NT$ then begin;
               FOLLOW($\beta_i$) ← FOLLOW($\beta_i$) ∪ TRAILER;
               if $\epsilon \in$ FIRST($\beta_i$)
                  then TRAILER ← TRAILER ∪ (FIRST($\beta_i$) − $\epsilon$);
                  else TRAILER ← FIRST($\beta_i$);
           end;
           else TRAILER ← FIRST($\beta_i$);     // is {$\beta_i$}
       end;
   end;
end;
```

■ **FIGURE 3.8** Computing FOLLOW Sets for Non-Terminal Symbols.

- Parsing Table

  [parsing table]

- example

  [transitions]