Programming languages & Compiler

Java compiler

Phase 2 : Parser generator

Team members

- Aya Ashraf (1)
- Rowan Adel (25)
- Sarah Ahmed (29)
- Sohayla Mohammed (32)

Introduction

Overview

This report will cover the structure and design we used to implement a Parser generator.

The tool was implemented with java in mind, but can be generalized to work with any language. We'll also summarize some limitations that we were faced with and assumptions we made along the way that eventually led us to the proposed design.

Problem statement

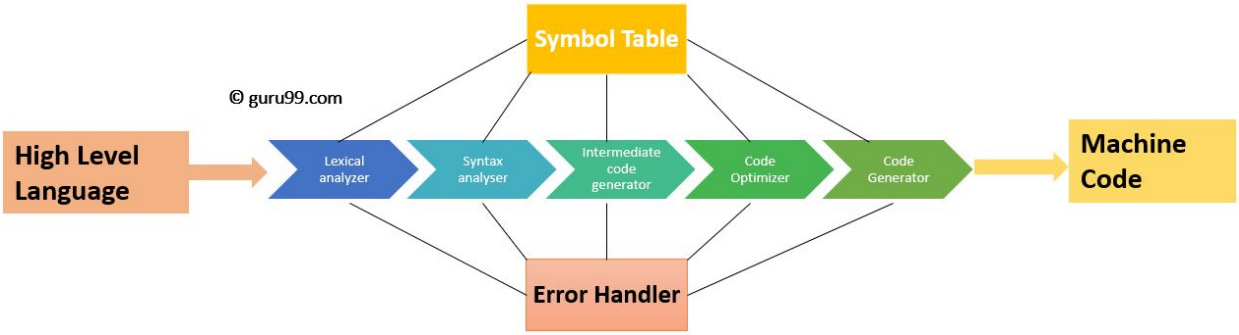
Design and implement a Parser generator tool using C++. The generated parser is required to produce some representation of the leftmost derivation for a correct input. The parser generator is required to be tested using the given context free grammar of a small subset of Java. It's required to combine the lexical analyzer generated in phase 1 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.

The tool should follow the phases studied in class :

- Produce LL(1) grammar.
- Find first.
- Find follow
- Construct parser table.

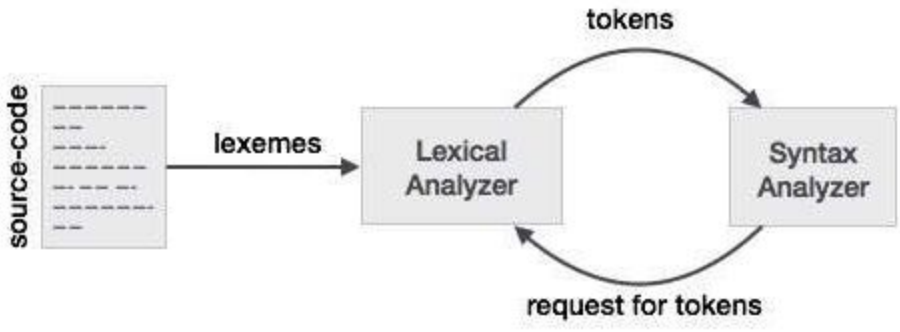
Background

The whole project aims to develop a suitable Syntax Directed Translation Scheme to convert Java code to Java bytecode, performing necessary lexical, syntax and static semantic analysis (such as type checking and Expressions Evaluation).



Phase Two is Parser Generator

- The parser generator expects an LL (1) grammar as input.
- It constructs a predictive parsing table for the grammar. The table is to be used to drive a predictive top-down parser.
- It should be linked with phase 1, by receiving the output token whenever a request is sent to the lexical analyzer.



Limitations

- Implementing optimized Left-Factoring, Left-Recursion removal algorithms.

Implementation

Overall design

Structure of Automata

Structure

LL(1) Grammar

- We represent the LL(1) grammar as a table in the form of a map
    - The table represents each non-terminal and its production(s).

First

- An Object of TableObject is created, which contains :
    - The first's value and information about the production.
- We represents the First's table as a map, which :
    - Represents each non terminal and the list of TableObject(s) associated with it.

Follow

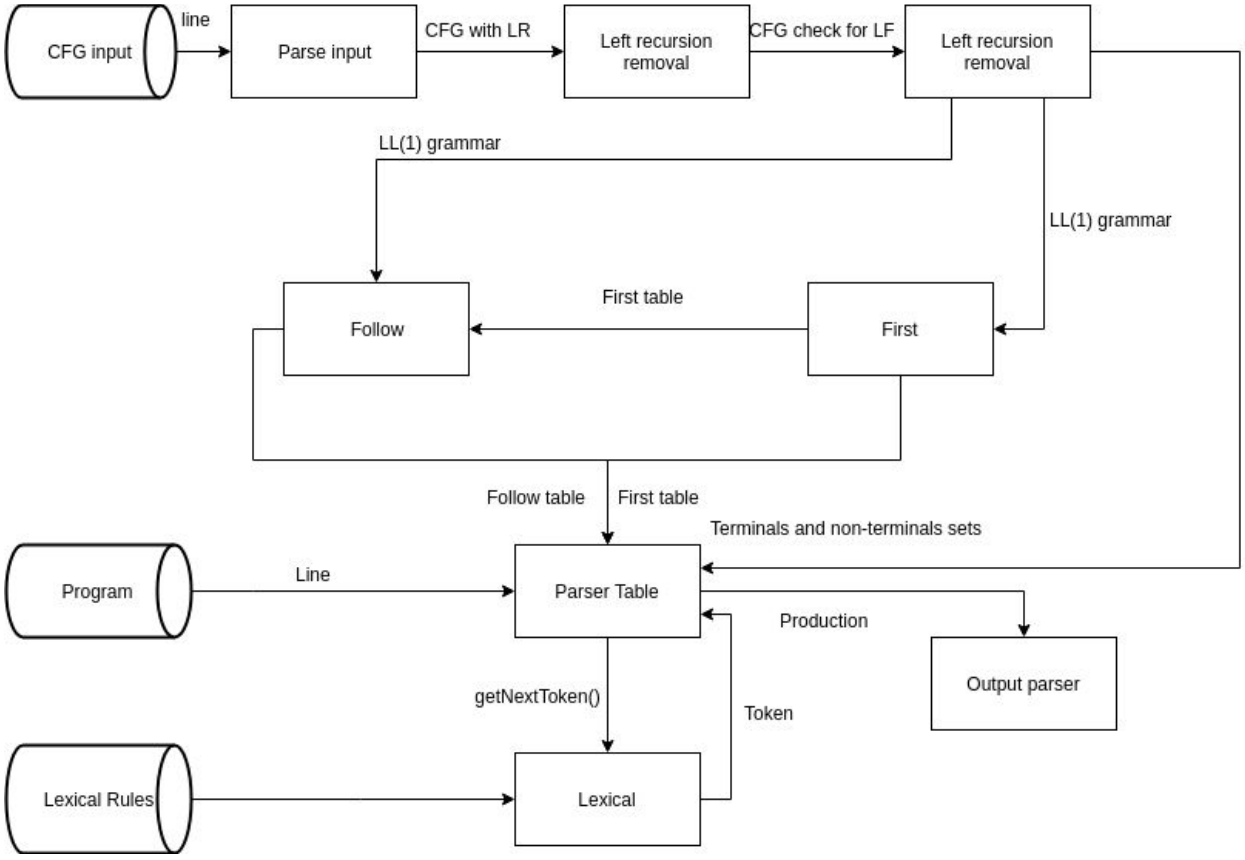
- We represents the Follow's table as a map, which :
    - Represents each non terminal and the list of follow(s) associated with it.

Parsing Table

- Represented as a table, where each row corresponds to a non-terminal, and each column corresponds to the set of terminals, and each entry is a production.
- We visualized this table as a map of map to give the illusion of a table.

Phases

We divided the lexical analyser into 7 phases represented in the diagram below.



We wanted the code to be as modular as possible to allow modifications along the way without worrying about changing a huge bulk of the code.

Design phases

File Parsing

Overview

Input

Grammar file

Output

CFG

Terminals

NonTerminals

Productions

Start Symbol

Algorithm

Data-structure

- vector<string>
    - For tokens.
- `string filename;`
- `CFG* grammer;`
- `set<string> nonTerminalOnRHS;`

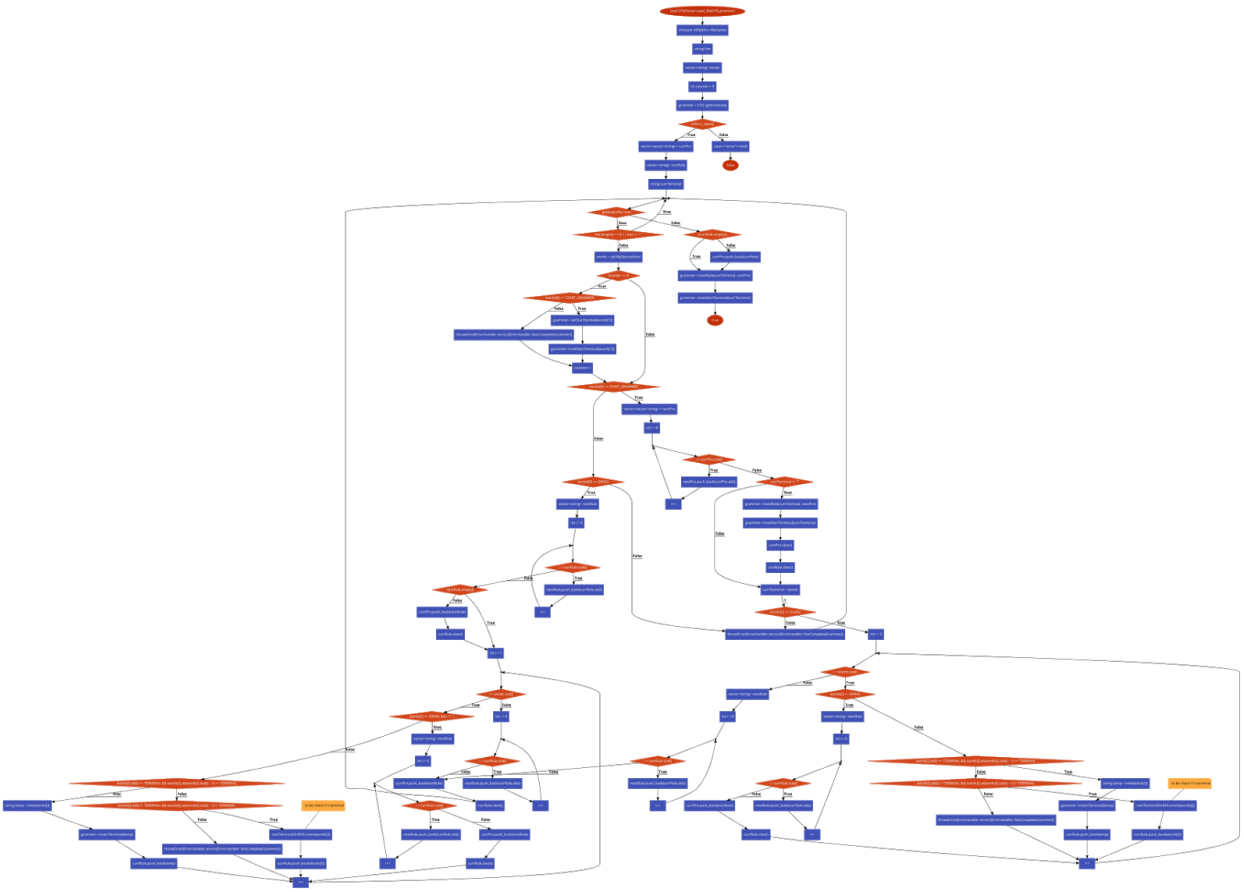
Main classes

CFG

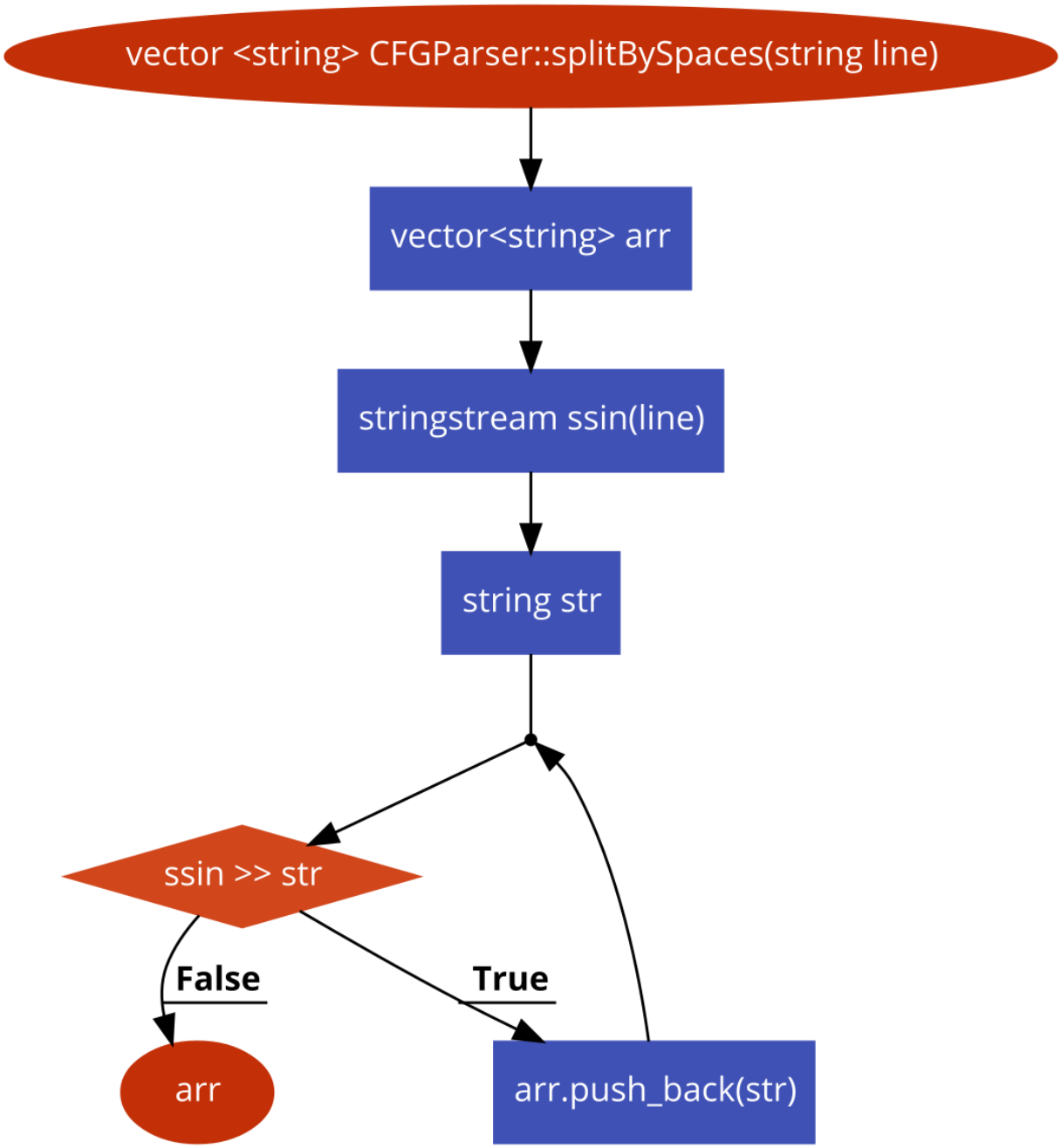
CFG Parser

Main Functions

```
bool read_file(CFG* grammer);
```

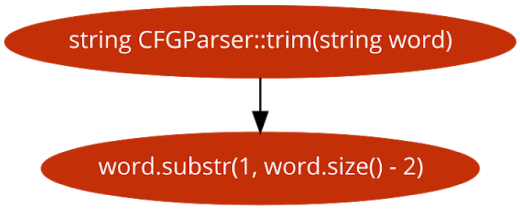


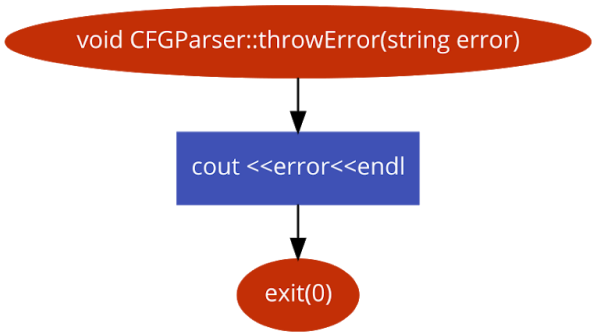
```
vector <string> splitBySpaces(string line);
```



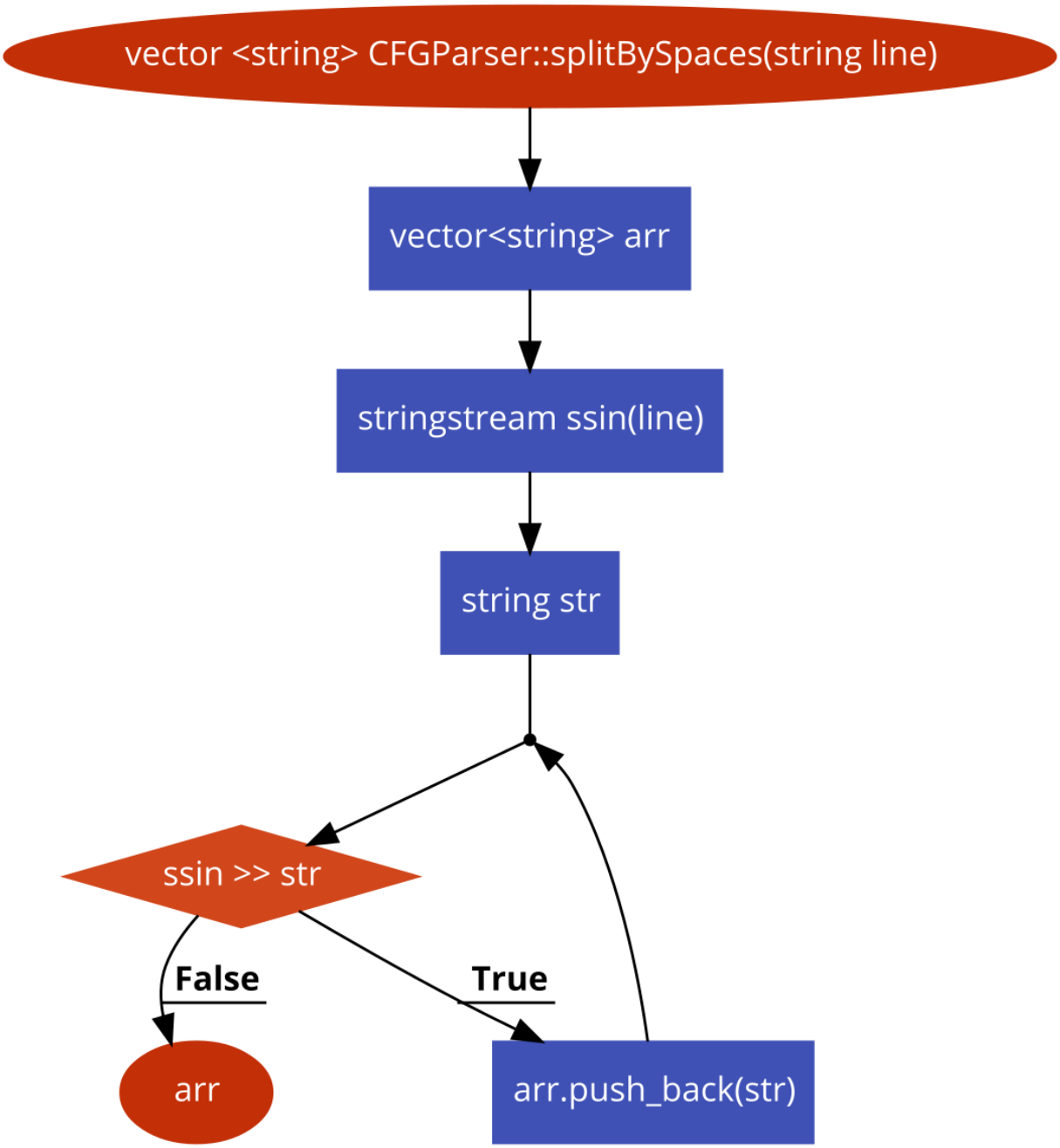
```
string trim(string word);

void throwError(string error);
```





```
bool checkNonTerminal(set<string> tokens, CFG* cfg);
```



Assumptions

1. Handle beginning with # and | only
2. = is handle to be assgin

Alternative designs

No

Left Recursion

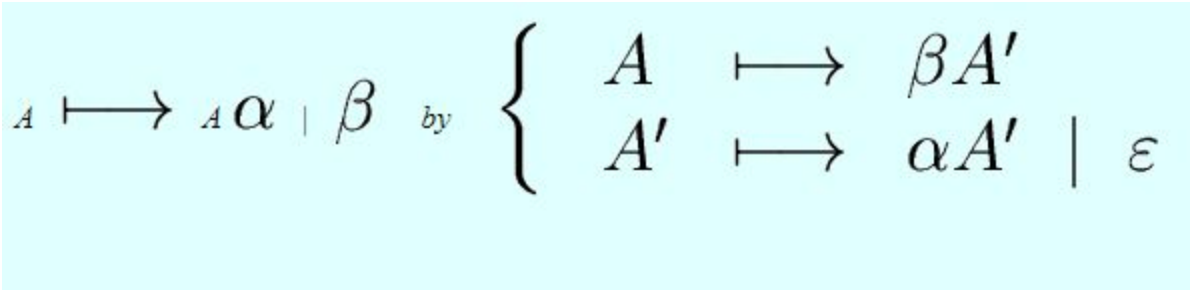
Overview

Left Recursion: A grammar is left recursive if it has a nonterminal A such that there is a derivation **A -> Aα | β** where α and β are sequences of terminals and nonterminals that do not start with A.

While designing a top down-parser, if the left recursion exist in the grammar then the parser falls in an infinite loop, here because A is trying to match A itself, which is not possible. We can eliminate the above left recursion by rewriting the offending production. As-

A -> βA'

A' -> αA' | epsilon



Input :-

Production with left factoring and left recursion

Output

Production without left recursion and with left factoring

Algorithm

Algorithm
EliminateLeftRecursion
Arrange the nonterminals in some order, A_1, A_2, A_3, ... A_n.
FOR i := 1 TO n DO
BEGIN {FOR i}
FOR j := 1 TO n DO
BEGIN {FOR j}
(1) Replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions:

$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

where:

$A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

are all the current A_j - productions.
(2) Eliminate the direct left recursion from the A_i productions
END {FOR j}
END {FOR i}

Data-structure

```
CFG* cfg;
```

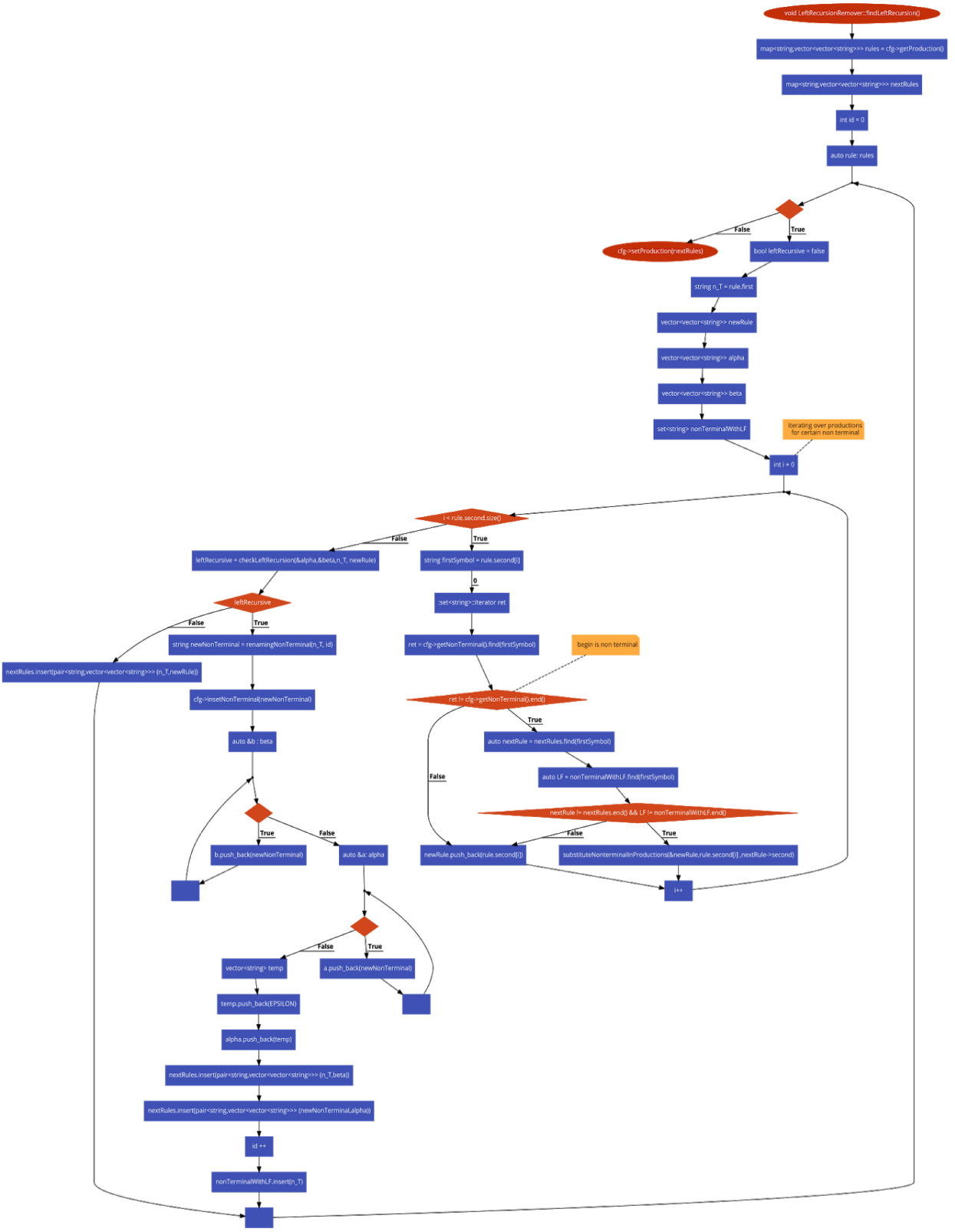
To get productions and non terminals

Main classes

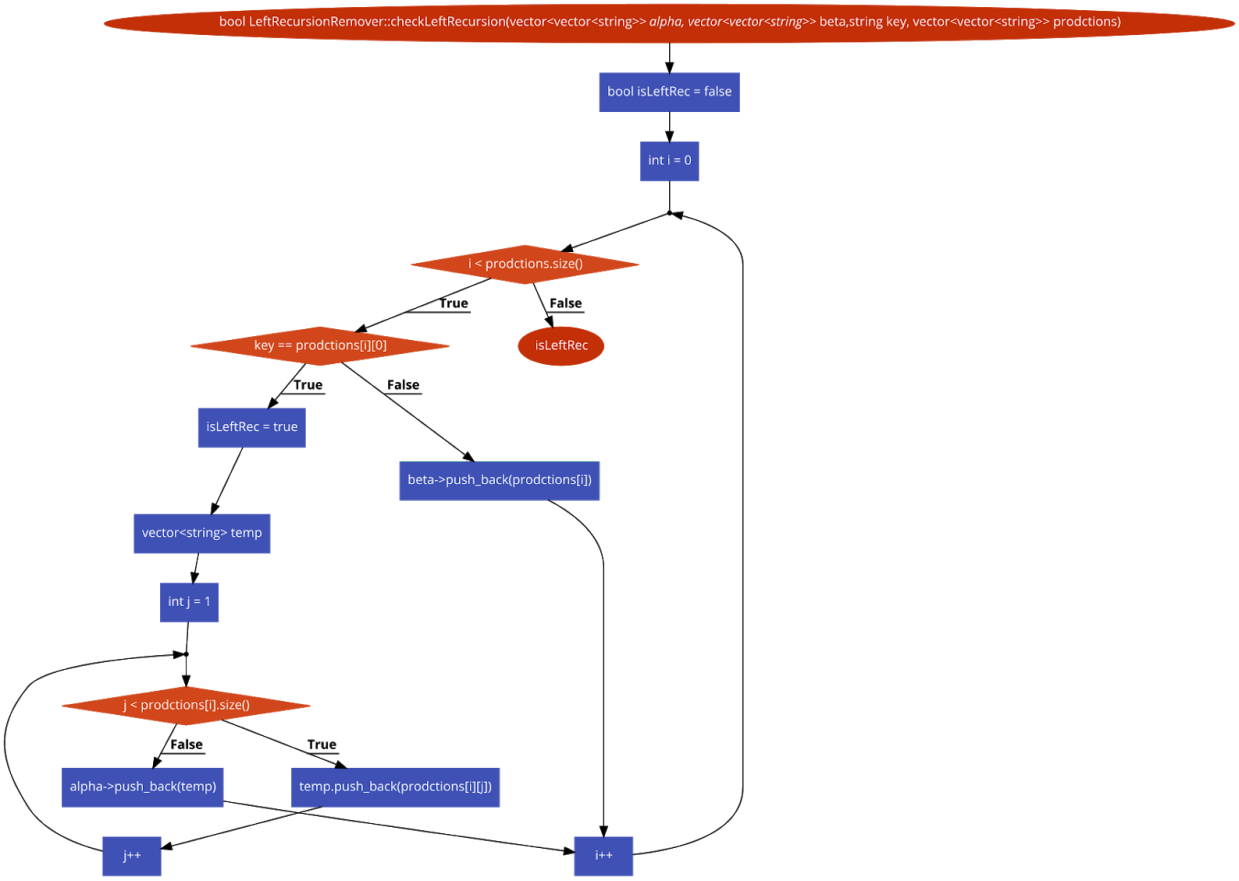
```
LeftRecursionRemover
```

Main Functions

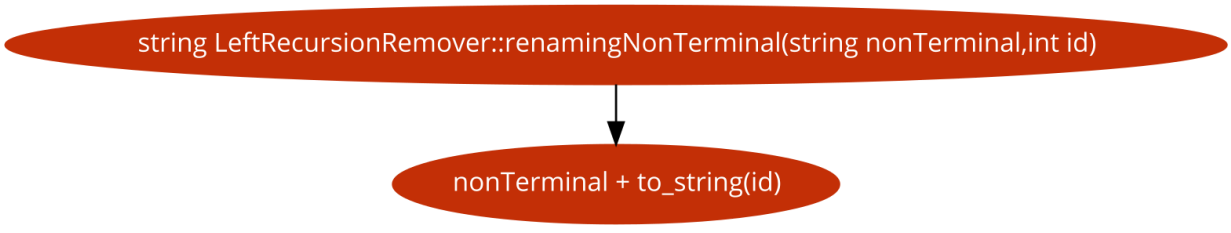
```
void findLeftRecursion();
```


void LeftRecursionRemover::findLeftRecursion()
map<string,vector<vector<string>>> rules = cfg->getProduction()
map<string,vector<vector<string>>> nextRules
int id = 0
auto rule: rules
False
True
cfg->setProduction(nextRules)
bool leftRecursive = false
string n_T = rule.first
vector<vector<string>> newRule
vector<vector<string>> alpha
vector<vector<string>> beta
set<string> nonTerminalWithLF
iterating over productions for certain non terminal
int i = 0
i < rule.second.size()
False
True
leftRecursive = checkLeftRecursion(&alpha,&beta,n_T, newRule)
string firstSymbol = rule.second[i]
0
:set<string>::iterator ret
leftRecursive
False
True
string newNonTerminal = renamingNonTerminal(n_T, id)
ret = cfg->getNonTerminal().find(firstSymbol)
begin is non terminal
nextRules.insert(pair<string,vector<vector<string>>> (n_T,newRule))
cfg->insetNonTerminal(newNonTerminal)
ret != cfg->getNonTerminal().end()
True
auto &b : beta
auto nextRule = nextRules.find(firstSymbol)
False
auto LF = nonTerminalWithLF.find(firstSymbol)
nextRule != nextRules.end() && LF != nonTerminalWithLF.end()
True
False
False
True
b.push_back(newNonTerminal)
auto &a: alpha
newRule.push_back(rule.second[i])
substituteNonterminalInProductions(&newRule,rule.second[i],nextRule->second)
i++
False
True
vector<string> temp
a.push_back(newNonTerminal)
temp.push_back(EPSILON)
alpha.push_back(temp)
nextRules.insert(pair<string,vector<vector<string>>> (n_T,beta))
nextRules.insert(pair<string,vector<vector<string>>> (newNonTerminal,alpha))
id ++
nonTerminalWithLF.insert(n_T)

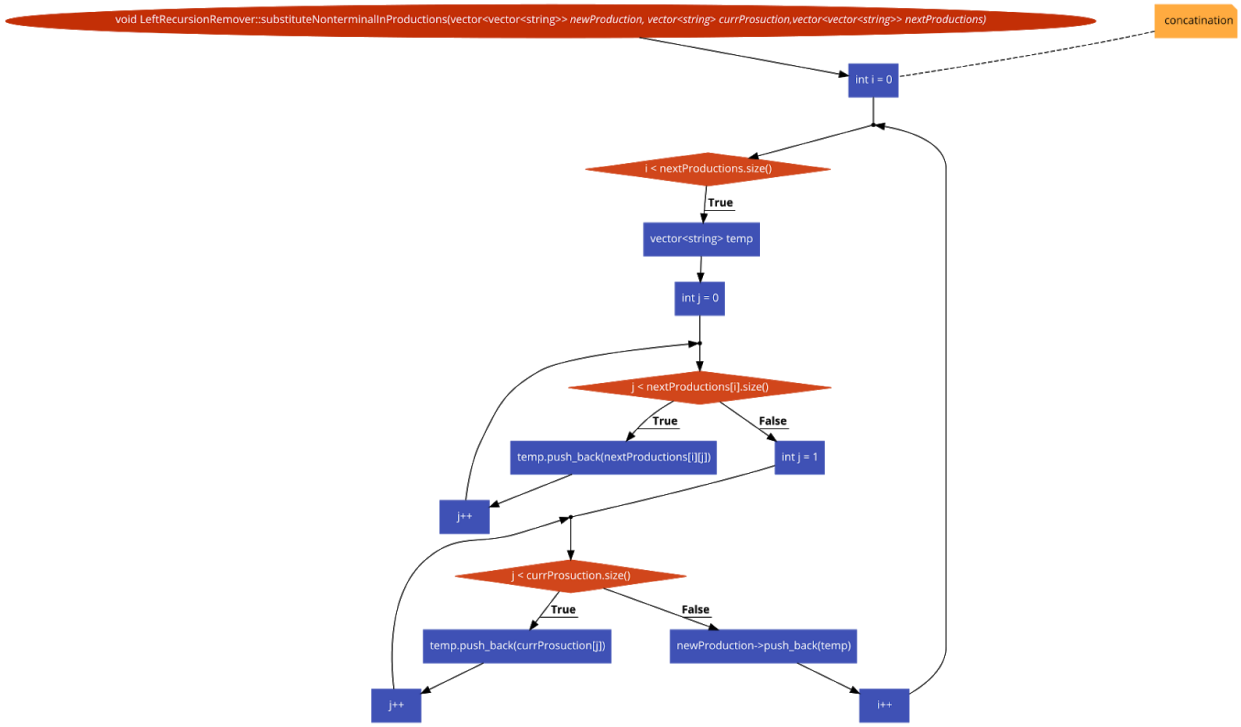
```
bool checkLeftRecursion (vector<vector<string>> *alpha,
vector<vector<string>> *beta, string key, vector<vector<string>>
prodctions);
```




```
string renamingNonTerminal(string nonTerminal,int id);
```



```
void substituteNonterminalInProductions(vector<vector<string>>
*newProduction, vector<string>
currProsuction,vector<vector<string>> nextProductions);
```



Assumptions

No assumption

Alternative designs

No Alternative desgin

Left Factoring

Overview

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead ,consider this example-

A -> qB | qC

where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

A -> qD

D -> B | C

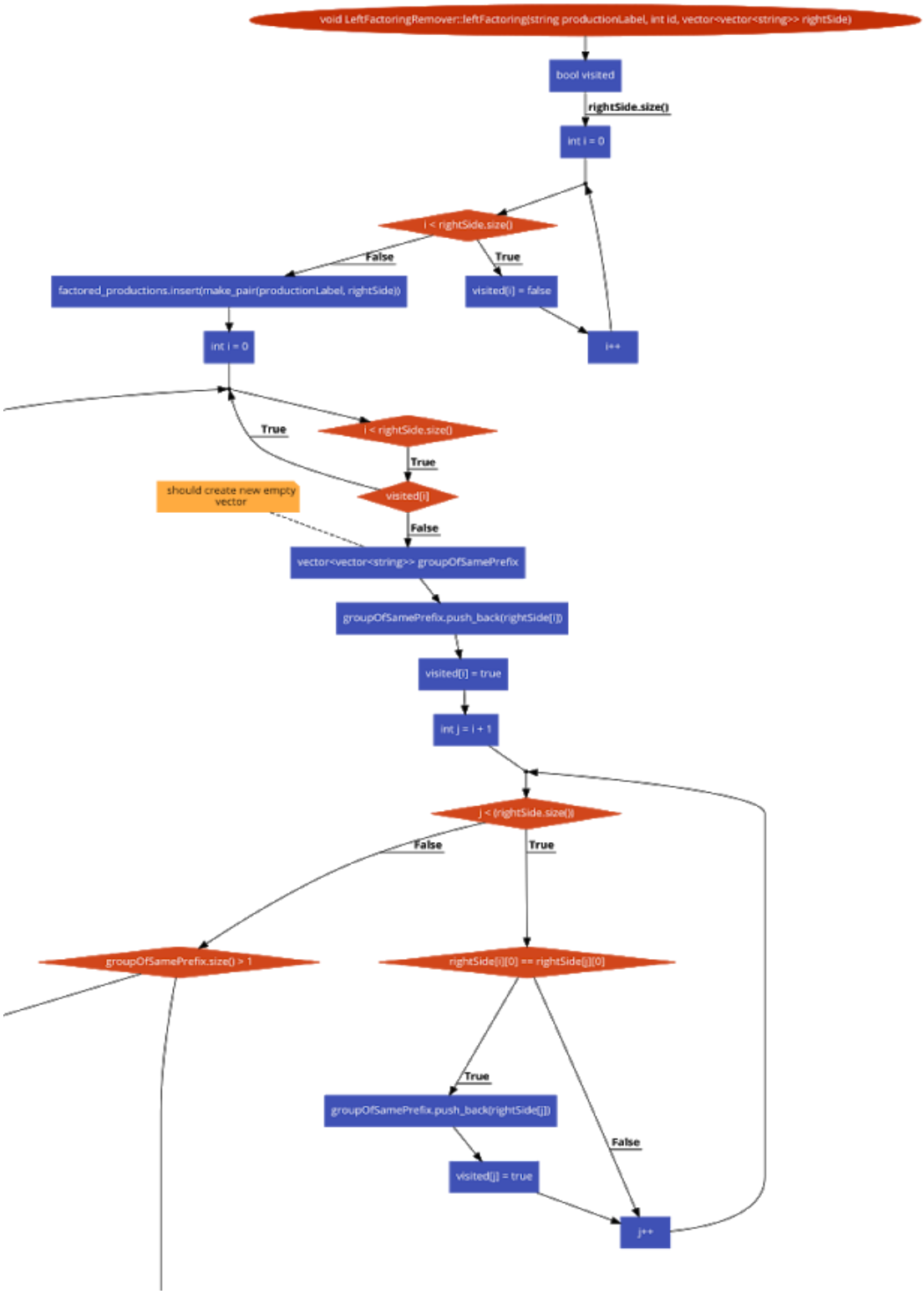
Input

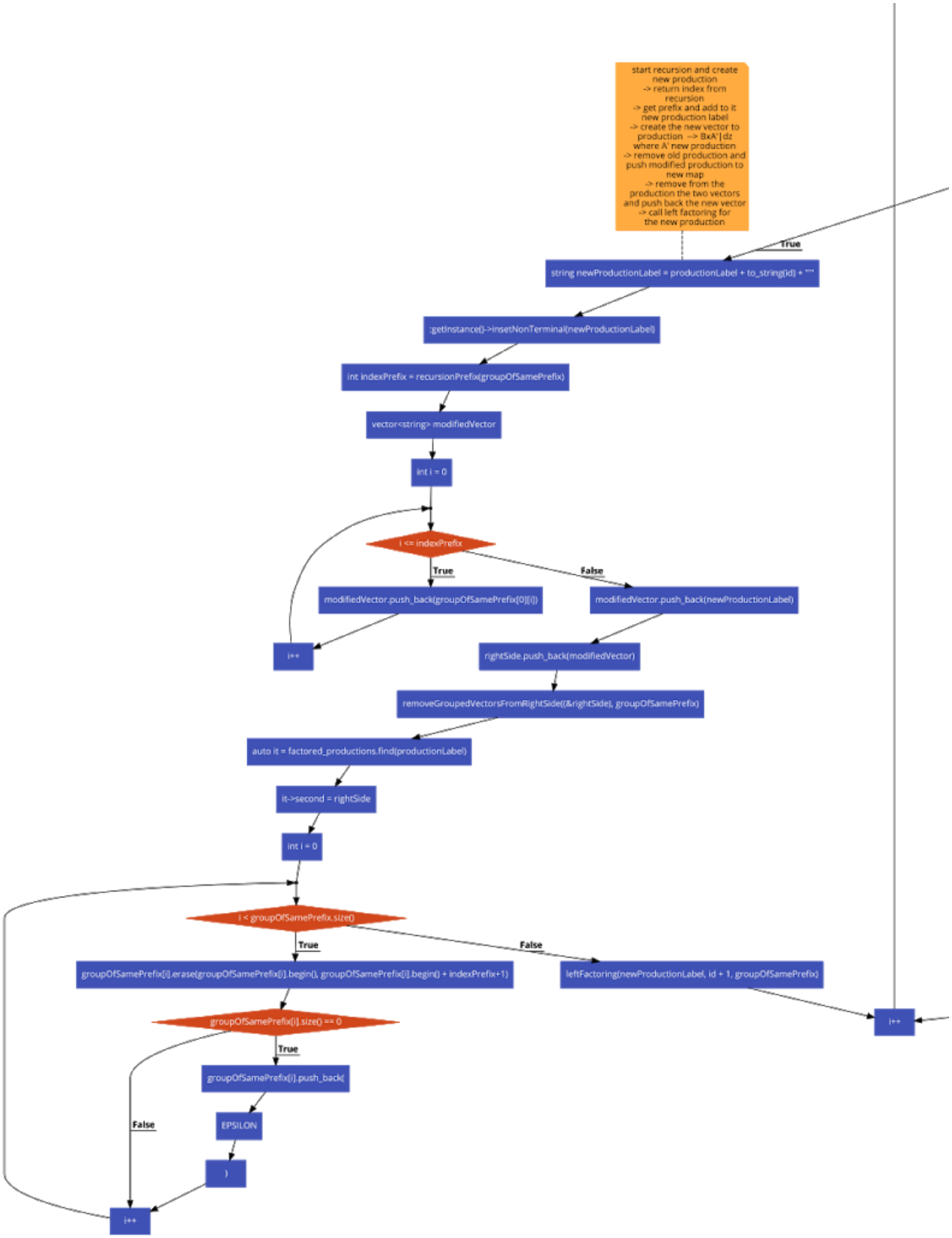
$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

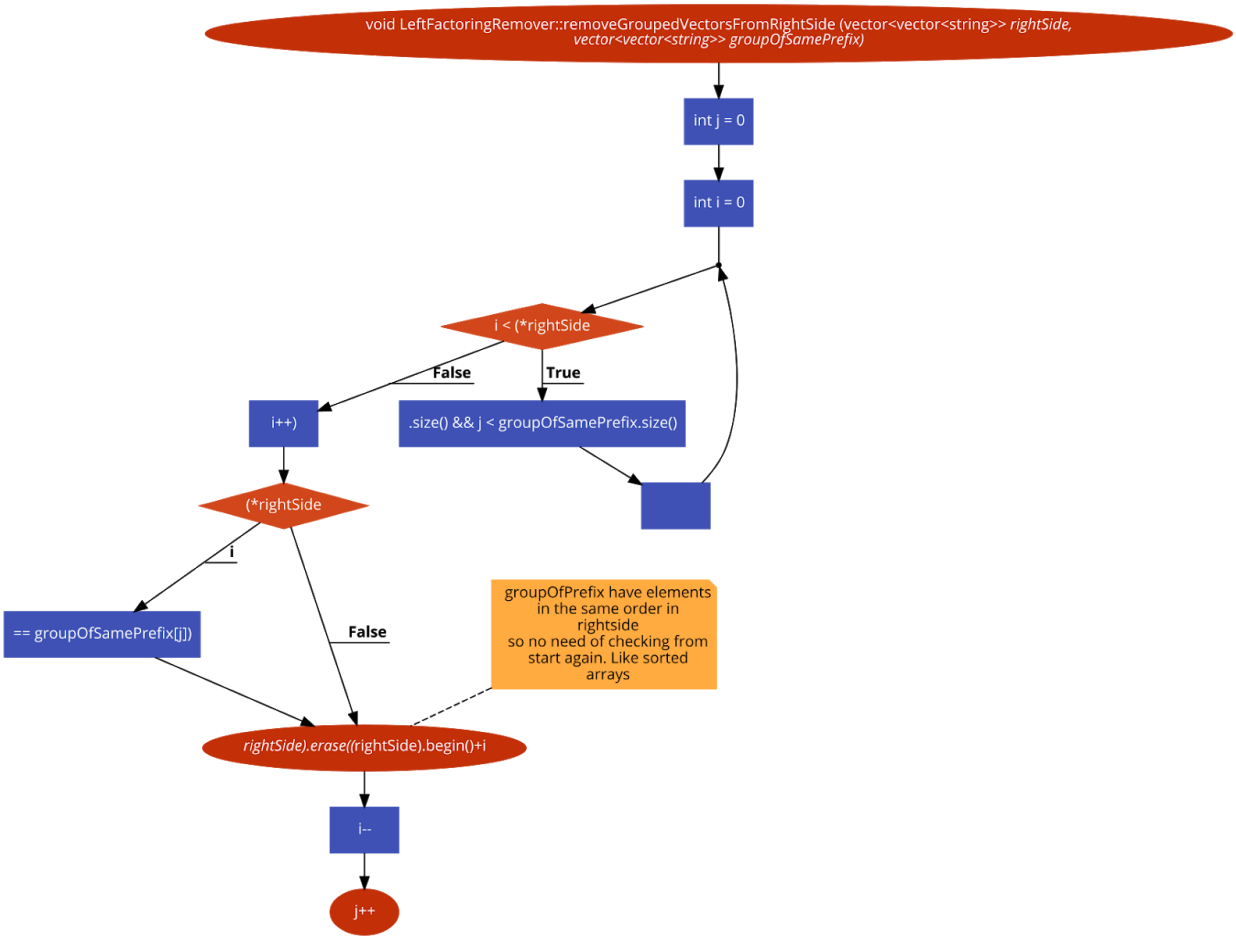
Output

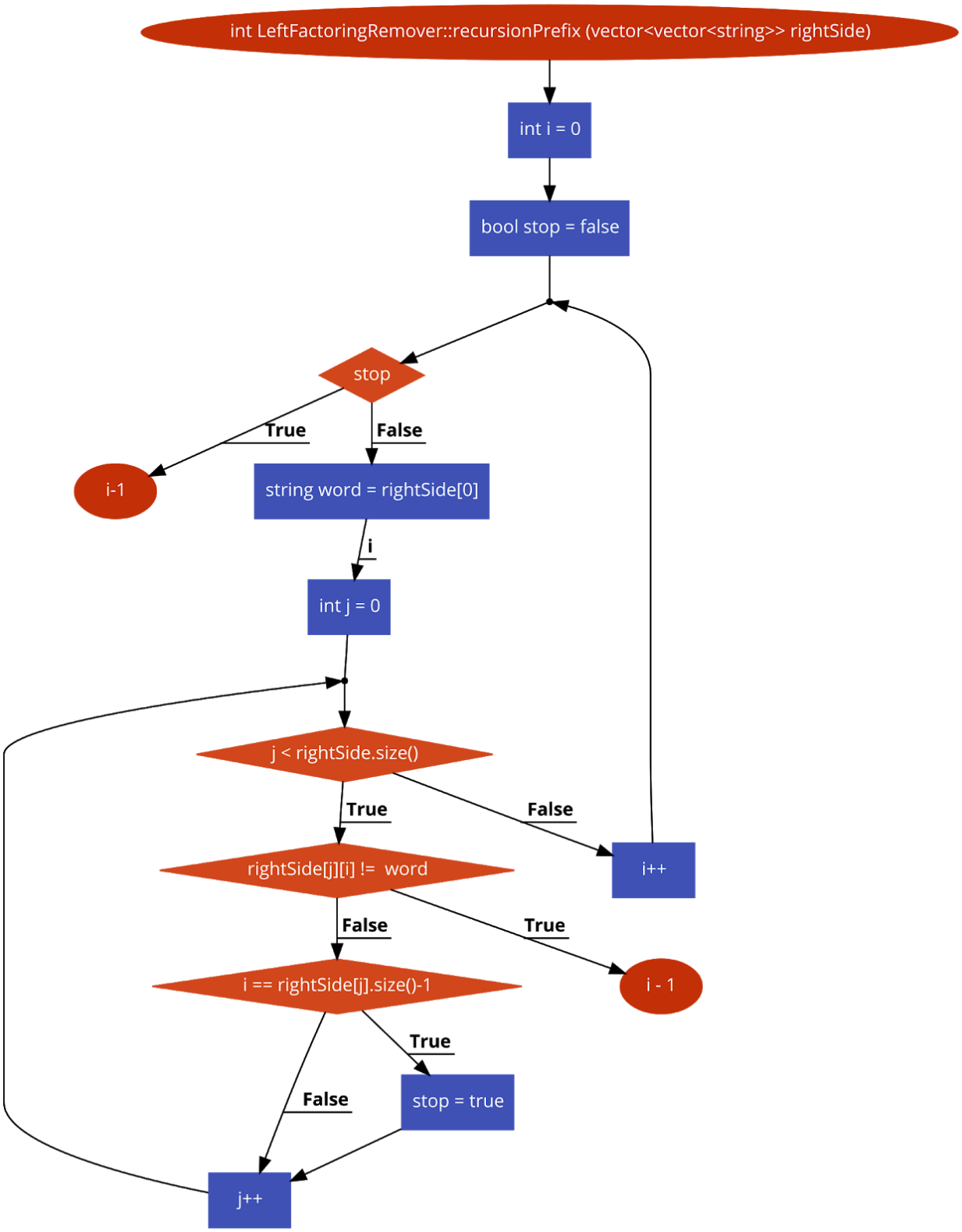
$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$$
$$A' \rightarrow bB \mid B$$
$$\Downarrow$$
$$A \rightarrow aA' \mid cdA''$$
$$A' \rightarrow bB \mid B$$
$$A'' \rightarrow g \mid eB \mid fB$$

Algorithm


void LeftFactoringRemover::leftFactoring(string productionLabel, int id, vector<vector<string>> rightSide)
bool visited
rightSide.size()
int i = 0
i < rightSide.size()
False
True
factored_productions.insert(make_pair(productionLabel, rightSide))
visited[i] = false
int i = 0
i++
True
i < rightSide.size()
True
should create new empty vector
visited[i]
False
vector<vector<string>> groupOfSamePrefix
groupOfSamePrefix.push_back(rightSide[i])
visited[i] = true
int j = i + 1
j < (rightSide.size())
False
True
groupOfSamePrefix.size() > 1
rightSide[i][0] == rightSide[j][0]
True
groupOfSamePrefix.push_back(rightSide[j])
False
visited[j] = true
j++


start recursion and create new production
-> return index from recursion
-> get prefix and add to it new production label
-> create the new vector to production --> BxA'|dz where A' new production
-> remove old production and push modified production to new map
-> remove from the production the two vectors and push back the new vector
-> call left factoring for the new production
True
string newProductionLabel = productionLabel + to_string(id) + "'"
:getInstance()->insetNonTerminal(newProductionLabel)
int indexPrefix = recursionPrefix(groupOfSamePrefix)
vector<string> modifiedVector
int i = 0
i <= indexPrefix
True
False
modifiedVector.push_back(groupOfSamePrefix[0][i])
modifiedVector.push_back(newProductionLabel)
i++
rightSide.push_back(modifiedVector)
removeGroupedVectorsFromRightSide(&rightSide, groupOfSamePrefix)
auto it = factored_productions.find(productionLabel)
it->second = rightSide
int i = 0
i < groupOfSamePrefix.size()
True
False
groupOfSamePrefix[i].erase(groupOfSamePrefix[i].begin(), groupOfSamePrefix[i].begin() + indexPrefix+1)
leftFactoring(newProductionLabel, id + 1, groupOfSamePrefix)
groupOfSamePrefix[i].size() == 0
True
groupOfSamePrefix[i].push_back(
EPSILON
)
False
i++
i++


void LeftFactoringRemover::removeGroupedVectorsFromRightSide (vector<vector<string>> rightSide, vector<vector<string>> groupOfSamePrefix)
int j = 0
int i = 0
i < (*rightSide
False
True
i++)
.size() && j < groupOfSamePrefix.size()
(*rightSide
i
== groupOfSamePrefix[j])
False
groupOfPrefix have elements in the same order in rightside so no need of checking from start again. Like sorted arrays
rightSide).erase((rightSide).begin()+i
i--
j++


int LeftFactoringRemover::recursionPrefix (vector<vector<string>> rightSide)
int i = 0
bool stop = false
stop
True
False
i-1
string word = rightSide[0]
i
int j = 0
j < rightSide.size()
True
False
rightSide[j][i] != word
j++
False
True
i == rightSide[j].size()-1
i - 1
True
False
stop = true
j++

Data-structure

map<string, vector<vector<string>>> non_t_productions: data structure to store the productions after left recursion.

map<string, vector<vector<string>>> factored_productions: data structure to store the result after left factoring grammar.

Main Functions

void leftFactoring(string productionLabel, int id, vector<vector<string>> rightSide): start left factoring for the given production passed in params.

void removeGroupedVectorsFromRightSide (vector<vector<string>>* rightSide, vector<vector<string>> groupOfSamePrefix): removes the group having same prefix from the production data structure

int recursionPrefix (vector<vector<string>> rightSide): return the index of the last word in Prefix

First

Overview

To avoid the need of backtrack, which is really a complex process to implement. There can be easier way to sort out this problem:

- If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

What do we mean by first :

- FIRST(A) is a set of the terminal symbols which occur as first symbols in strings derived from A where A is any string of grammar symbols.
- If A derives to eps, then eps is also in FIRST() .

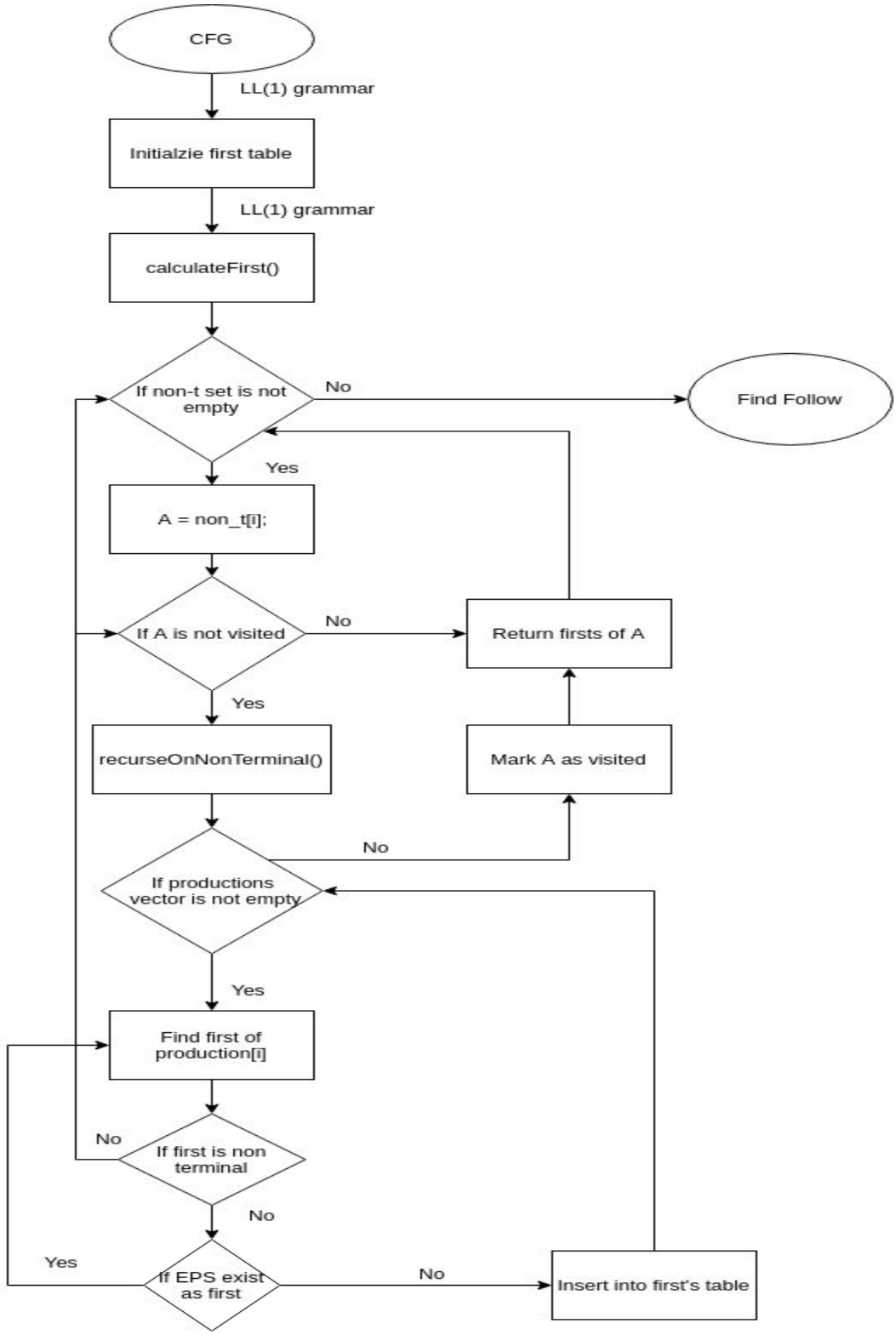
Input

- **map<string, vector<vector<string>>>**
    - Represents each non-terminal and its production(s).
- **set<string>**
    - One to represent the terminals in the CFG.
    - Another to represent the non-terminals.

Output

- **map<string, vector<TableObject*>>**
    - Represents each non-terminal and its first(s) and which production did the first come from.

Algorithm


CFG
LL(1) grammar
Initialzie first table
LL(1) grammar
calculateFirst()
If non-t set is not empty
No
Find Follow
Yes
A = non_t[i];
If A is not visited
No
Return firsts of A
Yes
recurseOnNonTerminal()
Mark A as visited
No
If productions vector is not empty
Yes
Find first of production[i]
No
If first is non terminal
No
Yes
If EPS exist as first
No
Insert into first's table

- Get table of non-terminals and its productions from CFG.
- Loop over the non terminals, and for each non visited one :
    - For each of the non-terminal's productions :
        - Get first of production.
        - Append first into the non-terminal's entry in the first's table.
    - Mark non-terminal as visited.
- Finding the first :
    - If first is terminal 'a' | eps
        - First = First ∪ {'a' | eps}.
    - If first is non-terminal A
        - Recurse to find the firsts of A first.
        - Append first(A) to the entry.
    - If First table contains eps :
        - Recurse of the same production after removing its prev first.

Data-structure

- **set<string> terminals**
    - Set of terminals defined.
- **set<string> non_terminals;**
    - Set of non terminals defined.
- **map<string, vector<vector<string>>> non_t_productions;**
    - LL(1) grammar table.
    - Key is a string which represents a non-terminal.
    - Value is a 2D vector :
        - Outer index indicate a production.
        - Inner index indicate production's terminals and non-terminals.
- **vector<string> visited;**
    - Keep track if non-terminal is already visited or not.
- **map<string, vector<TableObject*>> first;**
    - First's table where:
        - Key is a string representing the non-terminal.
        - Value is a vector of TableObjects, each TableObject contains :

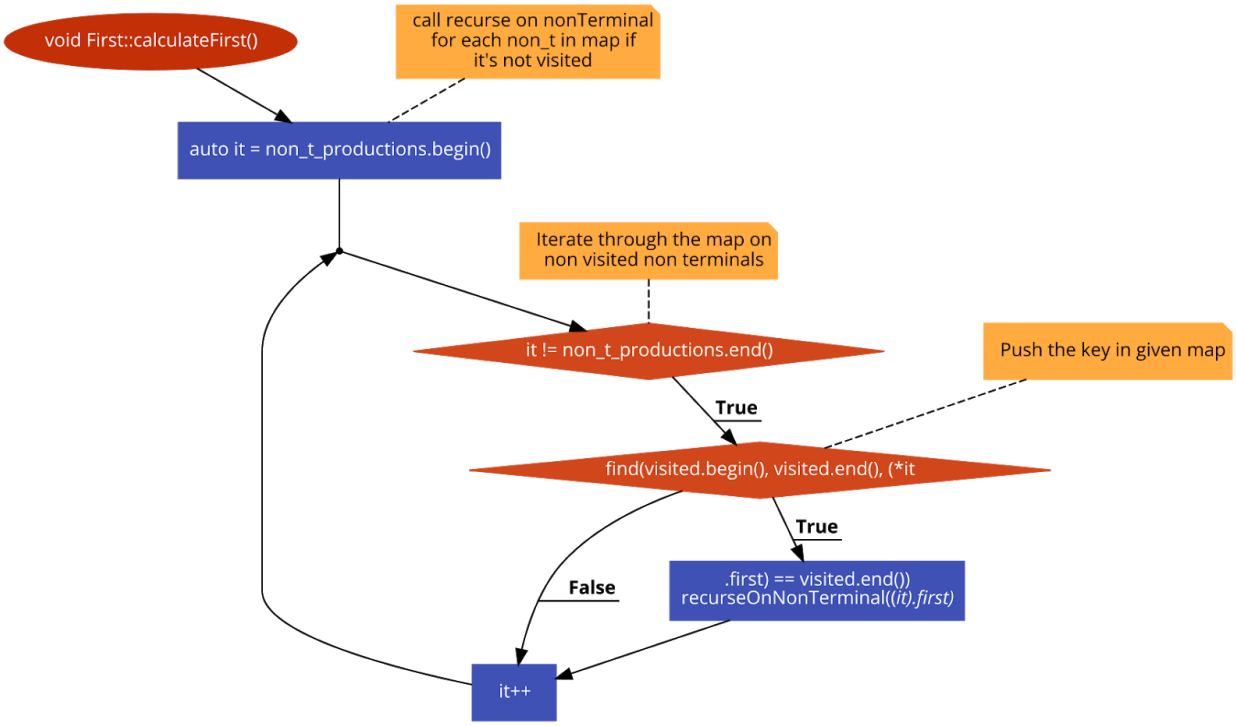
- Index to the production that produced the first's value.
- The first's value.

Main classes

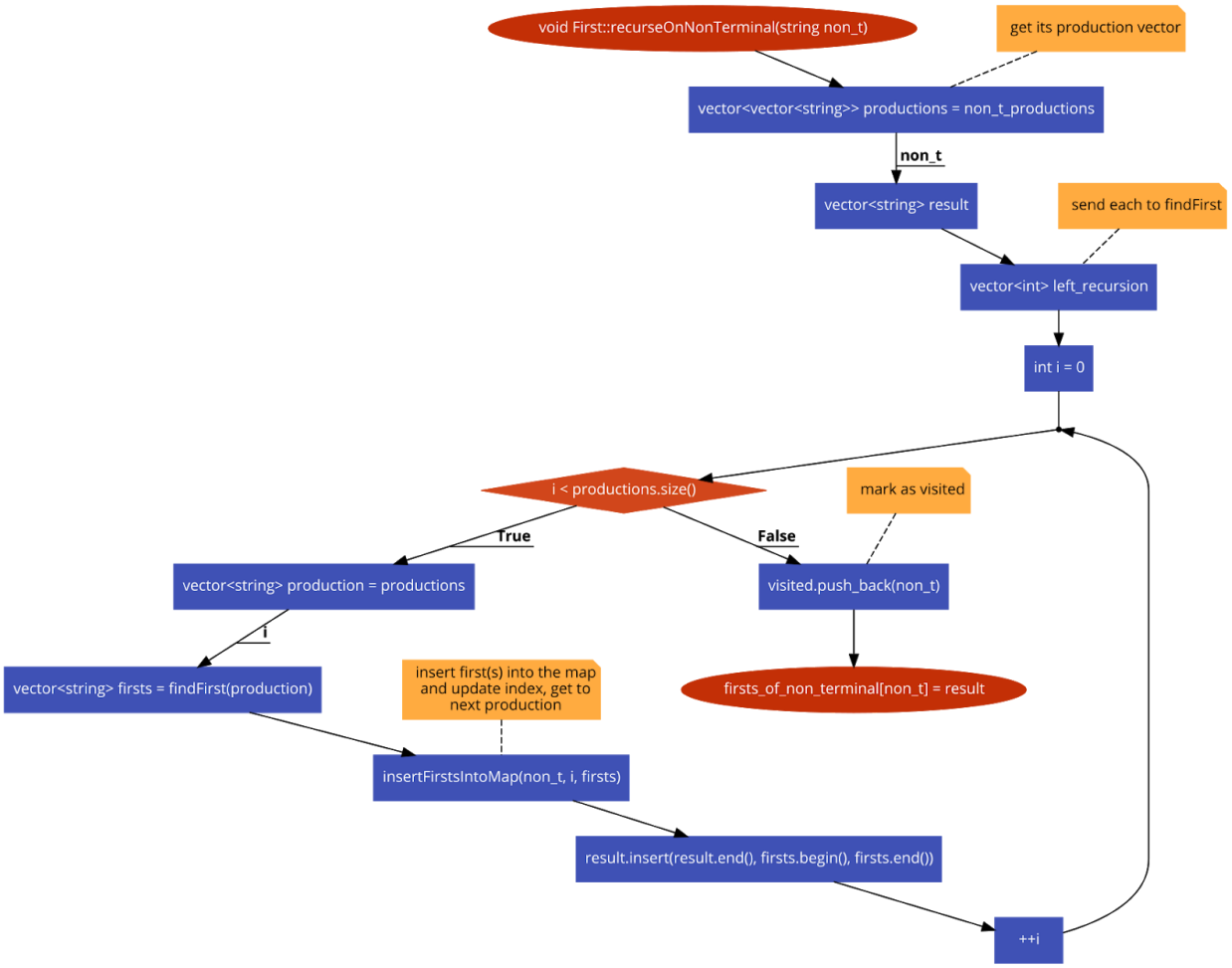
- CFG
  - Interface between parsing and First calculations.
  - Contains the LL(1) grammar.
- TableObject
  - To represent the first in terms of its value and its production.
- Firstandfollow_tables
  - Contains the tables for first and follow of each non-terminals.
  - Acts as an interface between the First and Follow phase and the table parser phase.
- First
  - Factory where firsts are computed for each non terminal.

Main Functions

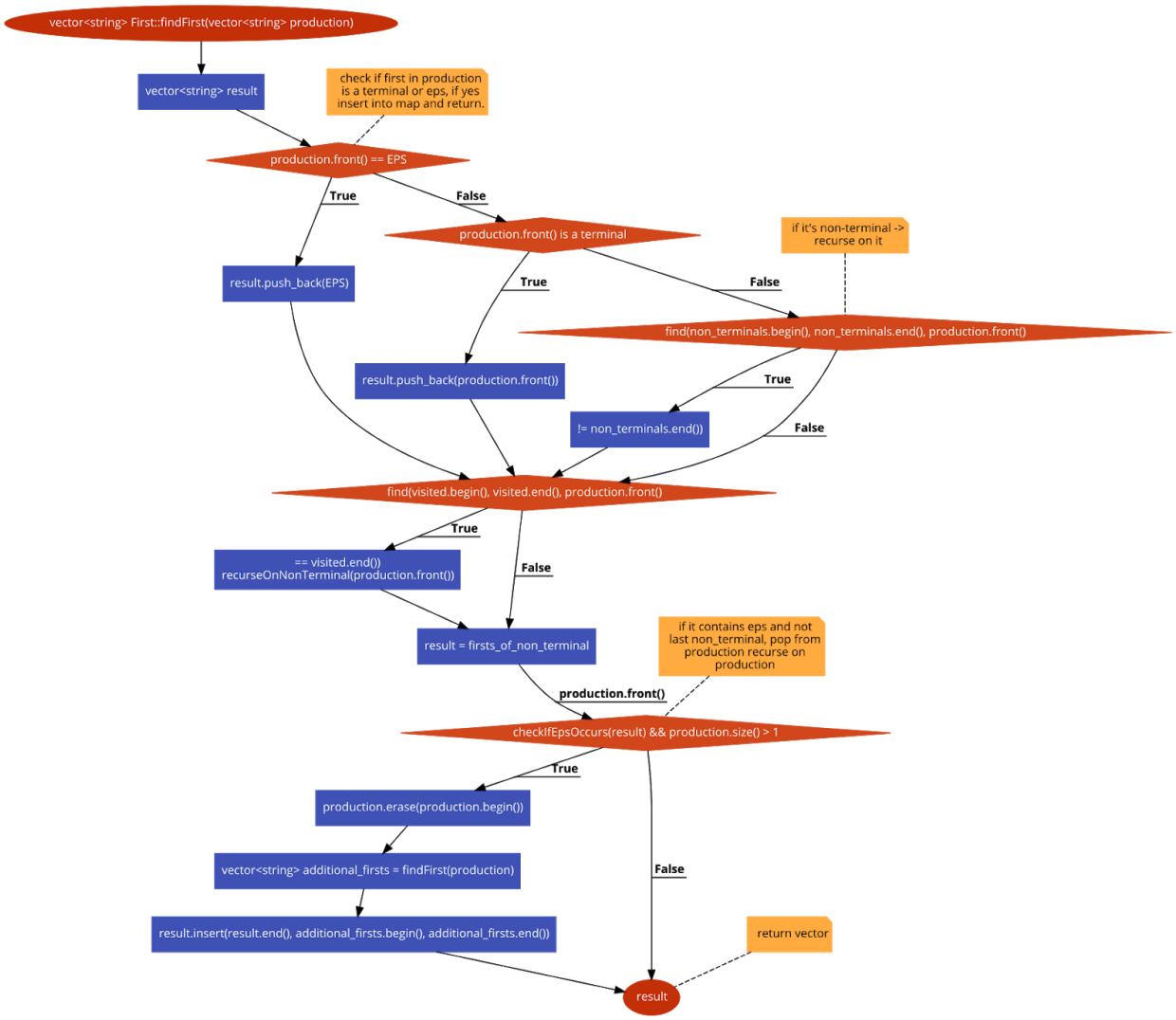
- **void calculateFirst();**
  - Visits all non-terminals.



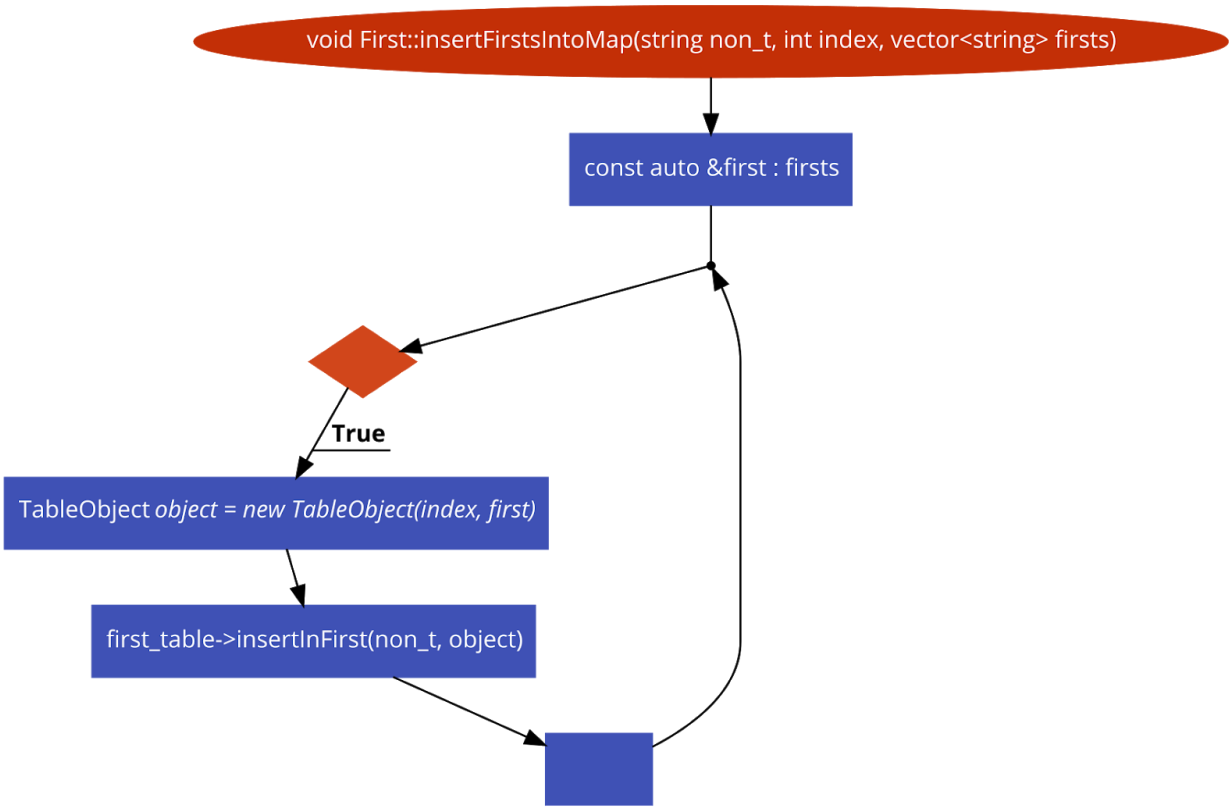
- **void recurseOnNonTerminal(string non_t);**
    - Find firsts for a specific non-terminal.
    - Loops over the non-terminal's productions.



- **vector<string> findFirst(vector<string> prod);**
    - Finds first in a specific production.



- **bool checkIfEpsOccurs(vector<string> firsts);**
    - Check if eps occurs in the first's of a production.
- **void insertFirstsIntoMap(string non_t, int index, vector<string> firsts);**
    - Inserts an entry for the firsts map.
    - For each string in firsts, an object of TableObject 'O' is made, where
        - O.index = index.
        - O.value = firsts[i];



Assumptions

- Epsilon is defined as eps.
- Can handle the grammar even if it's not LL(1).

Alternative designs

- We considered using a forward referencing technique instead of the recursion, where :
    - An extra data-structure would've been used. Whenever a first of a non-terminal depends on another non-terminal
        - An entry is inserted in the data-structure which indicates that whenever that non-terminal's first is computed append it to the first one.

Follow

Overview

- Define FOLLOW(S) for nonterminal S, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form S =>* aAa. There may have been symbols between S and a, at some time during the derivation, but if so, they derive ε and disappeared. A can also be the rightmost symbol in some sentential form, then $ is in FOLLOW(A), where $ is a special "endmarker" symbol that is not to be a symbol of any grammar.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

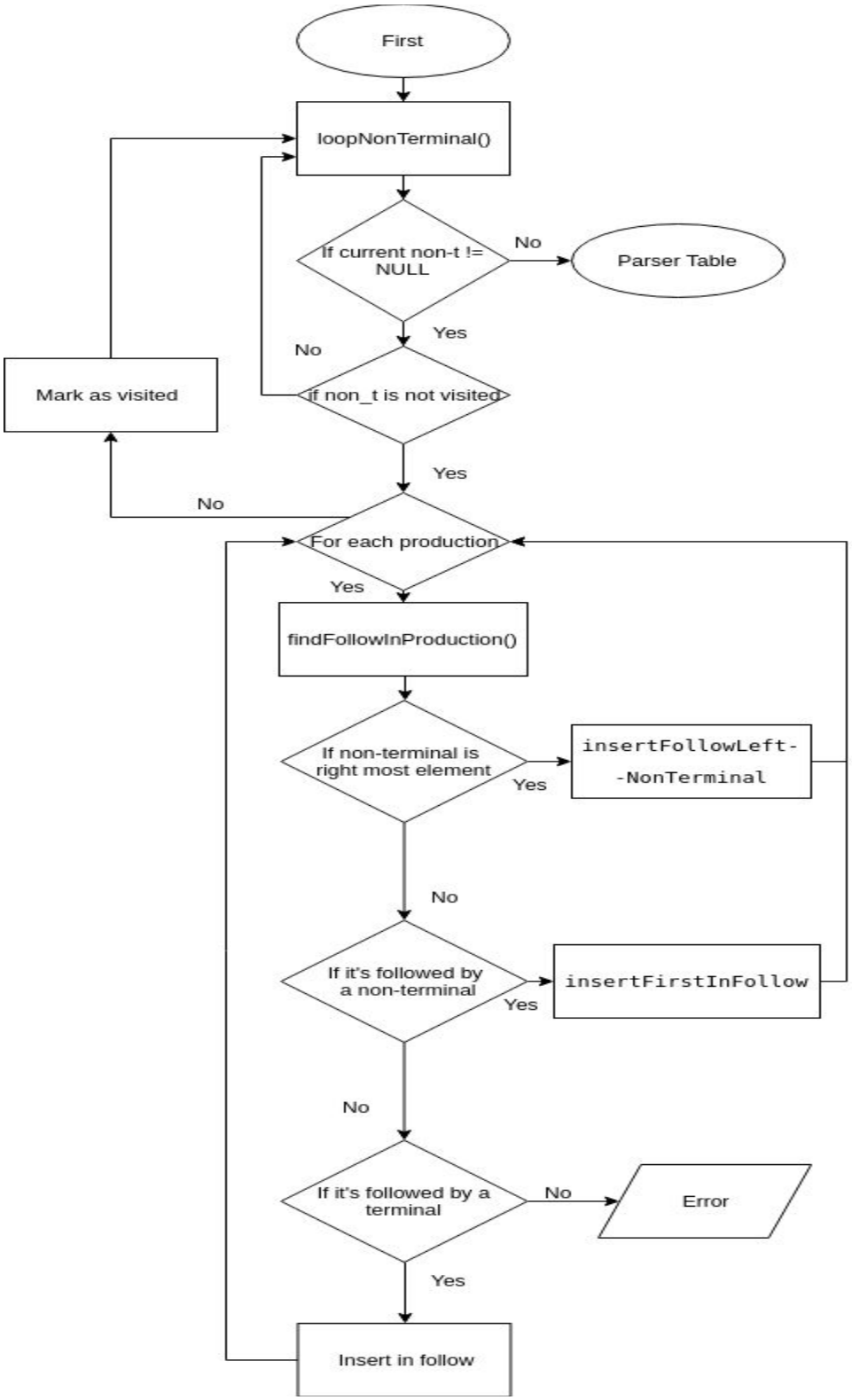
Input

- **map<string, vector<vector<string>>>**
    - Represents each non-terminal and its production(s).
- **set<string>**
    - One to represent the terminals in the CFG.
    - Another to represent the non-terminals.
- **map<string, vector<TableObject*>>**
    - Represents each non-terminal and its first(s) and which production did the first come from.

Output

- **map<string, vector<string>>**
    - Represents each non-terminal and its follow(s).

Algorithm


First
loopNonTerminal()
If current non-t != NULL
No
Parser Table
Yes
No
Mark as visited
if non_t is not visited
Yes
No
For each production
Yes
findFollowInProduction()
If non-terminal is right most element
Yes
insertFollowLeft--NonTerminal
No
If it's followed by a non-terminal
Yes
insertFirstInFollow
No
If it's followed by a terminal
No
Error
Yes
Insert in follow

- Get table of non-terminals and its productions from CFG.
- Loop over the non terminals, and for each A non visited one :
    - For each of the non-terminal B in the table :
        - For each of the non-terminals productions :
            - If A exists in B
                - Get follow of A.
                - Append follow into the A's entry in the follow's table.
- Mark A as visited.
- Finding the Follow :
    - If A is followed by terminal 'a'
        - First = First ∪ {'a'}
    - If A is followed by non-terminal C
        - Append first(C) to follow of A.
        - If an eps is in the first of C
        - Recurse on the same production after removing C.
    - If A is in the right-most hand side:
        - Recurse to find the Follow of B.
        - Append follow of B to A.

Data-structure

- **set<string> non_terminals;**
    - Set of non-terminals defined.
- **map<string, vector<vector<string>>> non_t_productions;**
    - LL(1) grammar table.
    - Key is a string which represents a non-terminal.
    - Value is a 2D vector :
        - Outer index indicate a production.
        - Inner index indicate production's terminals and non-terminals.
- **map<string, vector<string>> follows_of_non_terminal;**
    - Key is non-terminal value.
    - Value is the set of terminals the follows the non-terminal in any production.
- **map<string, vector<TableObject*>> firsts;**

    - First's table where:
        - Key is a string representing the non-terminal.
        - Value is a vector of TableObjects, each TableObject contains :
            - Index to the production that produced the first's value.
            - The first's value.
- **map<string, string> visited;**
    - To mark whether a non-terminal is visited or not.
    - Key is non-terminal.
    - Vlaue is also the non-terminal.

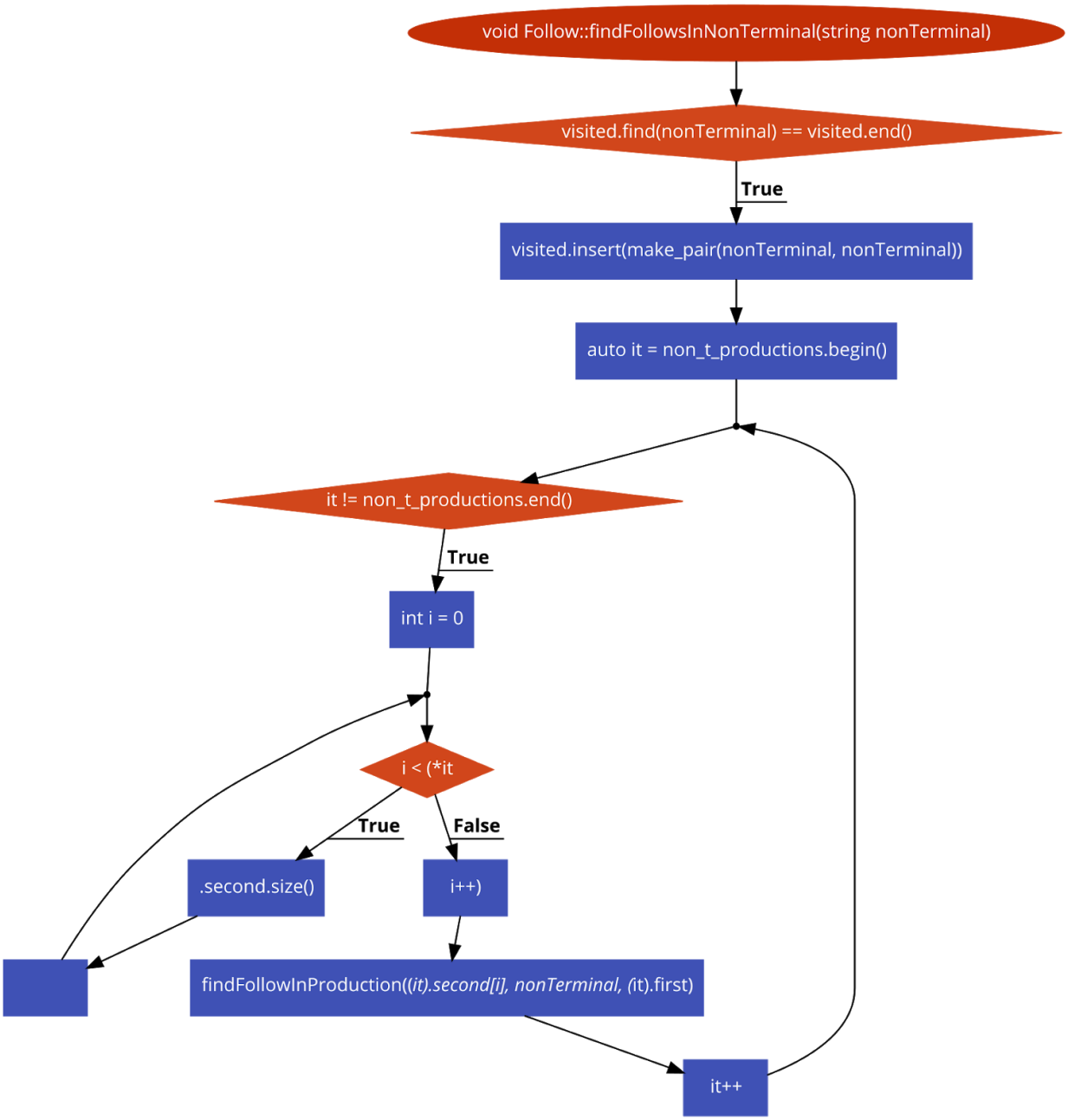
Main classes

- CFG
    - Interface between parsing and Follows calculations.
    - Contains the LL(1) grammar.
- Firstandfollow_tables
    - Contains the tables for first and follow of each non-terminals.
    - Acts as an interface between the First and Follow phase and the table parser phase.
- Follow
    - Factory where follows are computed for each non terminal.

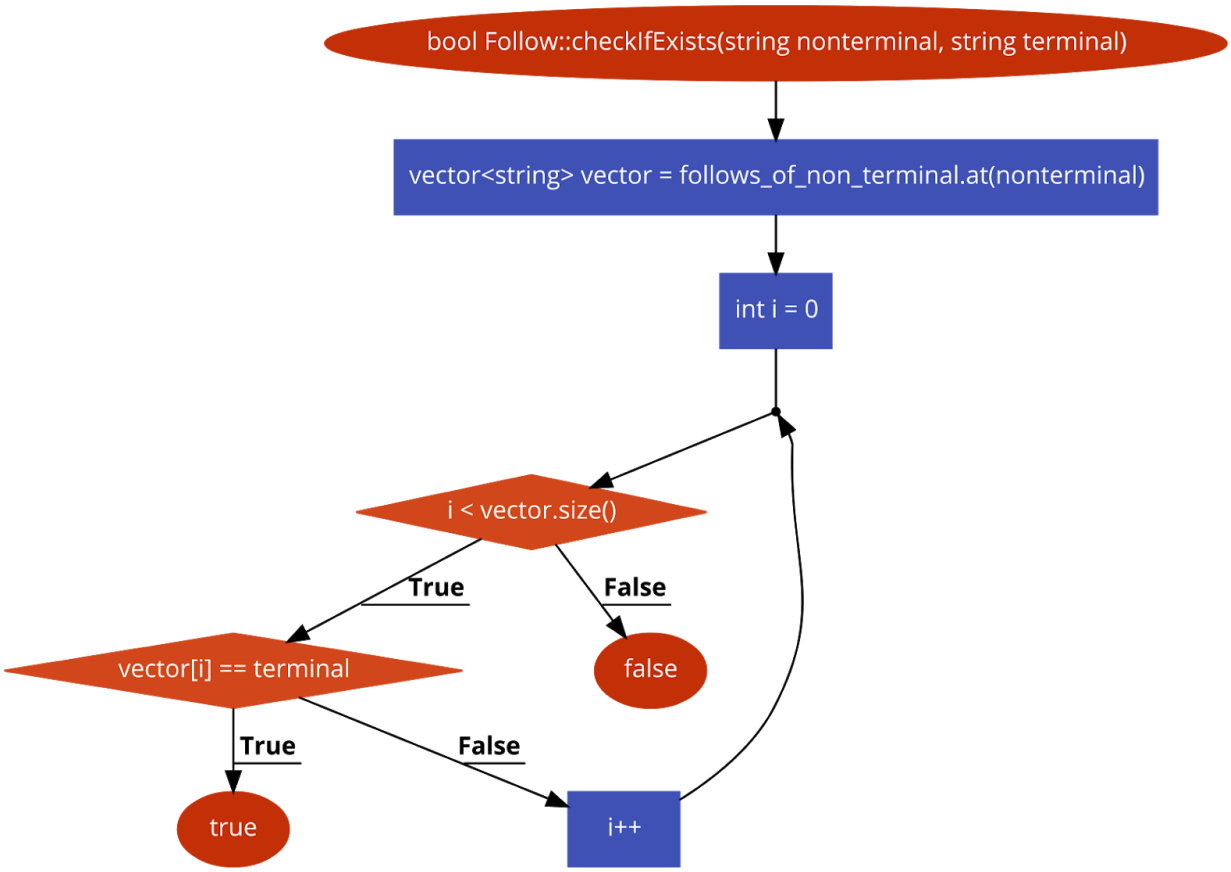
Main Functions

- **void insertFollow (string nonTerminal, string terminal);**
    - Checks if terminal is already found.
    - If not then adds it.

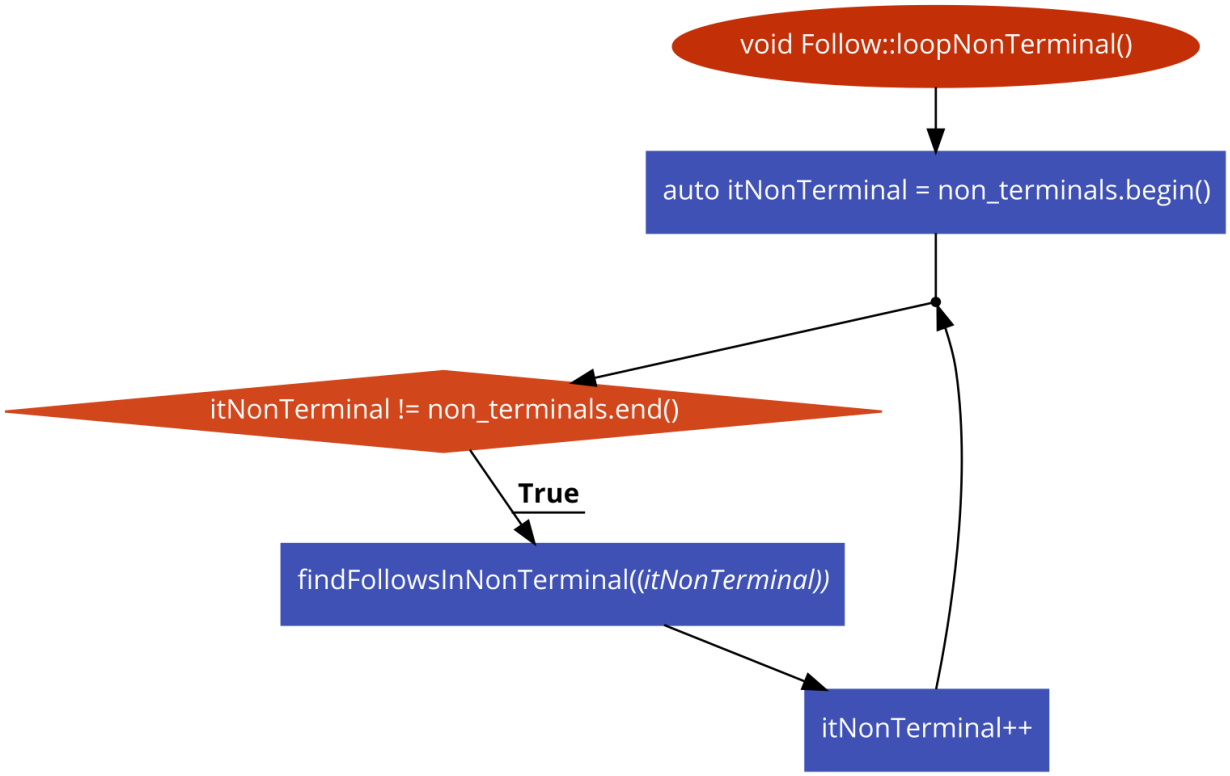
- **void findFollowsInNonTerminal(string nonTerminal);**
    - Finds the follows for a specific non-terminal.



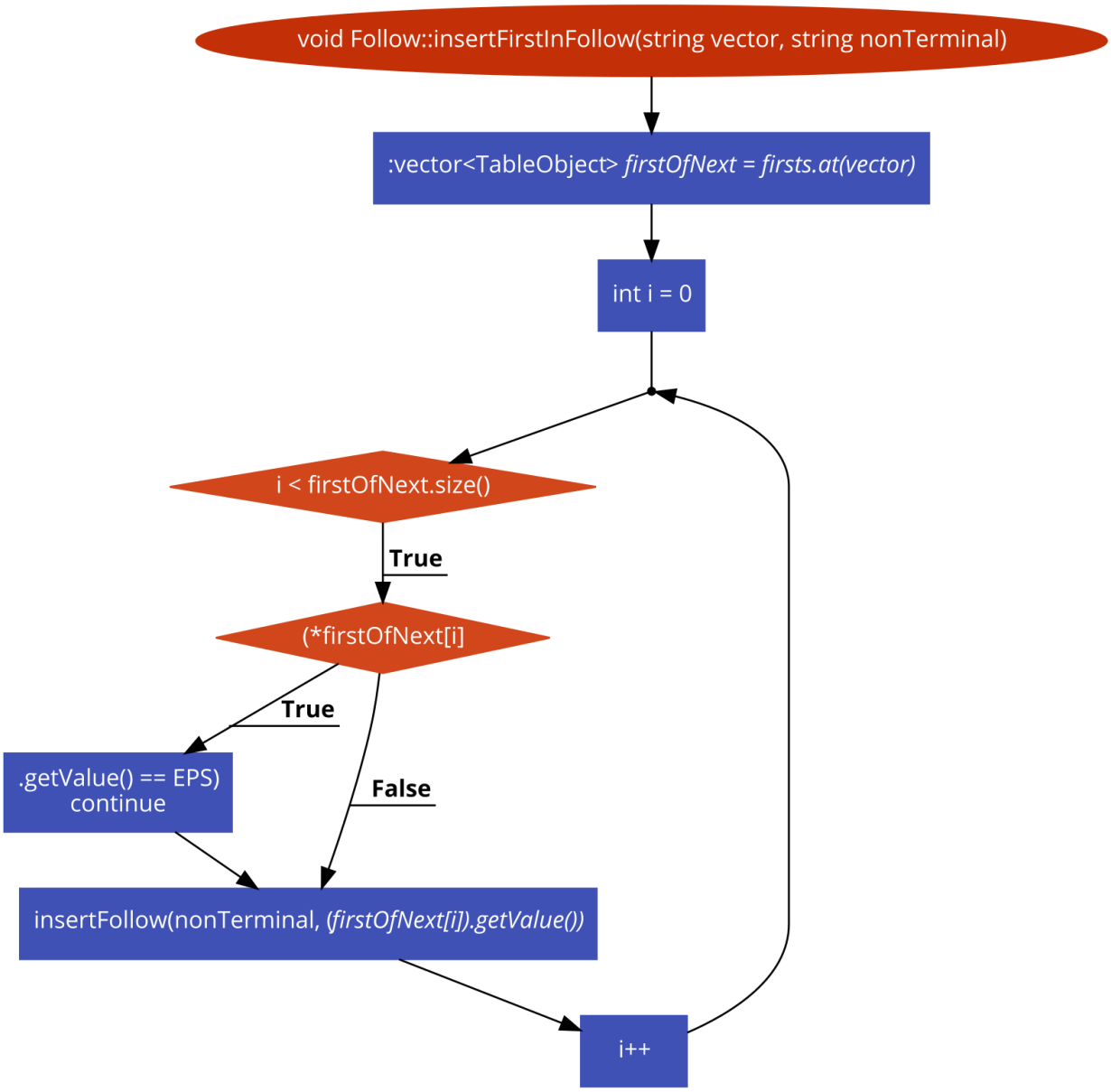
- **void findFollowInProduction(vector<string> &vector, string nonTerminal, string leftNonTerminal);**
    - Finds follow in a specific production of the non-terminal.
- **bool checkIfExists(string nonterminal, string terminal);**
    - Checks if terminal already exists.



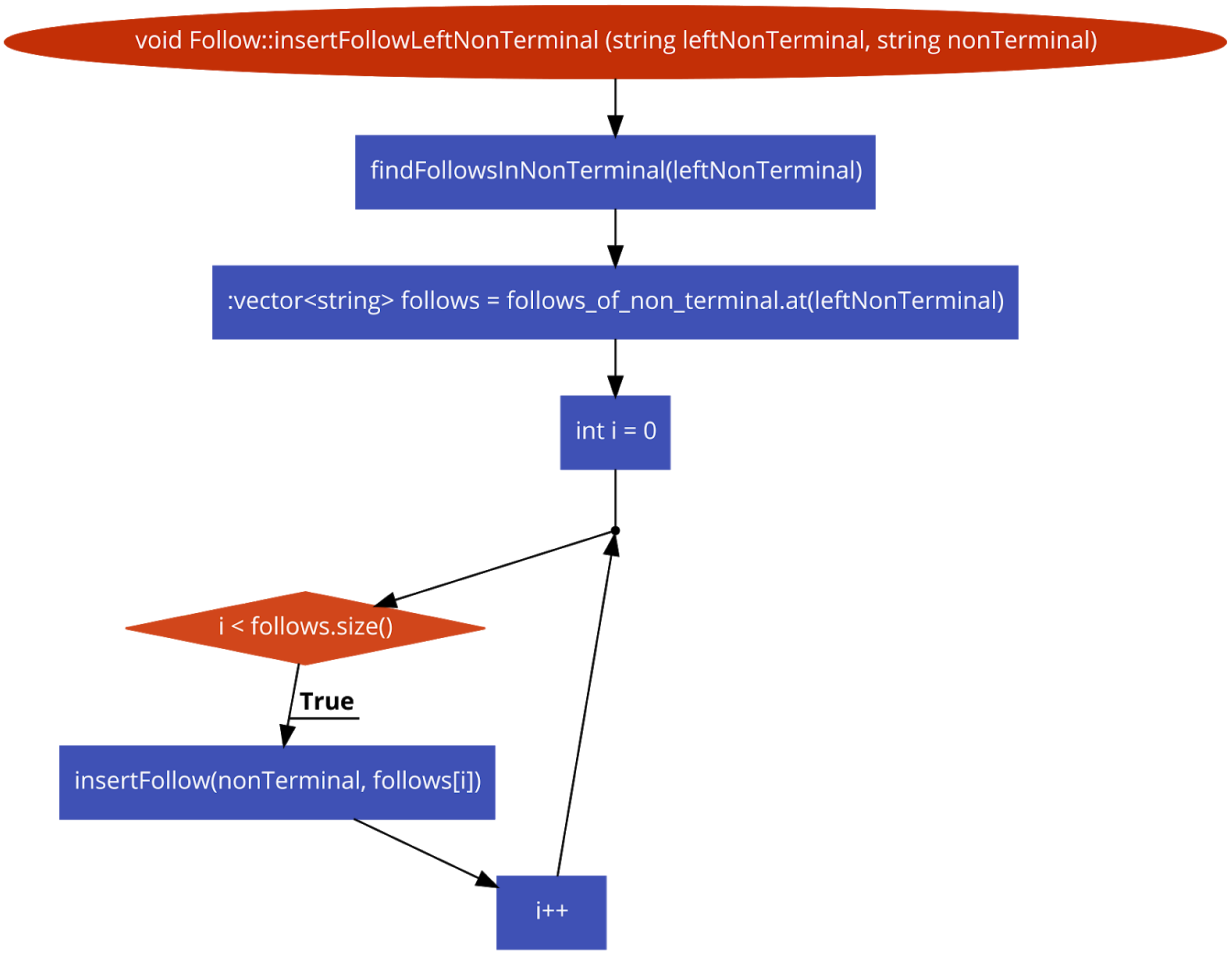
- **void insertDollarSign();**
  - Inserts dollar sign to the start state follow set.
- **void loopNonTerminal();**
  - Loop on each non-terminal to find its follows.



- **void insertFirstInFollow(string vector, string nonTerminal);**
    - If a first of a non-terminal is the follow of another.



- **void insertFollowLeftNonTerminal (string leftNonTerminal, string nonTerminal);**
    - If the follow of a non-terminal is the follow of the left hand-side non-terminal.



- **bool containsEps(string nonTerminal);**
    - Checks if firsts contains epsilon.

Assumptions

- Epsilon is defined as eps.
- Can handle the grammar even if it's not LL(1).

Alternative designs

- We considered using a forward referencing technique instead of the recursion, where :
    - An extra data-structure would've been used. Whenever a first of a non-terminal depends on another non-terminal whether first of follow

- An entry is inserted in the data-structure which indicates that whenever that non-terminal's first or follow is computed append it to the first one.

Parser Table

Overview

Construct a table of non terminals and terminals using two functions First and Follow .

During the construction of the table we can know if there is ambiguity in the grammar or not.

Input

- **map<string, vector<TableObject*>>**
  - Represents First of each non terminal.
- **map<string, vector<string>>**
  - Represents Follow of each non terminal.
- **set<string>**
  - Represents each of terminals and non terminals.

Output

- **map<string,map<string,int>>**
    - Represents the parsing table.

Algorithm

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,\$]

- All other undefined entries of the parsing table are error entries.

- First, make all the entries of the table Error.
- Loop on the first of each non terminal and put each production of the first in the table [non terminal] [terminal]
- If the first contains EPSILON loop on the follow and put EPSILON in the table [non terminal] [terminal]
- Else put SYNCH in each position that not contains a production.

Data-structure

- **map<string, vector<TableObject*>> first**
    - The key is the non terminal
    - The table object is an object consists of a string of the terminal and an integer refers to the number of the production of the non terminal.

- **map<string, vector<string>> follow**
    - The key is the non terminal
    - The vector<string> contains the follow of the non terminal.

Main classes

- ParsingTable

Main Functions

- Void createParsingTable()
- Void printError()
- map<string,map<string,int>> getParsingTable();

Assumptions

- EPS is defined as “\\L”
- ERROR is defined -1
- SYNCH is defined -2

Alternative designs

No alternative design

Output

Overview

Read the input file program and get its tokens and trace the tokens with the input grammar using the parsing table and uses Panic-Mode Error Recovery :

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - 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.

Input

- **map<string, vector<vector<string>>>**
  - Represents the productions of the grammar.
- **map<string,map<string,int>>**
  - Represents the parsing table.

Output

- Output file contains the content of the parsing stack in each loop.

Algorithm

- First put the dollar sign $ in the parsing stack and the start state of the grammar
- Loop until the stack is empty
- Get a token and check if it is a terminal or not
  - If not terminal discard this token
- Check if the top of the stack is terminal or non terminal

    - If terminal : check if it matches the token
        - If not matching print error missing the top of the stack then pop the top of the stack and continue with the current token.
        - If matching print match and pop the top of the stack and get the next token.
    - If non terminal : get the production from the parsing table and push it from back to the parsing stack and continue with the current token.
- Merging with Lexical
    - A struct state is introduced, which contains :
        - Int vec_ind
            - Index for the portion of the line we're reading.
        - Int ch_ind
            - Index for a character within the portion.
        - string token
            - Value of the matched token.
    - A function getNextToken(state s) is also introduced :
        - Takes s.vec_ind as the index to the portion of the line.
        - Takes s.ch_ind as the index to a character within the portion of the line.
        - And returns a new state where
            - s.token = matched_token
            - s.vec_ind = index of the portion to be matched next.
            - s.ch_ind = index of the character within the portion to start with next.

Data-structure

- **stack<string> parsingStack**

Main classes

- Output

Main Functions

- **void initialization()**
    - Initialize the parsing stack and push the dollar sign $ and the start state.
- **void readFile(string fileName)**
    - Read the input file program and split each line by spaces then send each word to the lexical to get the token then trace the token with the grammar.
- **bool isTerminal(string symbol)**
    - Take an input string and check if it exists in the terminal map.
- **void error(string top, string token, int errorNum)**
    - Print the necessary error.
- **void printStack()**
    - Print all the content of the parsing stack.

Assumptions

No assumptions

Alternative designs

Take the vector of tokens after the lexical ends then loop on each token and trace with the grammar

---

Additional

- TestFirstAndFollow
    - Is a tester interface for the first and follow construction.

Design Patterns

- Singleton
    - FirstandFollow_tables
    - CFG
    - ParserTable

- Factory
    - CFGparser
    - First
    - Follow
    - Output
- Builder
    - ReadProgram in Lexical analyser.
- Facade
    - Firstandfollow_tables
    - CFG
    - ParserTable
    -

Problems faced

- Finding an optimized algorithm to remove left recursion and for left-factoring.
- Merging lexical with parser.

What's Next

We need to perform semantic analysis and intermediate code generation during parsing – so we have one pass front-end compiler.

Tasks distribution

- Aya Ashraf
    - Parser, Left recursion, Left Factoring, report, testing.
- Rowan Adel
    - Parser table, Output, testing, report.
- Sarah Ahmed
    - Left factoring, Follow, testing, report
- Sohayla Mohammed

- First, testing, report.

References

https://www.gatevidyalay.com/left-factoring-examples-compiler-design/

https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/

https://web.cs.wpi.edu/~kal/PLT/PLT4.1.2.html