Programming languages & Compiler

Java compiler

Phase 1 : Lexical Analyser

Team members

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

Team members **1**

Introduction **4**

Overview 4

Problem statement 4

Background 4

Limitations 5

Implementation 6

Overall design 6

Structure of Automata 6

Structure 6

Why 7

Alternative designs 7

Phases 7

Design phases 8

Language Parser 8

Overview 8

Input 8

Output 8

Algorithm 9

Data-structure 10

Main classes 10
Main Functions 10
Assumptions 11
Alternative designs 11
Creating NFA 12
Overview 12
Input 12
Output 12
Algorithm 12
Data-structure 16
Main classes 17
Main Functions 17
Assumptions 22
Alternative designs 22
Deriving DFA 23
Overview 23
Input 23
Output 23
Algorithm 24
Data-structure 25
Functions 25
Minimizing DFA 26
Overview 26
Input 27
Output 27
Algorithm 28
Data-structure 28
Functions 29
Assumptions 36
Alternative designs 37
Parse program and output 37
Overview 37
Input 37
Output 37
Algorithm 37
Data-structure 38
Functions 38
Assumptions 43
Alternative designs 43

Design Patterns **44**

Problems faced **44**

Sample runs **45**

What's Next **49**

Tasks distribution **49**

References **50**

Introduction

Overview

This report will cover the structure and design we used to implement an automatic lexical analyzer generator tool.

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

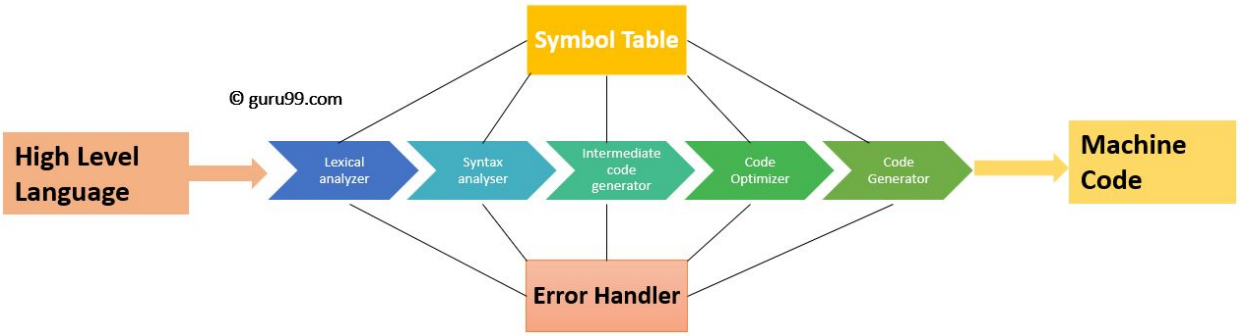
Design and implement a lexical analyzer generator tool using C++. The lexical analyzer generator is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens.

The tool should follow the phases studied in class :

- Construct NFA
- Derive DFA
- Minimize DFA

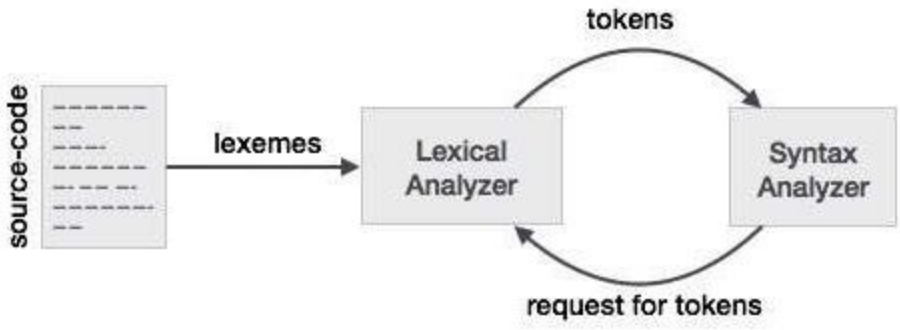
Background

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 one is Lexical Analyser

- It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- If the lexical analyzer finds a token invalid, it generates an error.



Limitations

- In generalizing the generator
    - The wide-span of allowed regexes and characters sequences made it harder for the parser to include all conditions.
- Timing limitations
    - Given more time the code could've been more optimized.

Implementation

Overall design

Structure of Automata

Structure

- The automata is considered as an instance of **Graph.**
- The states are **Node,** transitions are **Edges** from a source **Node** to a destination **Node.**
- Each Graph contains :
    - vector<Node*>
        - Contains the states in the Graph.
    - vector<Edge*>
        - Contains all edges in graph.
    - Node* start
        - Pointer to the start node.
    - Node* end
        - Used during the construction of the NFA to check for acceptance state for each subgraph.
- Each Node contains:
    - Int Id
        - Number that distinguish the states from each other.
    - String status
        - By default it's "n" indicating not acceptance state.
        - If it's acceptance state then it holds a string of what it accepts (e.g id etc).
    - Int priority
        - Indicates the priority of the acceptance state.
        - By default it's -1; each non acceptance state has prioriy of -1.
        - If it's a keyword then 0 is used to indicate highest possible prioirty, otherwise it depends on the order in input language file.

    - vector<Edge*> inward
        - for all inward edges(current node is destination for).
    - vector<Edge*> outward
        - for all outward edges (current node is source for).
- Each Edge contains :
    - Definition* weight
        - A definition that represents the weight.
        - Node* src
            - Source node.
        - Node* dst
            - Destination node.
- Each Definition :
    - A graph of its own.

Why

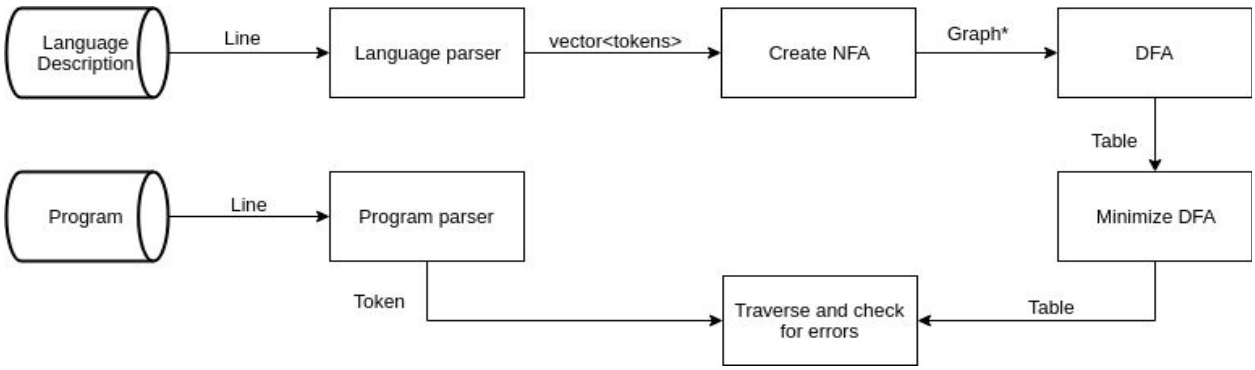
- How our mind visualizes the automata.
- It might not be the most optimized but the simplest.

Alternative designs

- Our initial thought was using Tri-node data structure
    - //elaborate
- We thought about a tree, but it wouldn't have worked

Phases

We divided the lexical analyser into 6 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

Language Parser

Overview

In this phase, input file for the language description is read line by line.

Each line is tokenized into a vector<string> which is sent to the next phase “NFA”

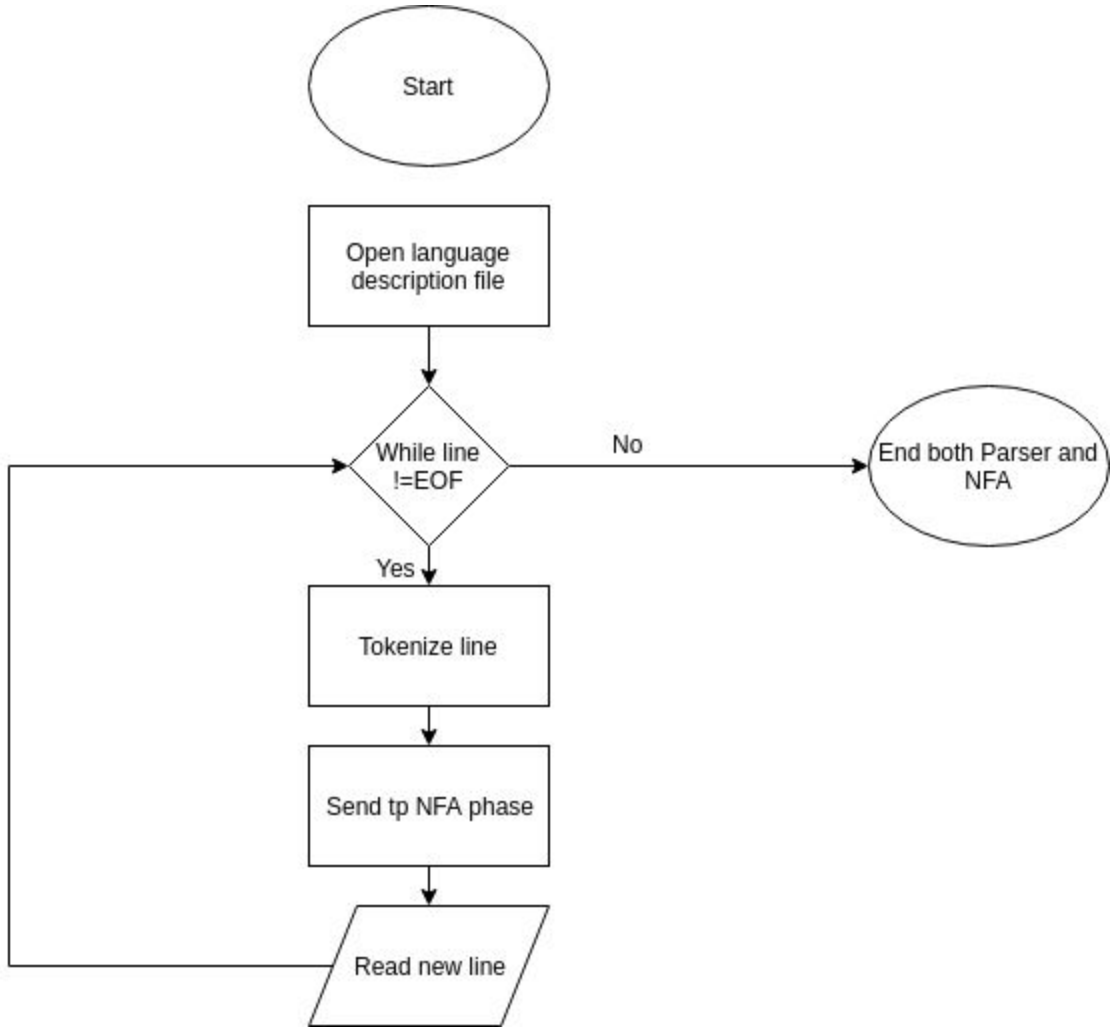
Input

- Language Description file.

Output

- Tokenized line (vector<string>).

Algorithm



- Each line is read by the file reader.
- For each line read :
    - Tokenize the line into a vector of string.
    - The vector is sent to NFA phase to construct one of the subgraphs that will eventually be part of of the NFA graph.
- Continue.
- Tokenization
    - First split by spaces then each token is traversed to check if it needs to be further split.

Data-structure

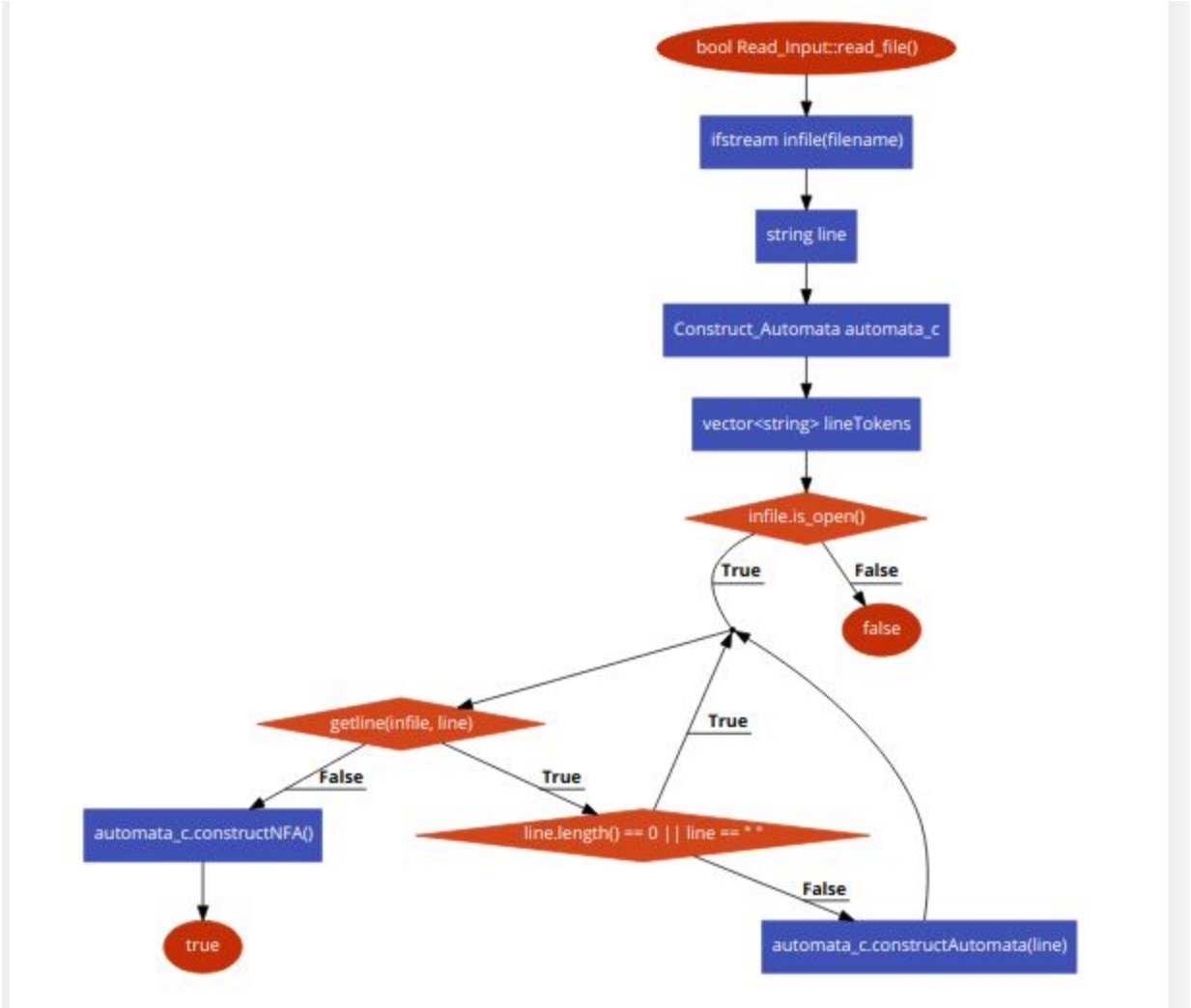
- vector<string>
    - For tokens.

Main classes

- Read_Input
    - Reading of input file.
- Tokenizing
    - Tokenizer for lines.

Main Functions

- read_file()
    - Reads input file and sends each line read to NFA.



- parseString()
    - Tokenizes each line.

Assumptions

- Any reserved symbol needed to be used within the language, is preceded by an escape backslash character.
- Only the following symbols are used in regular definitions and regular expressions : - | + * ( ).

Alternative designs

- Considered only splitting by spaces, didn't work out.

- Considered using regex
    - Time complexity would've been much higher.

Creating NFA

Overview

NFA

A **nondeterministic finite automaton** (**NFA**), or nondeterministic finite state machine, does not need to obey The following restrictions

- each of its transitions is *uniquely* determined by its source state and input symbol, and
- reading an input symbol is required for each state transition.

Thompson's construction Thompson's constructionThompson's constructionThompson's constructionThompson's construction is an algorithm for compiling a regular expression to an NFA that can efficiently perform pattern matching on strings.

In this phase, NFA graph is constructed.

Each line sent from Parser is made into a subgraph, at the end all the subgraphs are ORed together.

This output of this phase is sent to the DFA phase to perform a powerset construction algorithm.

Input

vector<string> representing the line tokens -> from Parser phase.

Output

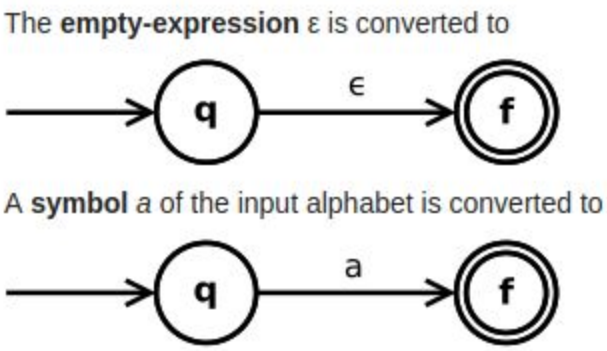
NFA graph.

Algorithm

- The algorithm works recursively by splitting each token in vector into its constituent subexpressions, from which the NFA will be constructed using a set of rules. More precisely,

from a regular expression E, the obtained automaton A with the transition function δ respects the following properties:

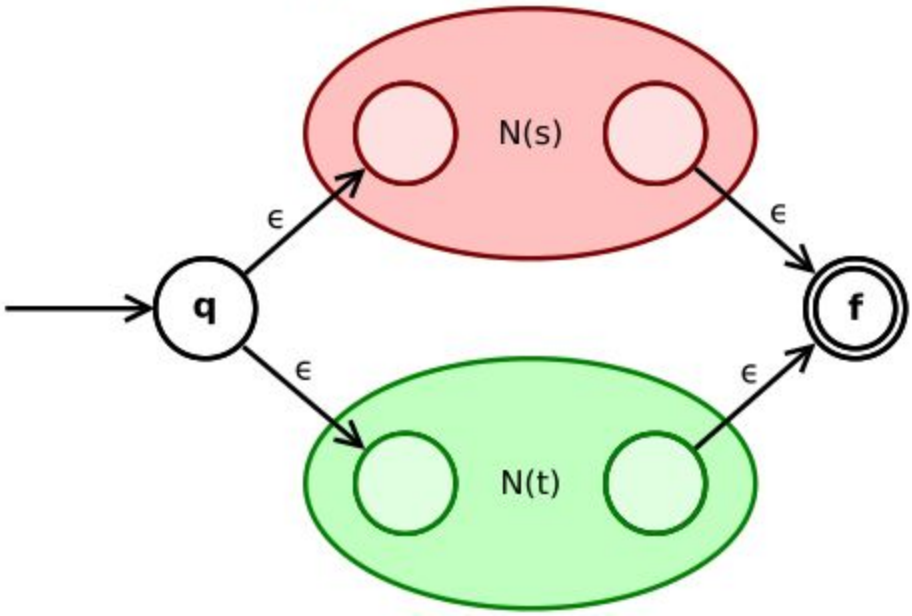
- A has exactly one initial state q0, which is not accessible from any other state. That is, for any state q and any letter a, $\delta(q, a)$ does not contain q_0.
    - A has exactly one final state q_f, which is not co-accessible from any other state. That is, for any letter a, $\delta(qf, a) = \phi$.
- At the start, we check if the line sent is a definition or a regular expression.
- For a definition :
    - Its expression is split into it's most simple form then an automata is created.
    - A map<string, Definition*> is used to save the created graphs for the definition, key is the LHS of the expression (e.g digit, letter, (, ;, ...etc), which is mapped to the already created automata, as to use it again for any future uses of the same definition.
- For an expression :
    - A subgraph of the NFA is created and added to a vector<string>.
    - This vector will later be used to construct the full NFA automata.
- For keyword/Punctuation
    - Same as expression.
- Creating a Graph
    - Recursively we split the given token over (| * + .).
        - Forward Tracking
            - Create graph for each sub-token generated.



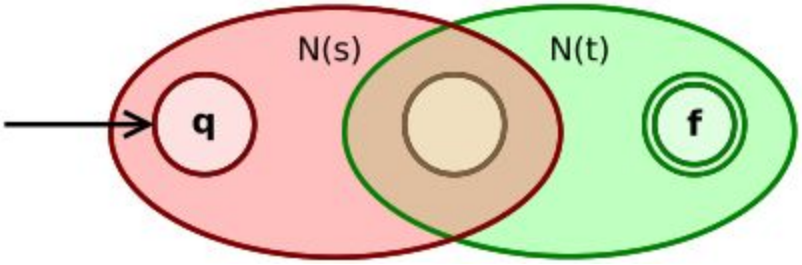
        - Backtracking
            - Merge each two consecutive graphs into one, according to the (| * + .) between them.

- Rules :

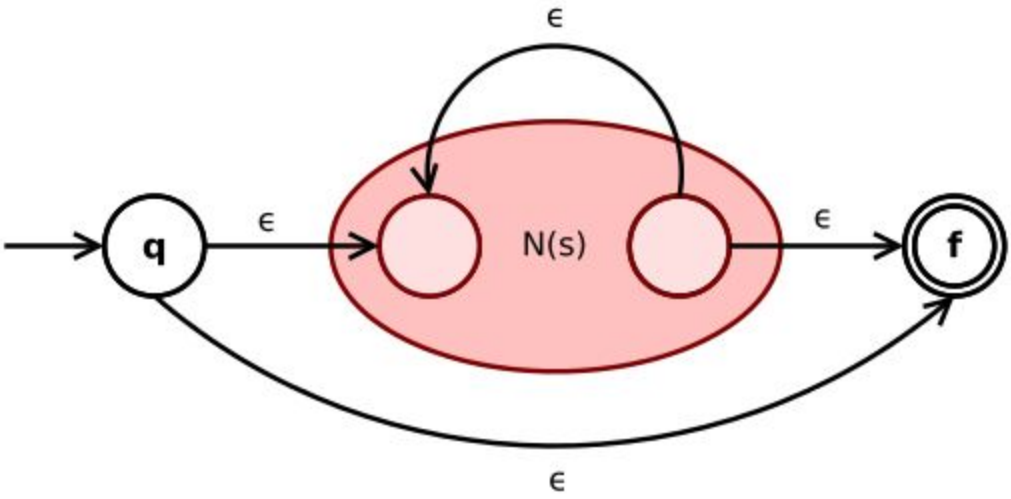
The **union expression** *s*|*t* is converted to



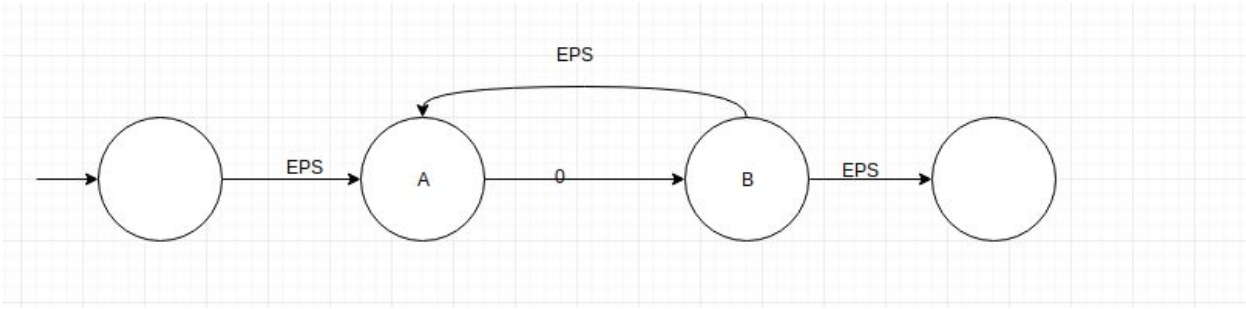
The **concatenation expression** *st* is converted to



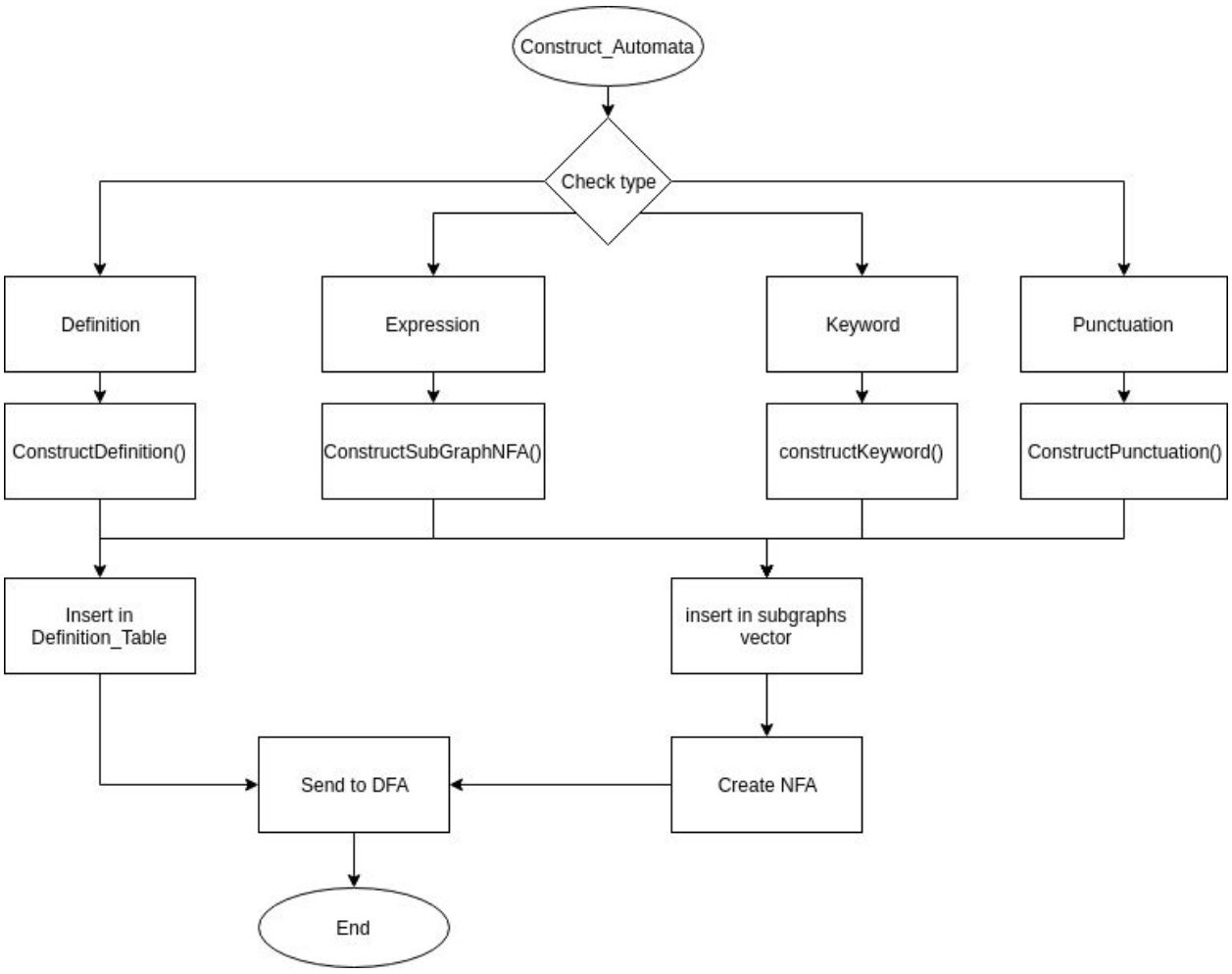
The **Kleene star expression** *s** is converted to



Positive closure of L



- Priority
  - + / *
  - .
  - |
- Brackets are specially handled as a separate creation of automata.
- For each definition that is not in the definitions table, add it to the definitions table.



Data-structure

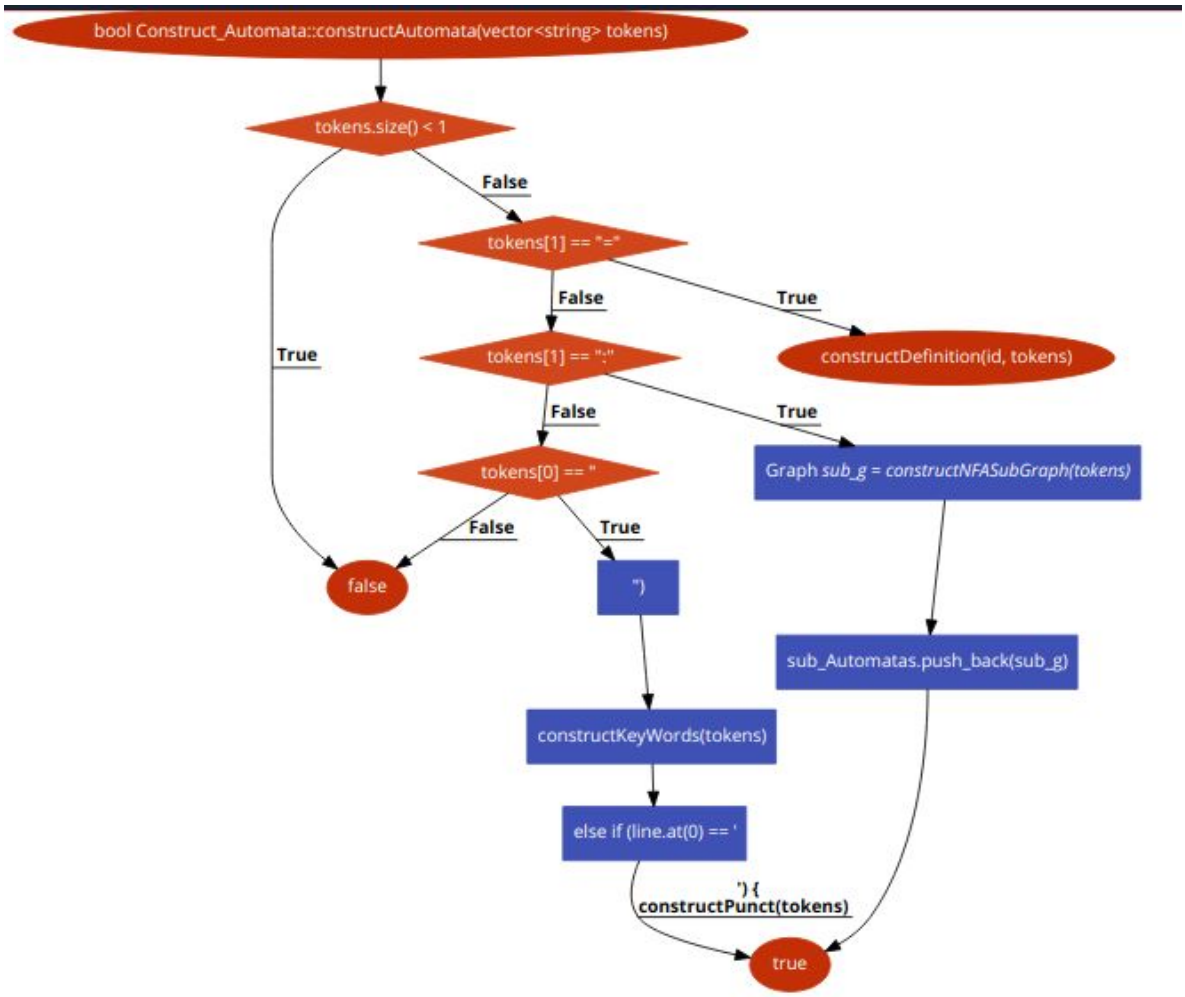
- vector<string>
    - One holds the tokens of the expression/definition.
    - Another holds helpers that'll be used in the merging process (| + * . ...etc).
- vectro<Graph*>
    - To hold the subgraphs created to be merged later.
- vector<Node*>
    - From Node class to hold the states in a graph.
- vector<Graph*>
    - From Node class to hold the edges in a graph.
- map<string, Definition*>
    - The definitions table, each key string maps to a Definition pointer.

Main classes

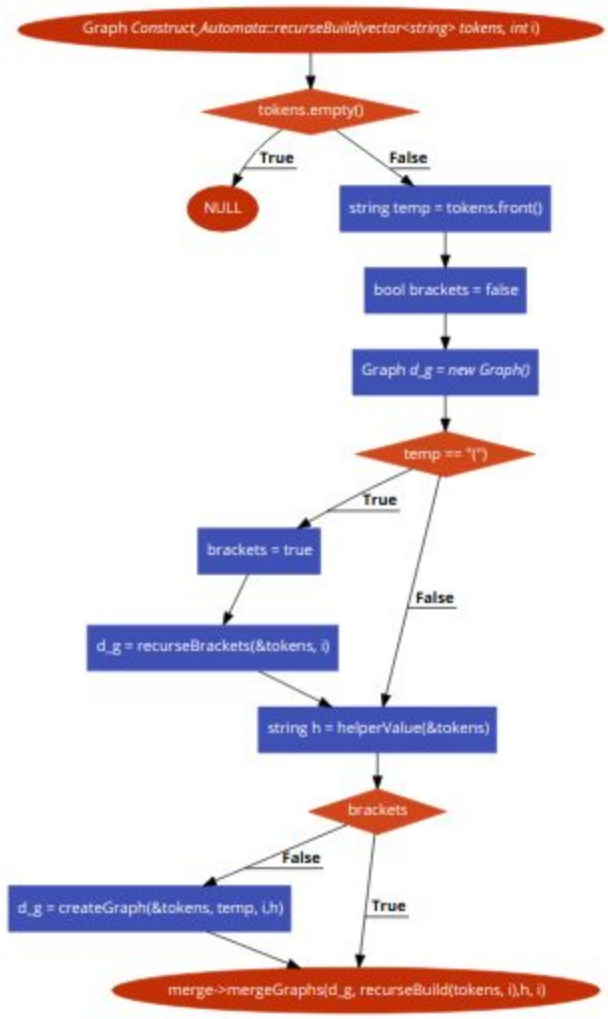
- Construct_Automata
    - Responsible for constructing the automatas.
    - Each tokens vector is sent to this class to be further manipulated and take action accordingly.
- NFA
    - Holds the NFA automata generated by Construct_Automata.
- Helpers
    - Holds the helpers mentioned above (| * + ...).
    - Responsible for merging two graphs based on a “helper” given.
    - The merging steps is mentioned above.
- Definitions_Table
    - Holds the definitions used in the NFA.
- Definition
    - Weight of any transition in the graph.

Main Functions

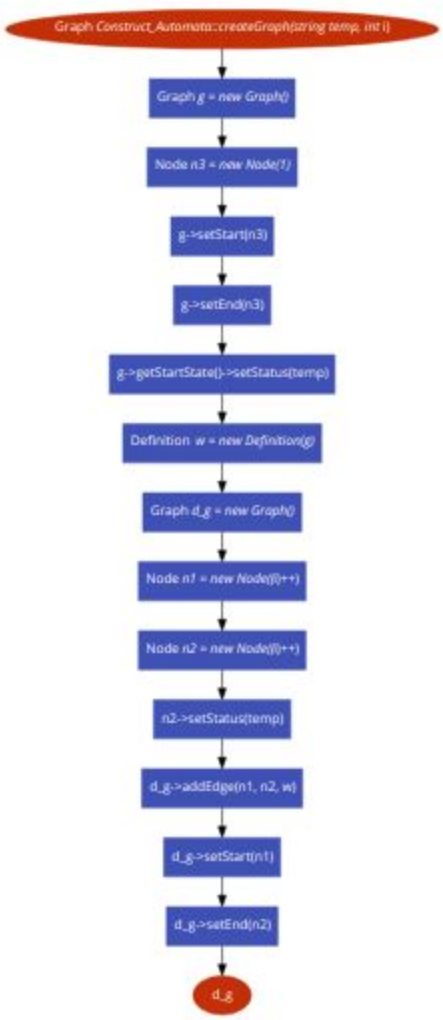
- In Construct_Automata :
    - bool constructAutomata();
        - Chooses type of graph to be made
            - Subgraph for NFA.
                - Graph* constructNFASubGraph(vector<string> tokens);
            - Definition.
                - bool constructDefinition(string id, vector<string> definition);
            - Keyword..etc
                - void constructKeyWords(vector<string> tokens);
                - void constructPunct(vector<string> tokens);



- bool constructNFA();
  - Construct full NFA from subgraphs created by adding a new start Node and transitions from the new start to all the start states of the subgraphs.
- Graph* recurseBuild(vector<string> tokens, int* i);
  - Controls the recursive build of the Graph.



- Graph *recurseBrackets(vector<string> *pVector, int *pInt);
  - If brackets are encountered, it's treated as a special case
- string helperValue(vector<string> *pVector);
  - Gets the helper between two graphs that will merge them together.
- Graph *createGraph(string basic_string, int *pInt);
  - Last step in forward tracking.
  - It creates a Graph* from a given id.



- Graph *splitToken(string basic_string, int *pInt, string h);
  - If input is not a definition, special character, the input is split into simple tokens concatenated together.
- Graph *createGraph(vector<string> *pVector, string basic_string, int *pInt, string h);
  - Chooses which path the creation would go to
    - creategraph() -> the token reached its simplest form.
    - splitToken() -> the token needs to be further simplified.
    - createGraphFromExistingDefintition() -> a definition exist, so no need to create a new one since createGraph() creates a new definition.

- Graph* createGraphFromExistingDefintition(Definition* def, int* i, string temp) ;
- vector<string> expandDef(string def);
  - Expand given token example
    - 0-9 will be expanded to (0,1,2,3, ….9)

- In Helpers
  - Graph* mergeGraphs(Graph* graph_1, Graph* graph_2h, string helper, int* id);
    - Given two graphs, a helper between them choose the appropriate function to merge them from :
      - Graph *mergeOr(Graph *pGraph, Graph *graph_2, int* id);
        - Adds new start and end states.
        - Adds epsilon transitions from the new start to the start of the given graphs.
        - Adds epsilon transitions from the end of the given graphs, to the new end;
        - Status is updated to have both the status of the end nodes of the given graphs “|” together, example :
          - End1_st -> A
          - End2_st -> B
          - newEnd_st -> A|B
      - Graph *mergePlus(Graph *pGraph, int* id);
        - Adds new start and end states.
        - Adds epsilon transition from the new start to the start of the given graph.
        - Adds epsilon transition from the end of the given graph, to the old start of the given graph;
        - Adds epsilon transition from the end of the given graph, to the new end;
        - Status is updated to have both the status of the end nodes of the given graphs “|” together, example :
          - End_st -> A
          - newEnd_st -> A*

- Graph *mergeAst(Graph *pGraph, int* id);
    - Calls mergePlus() first.
    - Adds epsilon transition from the new start to the new end.
    - Status is updated to have both the status of the end nodes of the given graphs “|” together, example :
        - End_st -> A
        - newEnd_st -> A+
- Graph *mergeCont(Graph *pGraph, Graph *graph_2, int* id);
    - Adds state.
    - Adds epsilon transition from the end state of the first graph to the new state.
    - Adds epsilon transition from the new state to the start state of second graph.
    - Status is updated to have both the status of the end nodes of the given graphs “|” together, example :
        - End1_st -> A
        - End2_st -> B
        - newEnd_st -> AB

Assumptions

- If the given token is not a definitions or a special character, it’s further split into simpler tokens.
-

Alternative designs

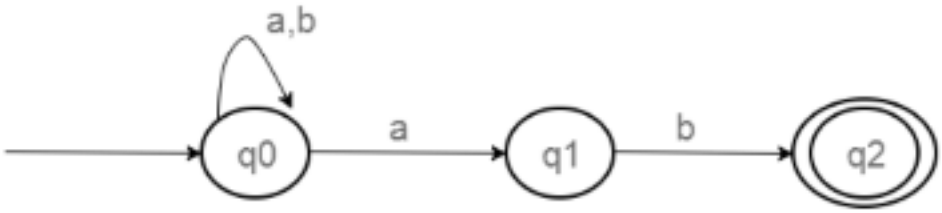
- A more optimal solution would’ve been splitting by the “|”, then creating all the graphs then Oring them together.
    - We will consider doing this before the next phase.

Deriving DFA

Overview

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.

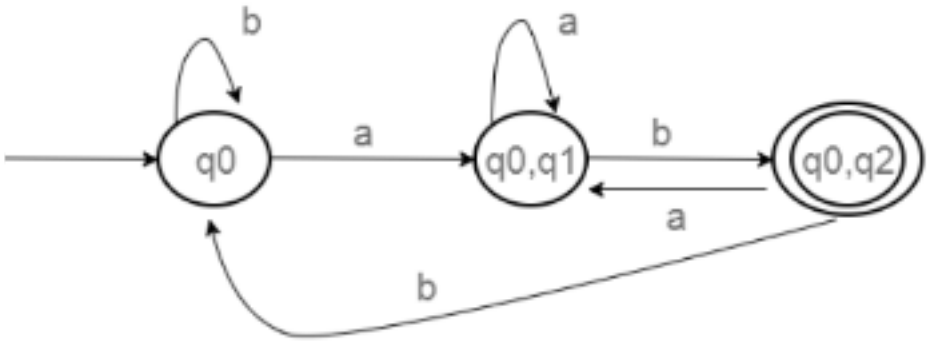
Input



Output

Note the actual output is the table, the graph is for demonstration only.

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0



Algorithm

Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q', ∑, q0, δ', F' > can be constructed for language L as:

Step 1: Initially Q' = ϕ.

Step 2: Add q0 to Q'.

Step 3: For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

To implement the previous algorithm we transform the NFA graph to a transition table as shown. This table helps in finding all possible transition for a certain input.

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Data-structure

map< Node*, map<Definition*,set<Node*>>> nfaStateTable: this data structure contains the NFA transition Table after BFS() function is called.

vector< set<Node*>> stateMappingTable: we save for each new state in the DFA transition table its old set of state from the NFA to be able to check if the DFA transition table contains same state or not.

vector< pair <Node*, map<Definition*,Node*>>> transitionStateTable:

Functions

set<Node*> getTransitionStates(Node* state, Definition* def): returns the next transition for a state (q) fromNFA transition table.

set<Node*> getEpsilonClosure(Node* state): gets the epsilon closure for a certain state by looping until all epsilon closures for each state is added to a set to form a single state (q') in the DFA transition table.

Node* getNodeStatus(set<Node*> set): returns the node with highest priority to add its status to the new Node created in the DFA transition table.

void insertNewStateInInitialTable(Node* node): it inserts a new state in NFA transition table after creating a map containing the edge weight (transition) in NFA graph and the states it reaches for each transition

void BFS(Node* start): it traverses the NFA graph to create the NFA transition table as part of the preparations done to construct the DFA transition table.

set<Node*> loopDefinition(Definition* def, Node* node): it loops for each definition and returns for each transition the destination states to be set in the map. This function is called in the insertNewStateInIntialTable function.

int tableContainsTheSameState(set<Node*> state): this function checks if the transition state found does exist in the DFA transition table, if it does it returns its id, else it return 0 and we add a new state in the table.

DFA( ): the DFA constructor follows the algorithm mentioned above using the helper of previous functions. For each state in the DFA transition table not visited yet, we get its destination set for each transition, get their epsilon closure and convert it to a new state in the DFA transition table. When no state for a certain transition is available, we direct the transition to a dummy state representing no accepted state.

Minimizing DFA

Overview

For each regular language, there also exists a **minimal automaton** that accepts it, that is, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching.

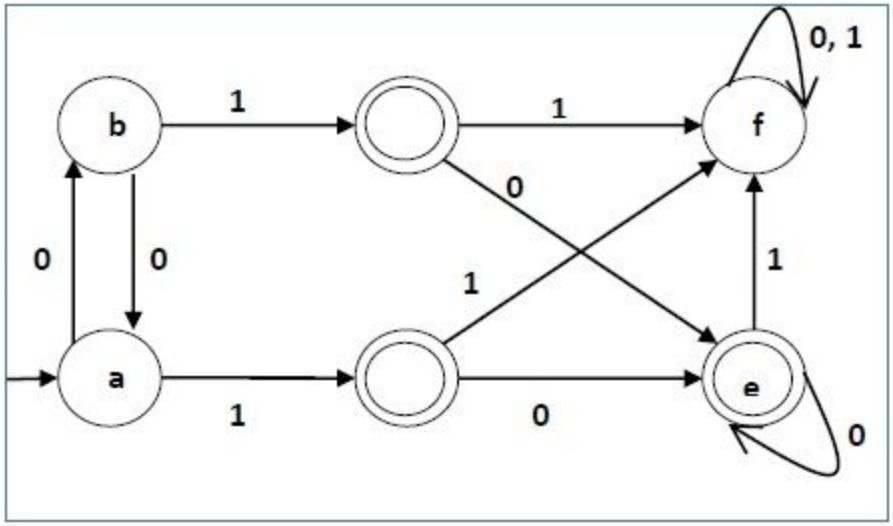
There are two classes of states that can be removed or merged from the original DFA without affecting the language it accepts to minimize it.

- **Unreachable states** are the states that are not reachable from the initial state of the DFA, for any input string.
- **Non Distinguishable states** are those that cannot be distinguished from one another for any input string.

DFA minimization is usually done in three steps, corresponding to the removal or merger of the relevant states. Since the elimination of non distinguishable states is computationally the most expensive one, it is usually done as the last step.

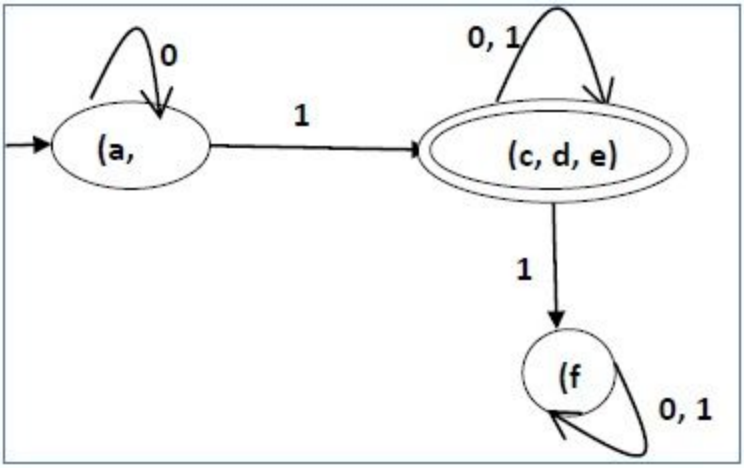
Input

- DFA not minimized



Output

- Minimized DFA



Algorithm

1. Start with an initial partition with two groups, F and S F, the accepting and non accepting states of D.
2. Apply the procedure of Fig. 3.64 to construct a new partition new. initially, let new = ; for (each group G of) f partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group of ; /* at worst, a state will be in a subgroup by itself */ replace G in new by the set of all subgroups formed; g Figure 3.64: Construction of new
3. 3If new = , let nal = and continue with step (4). Otherwise, repeat step (2) with new in place of .
4. Choose one state in each group of nal as the representative for that group. The representatives will be the states of the minimum-state DFA D0. The other components of D0 are constructed as follows
    a. The start state of D0 is the representative of the group containing the start state of D.
    b. The accepting states of D0 are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only nonaccepting states, because we started by separating those two classes of states, and the procedure of Fig. 3.64 always forms new groups that are subgroups of previously constructed groups.
    c. Let s be the representative of some group G of nal, and let the transition of D from s on input a be to state t. Let r be the representative oft's group H. Then in D0, there is a transition from s to r on input a. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according to Fig. 3.64.

Data-structure

- vector<pair<Node*, map<Definition*, Node*>>> minimize Transition State Table

- This is the table which contains each state and her transition under specific Definition

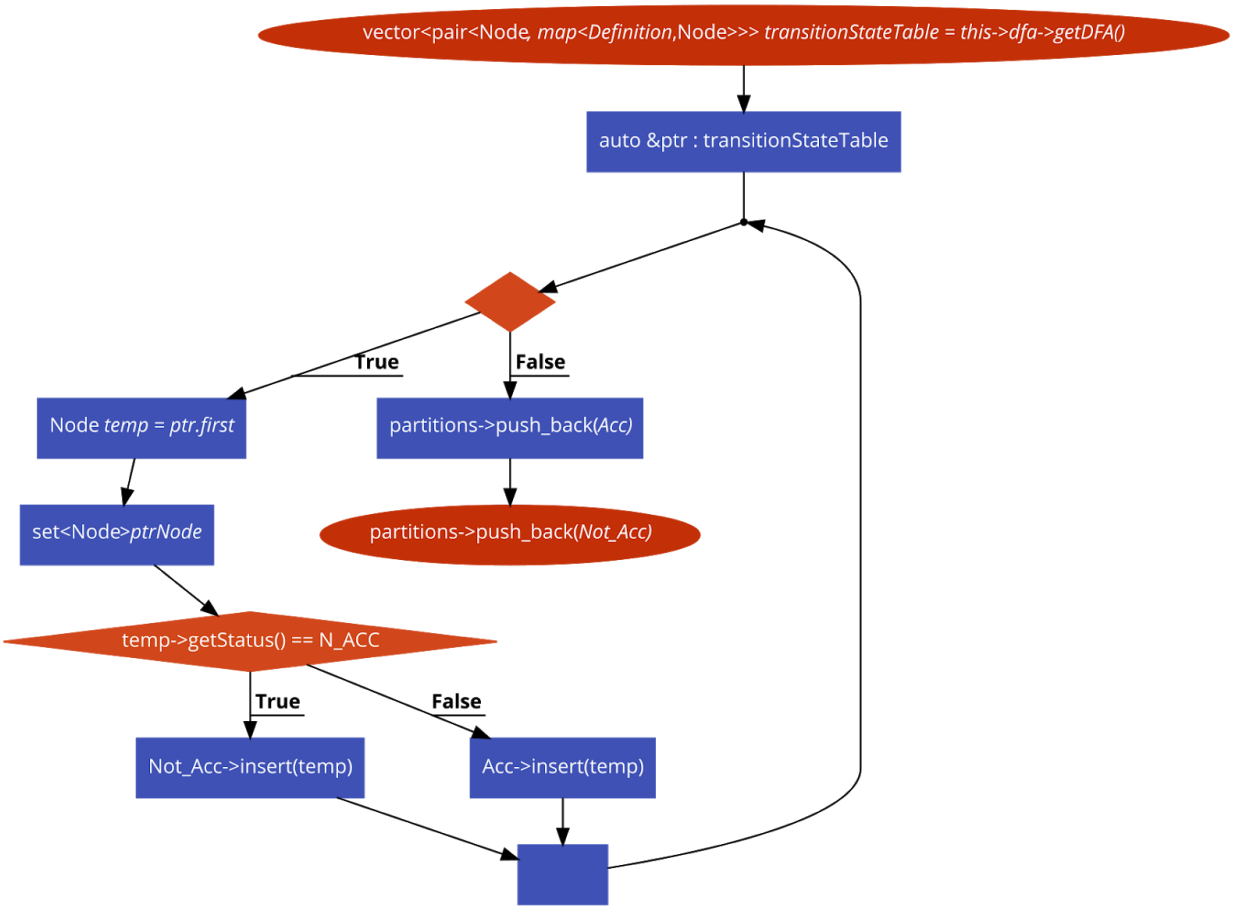
  - Example :-

q	δ(q,0)	δ(q,1)
a	b	c
b	a	d
c	e	f
d	e	f
e	e	f
f	f	f

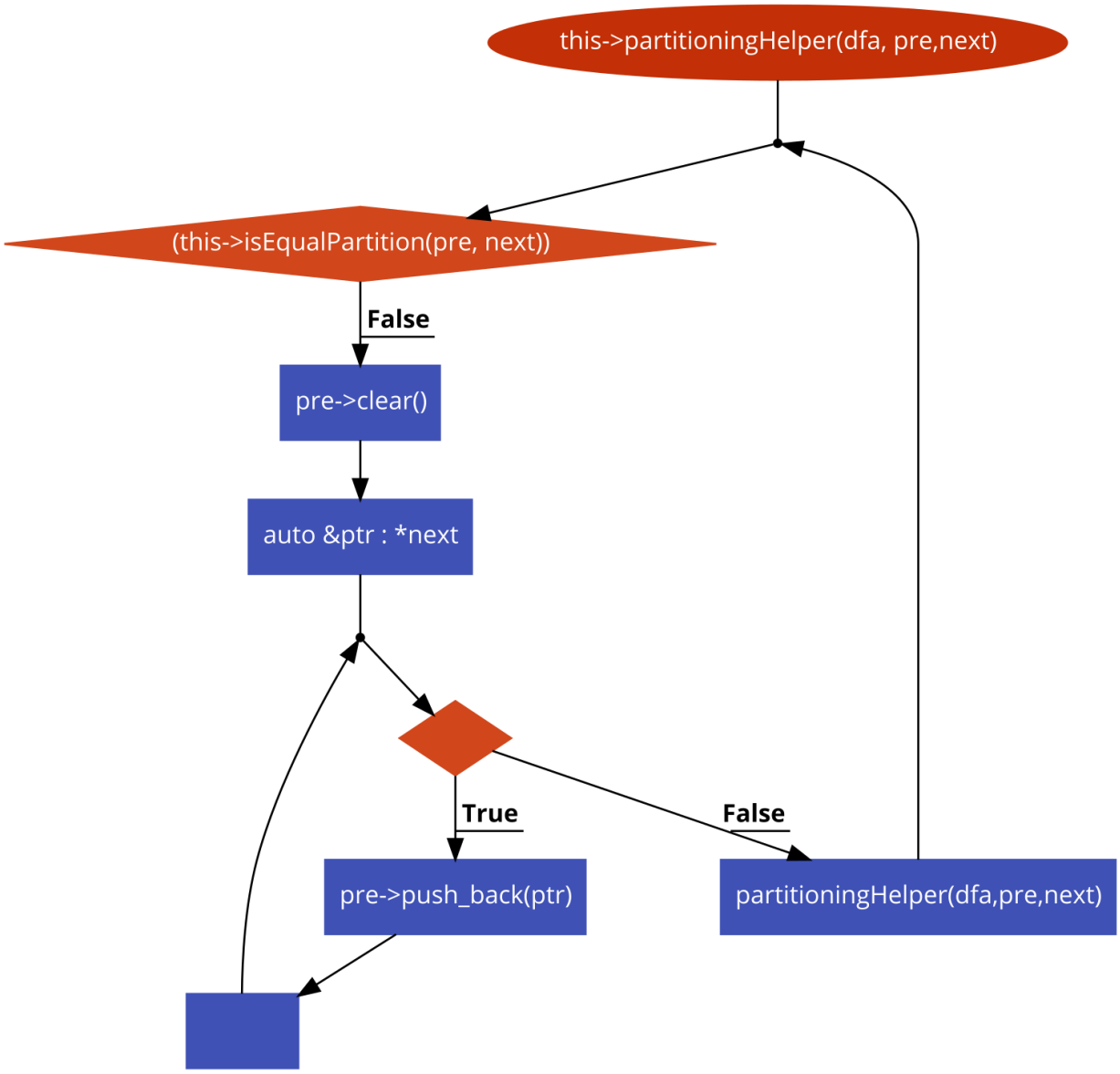
- set<Node*> Accepted;
  - Contain all accepted state in first partition.
- set<Node*> Not Accepted;
  - Contain all non accepted state in first partition.
- vector<set<Node*>> previous Partition;
  - This is a helper data structure contain the previous set partition.
- vector<set<Node*>> next Partition;
  - This is a helper data structure to put new sets in the current partition state.
- Node* startState;
  - To indicate the start state of DFA after minimization.

Functions

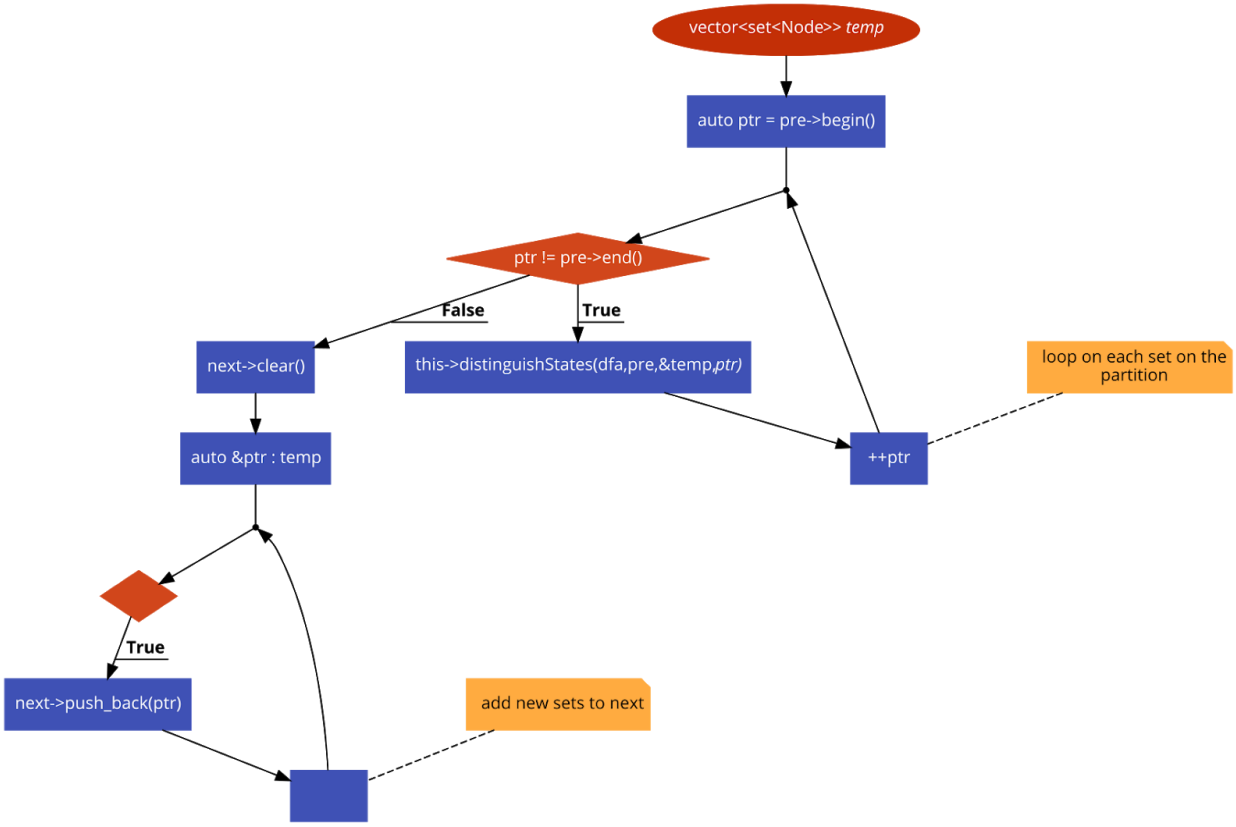
- **void** setFirstPartition(set<Node*> *Acc, set<Node*>* Not_Acc, vector<set<Node*>> *partitions);



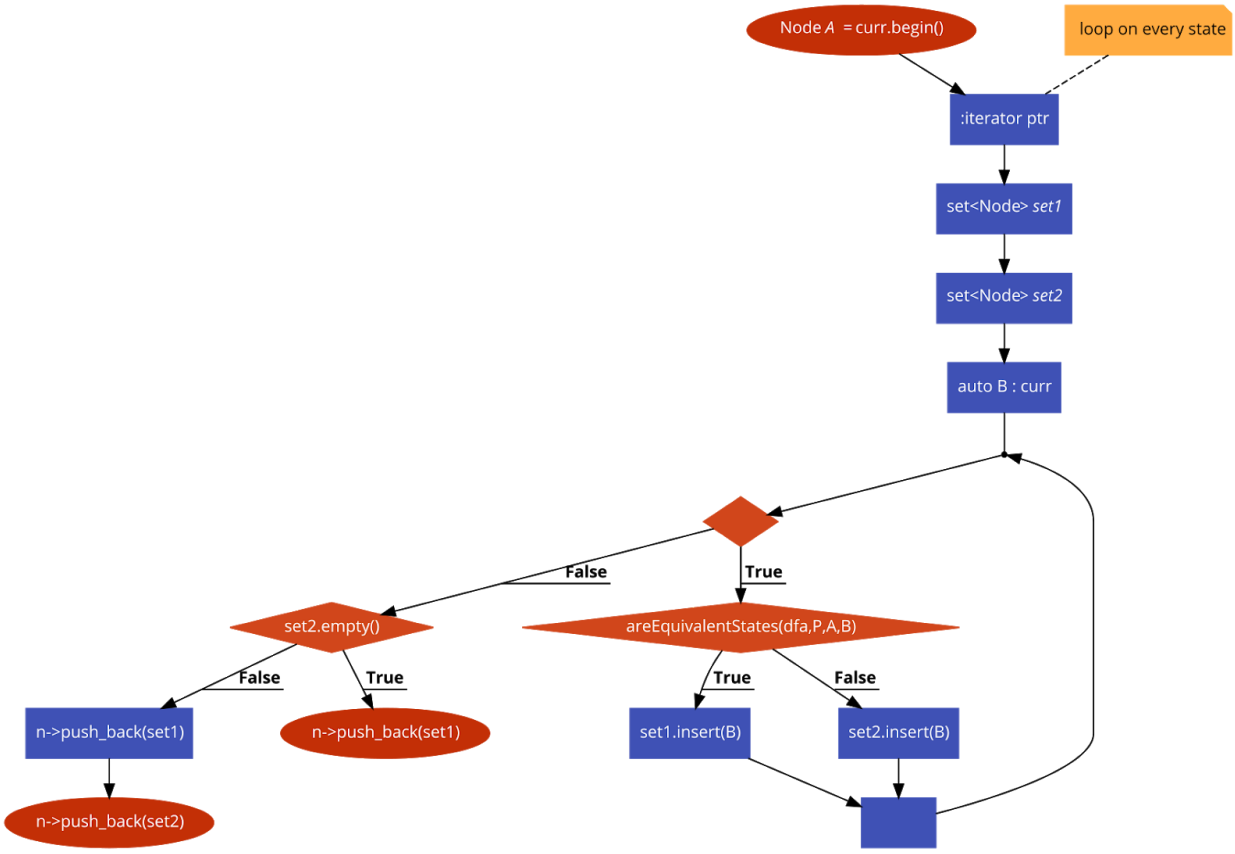
- **void** partitioning(DFA* dfa, vector<set<Node*>>*pre, vector<set<Node*>> *next);



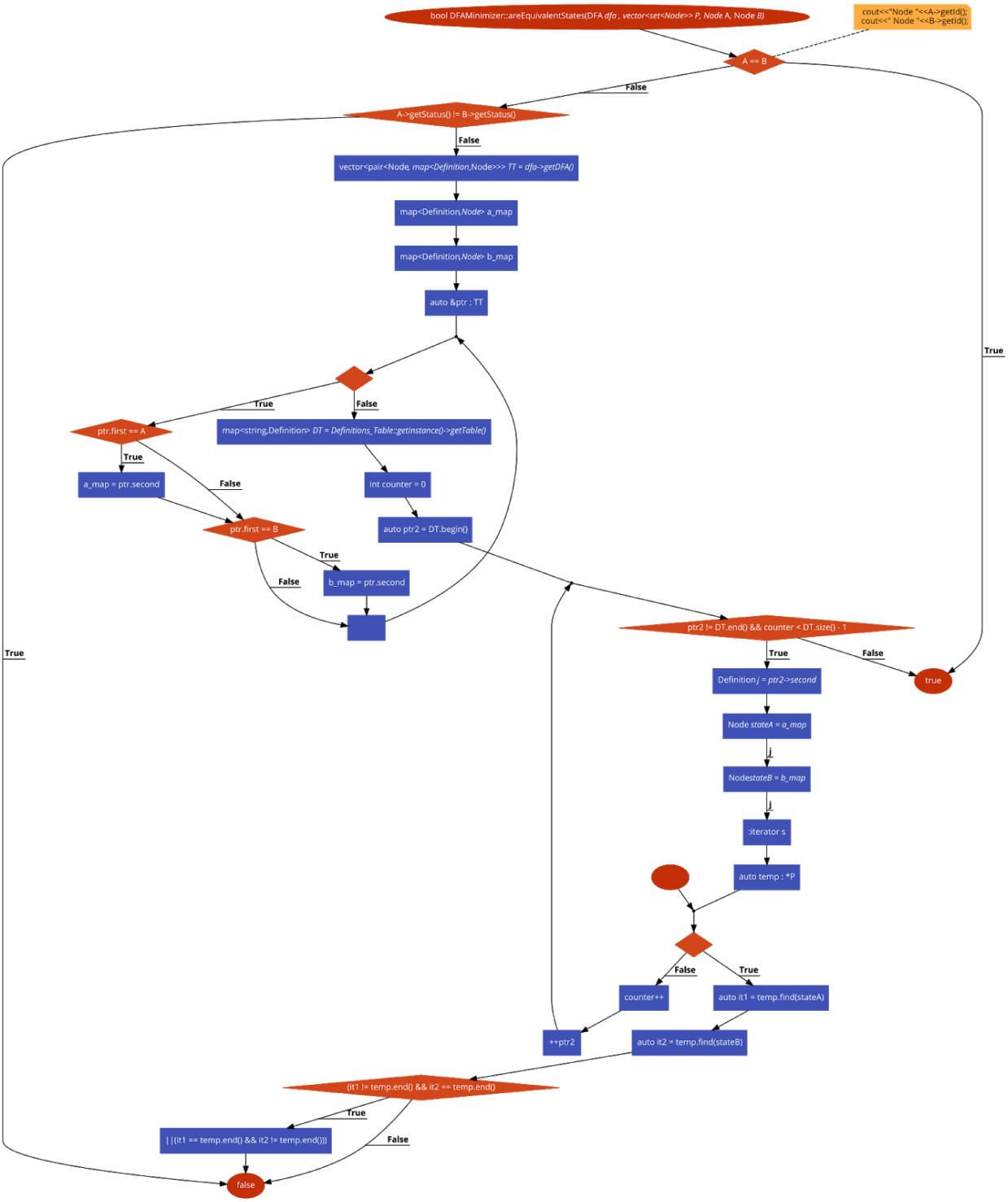
- **void** partitioningHelper(DFA* dfa,vector<set<Node*>>*pre,vector<set<Node*>>*next);



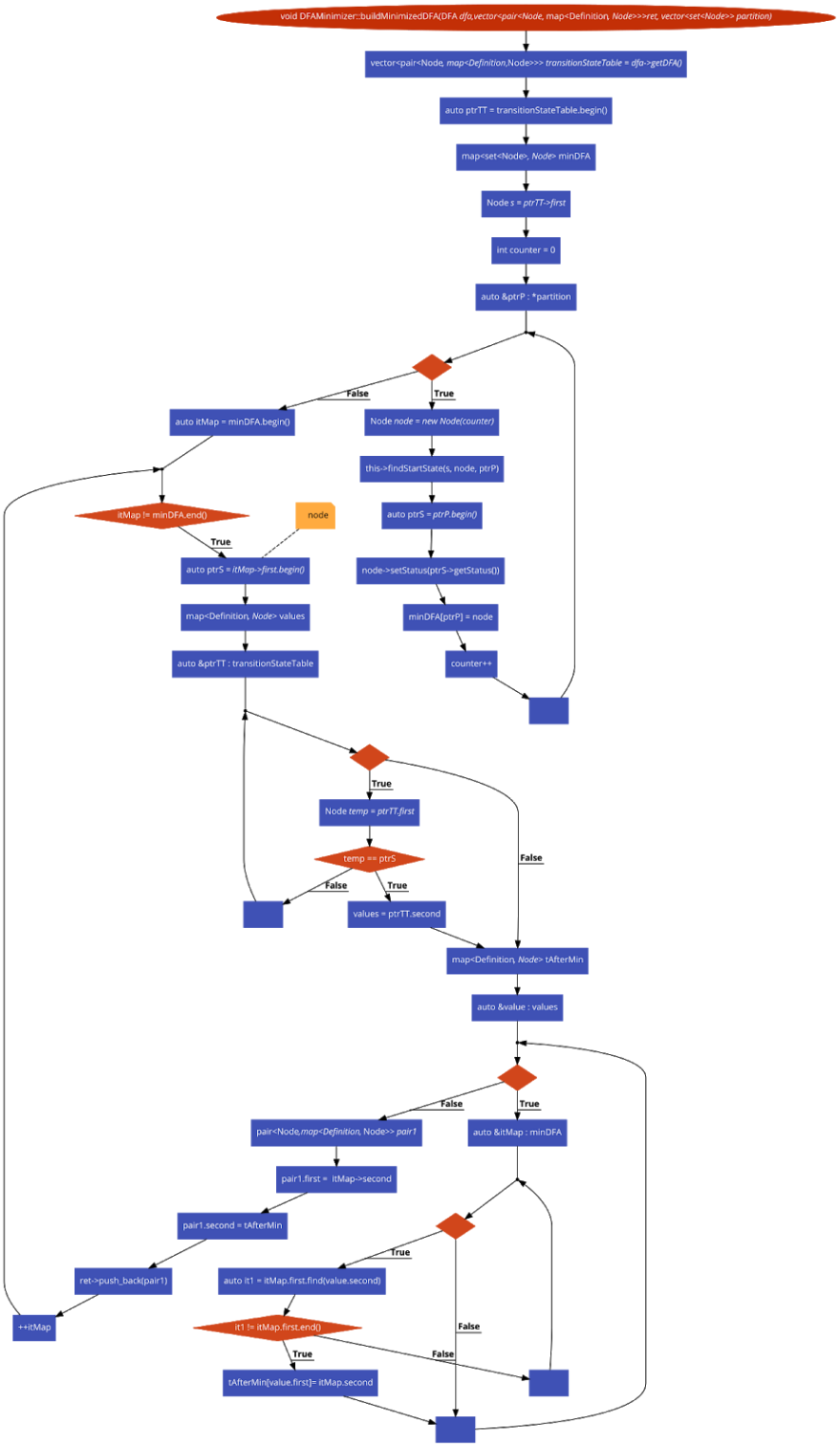
- **void** distinguishStates(DFA* dfa,vector<set<Node*>> *P , vector<set<Node*>> *n, set<Node*> curr);



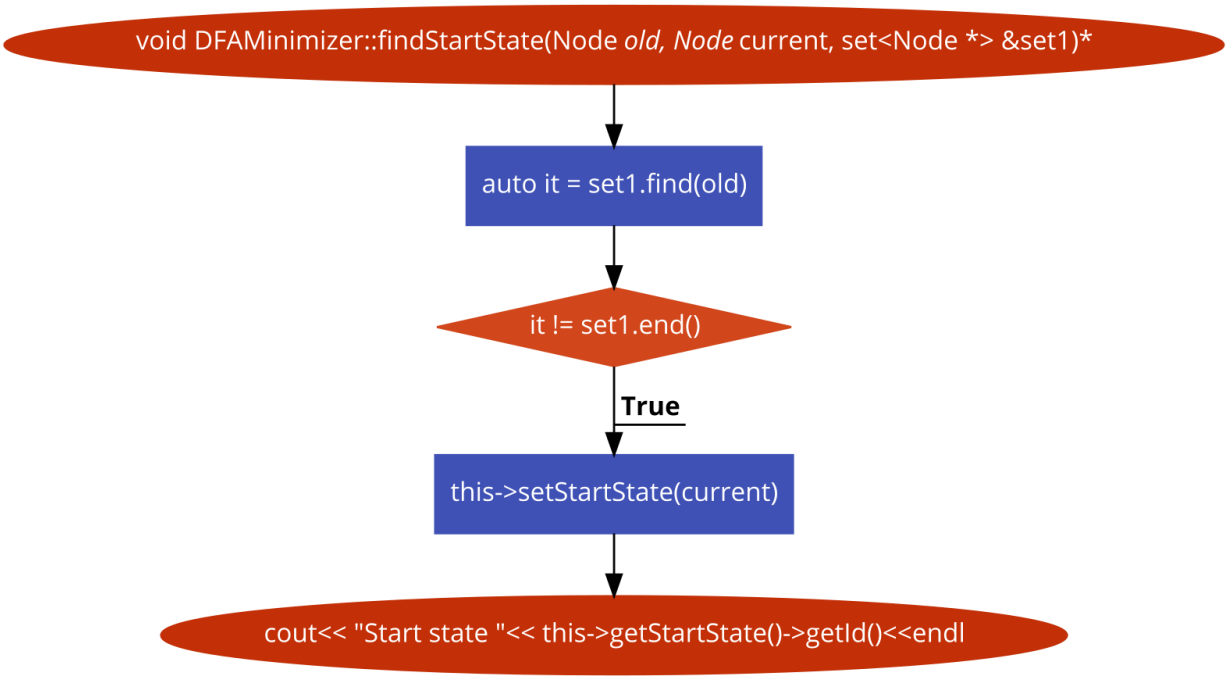
- **bool** areEquivalentStates(DFA* dfa , vector<set<Node*>> *P, Node* A , Node *B);


bool DFAMinimizer::areEquivalentStates(DFA dfa , vector<set<Node>> P, Node A, Node B)
cout<<"Node "<<A->getId();
cout<<" Node "<<B->getId();
A == B
False
True
A->getStatus() != B->getStatus()
False
vector<pair<Node, map<Definition,Node>>> TT = dfa->getDFA()
map<Definition,Node> a_map
map<Definition,Node> b_map
auto &ptr : TT
True
False
ptr.first == A
map<string,Definition> DT = Definitions_Table::getInstance()->getTable()
True
a_map = ptr.second
False
int counter = 0
ptr.first == B
auto ptr2 = DT.begin()
True
False
b_map = ptr.second
True
ptr2 != DT.end() && counter < DT.size() - 1
True
False
true
Definition j = ptr2->second
Node stateA = a_map
j
NodestateB = b_map
j
:iterator s
auto temp : *P
False
True
counter++
auto it1 = temp.find(stateA)
++ptr2
auto it2 = temp.find(stateB)
(it1 != temp.end() && it2 == temp.end())
True
False
| |(it1 == temp.end() && it2 != temp.end()))
false
True
True

- **void** buildMinimizedDFA(DFA* dfa,vector<pair<Node*, map<Definition*, Node*>>>*ret, vector<set<Node*>> *partition);



- **void** findStartState(Node *old, Node *current, set<Node *> &set1);



- /* Static access method. */
- **static** DFAMinimizer* getInstance();
  - We make DFA minimizer using Singleton Design pattern the table build ones
  - And the Class is Static to be seen from all the program
- vector<pair<Node*,map<Definition*,Node*>>> *getMinimizedDFA();

Assumptions

- All States in the Same Set has the same priority.
- DFA always has accepted and not accepted state.

Alternative designs

Make A class contain the structure of DFA instead using vector<pair<Node*, map<Definition*, Node*>>>

Parse program and output

Overview

Deterministic finite automaton. ... In a **DFA**, there is only one active state at any given time and there is only one possible transition for each input character. While the **pattern matching** steps for a **DFA** are straightforward, the construction of a**DFA** that **matches** multiple strings is slightly more complex.

Input

- Transition Table from the minimized DFA.
- Input file java program.

Output

- File contains all the tokens of the java program.

Algorithm

- Loop on the program file and split each line by spaces into tokens.
- Take each token and loop on to take each character and send it to the minimized DFA with the start state to get the next state of the DFA.
- If there isn't a next state (return null) then the token didn't match any pattern.
- If the next state is acceptance state save it to backtrack.
- If the loop reach the last character and the current state is not acceptance state back to the last acceptance state and push it to the tokens vector and start the second loop from the first character after the acceptance character and if there wasn't an acceptance state before, the token didn't match any pattern.

  int nextState(s;a) {

  if check base s a s return next base s a ;

else return nextState default s a ;

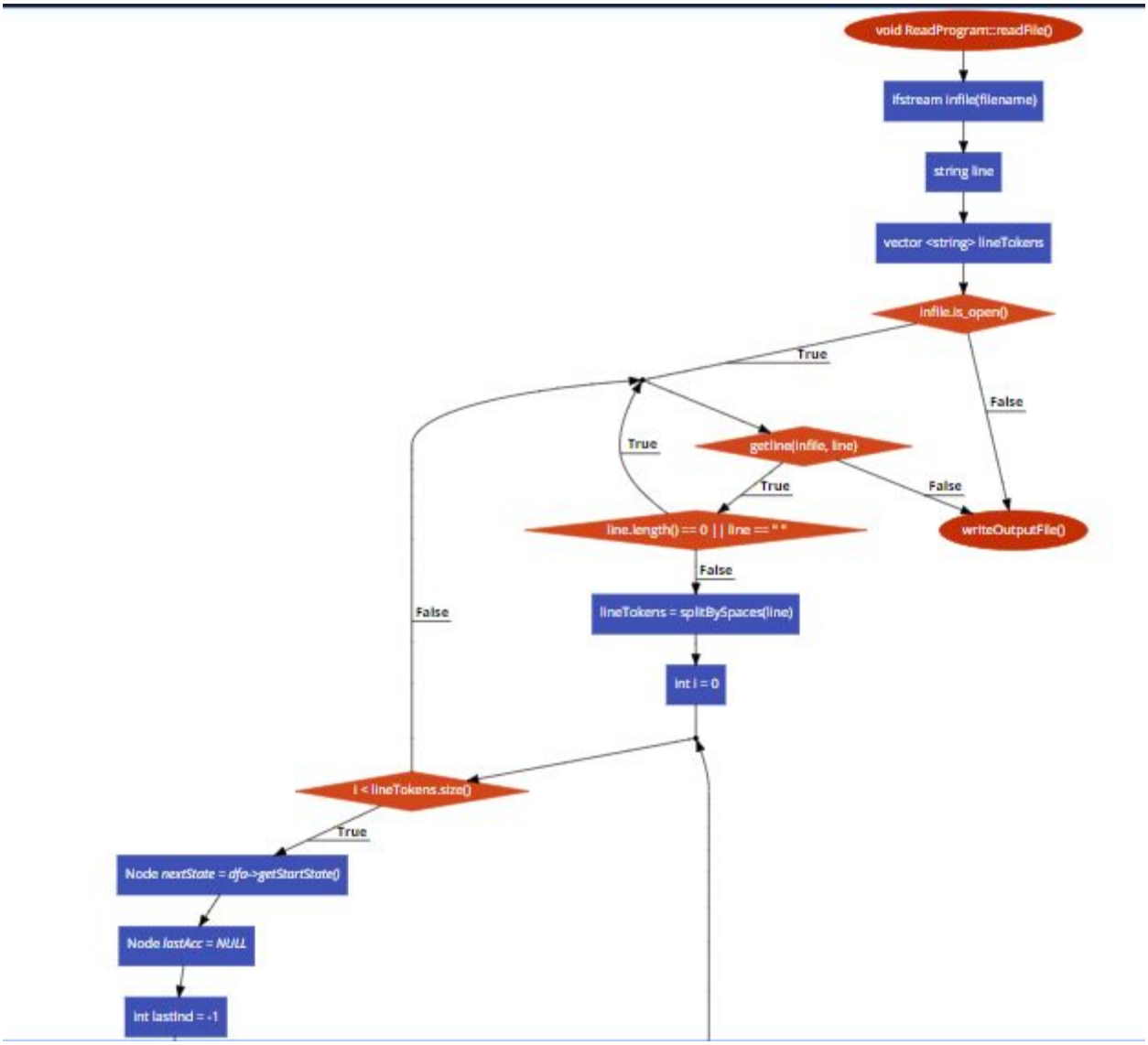
}

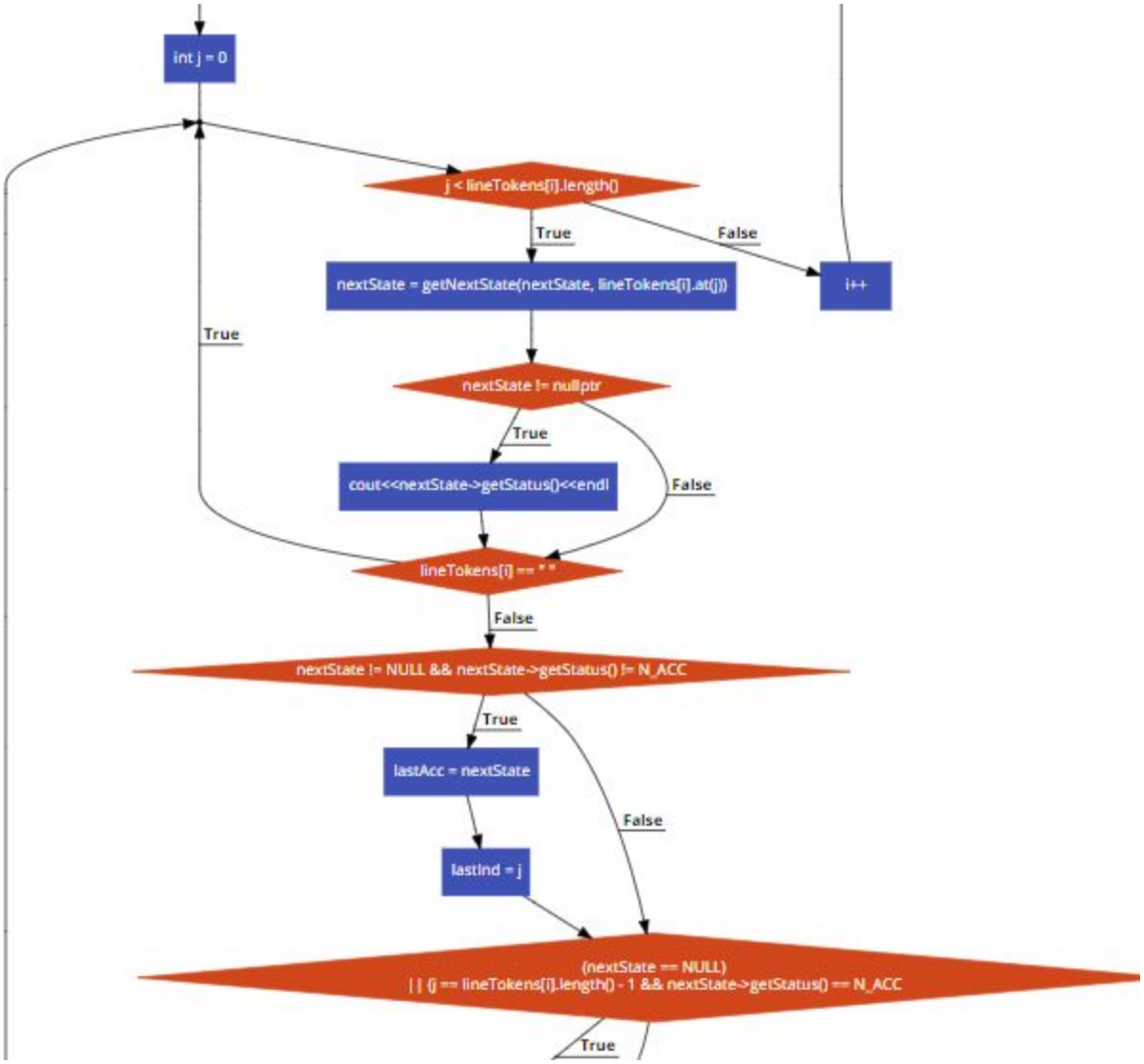
Data-structure

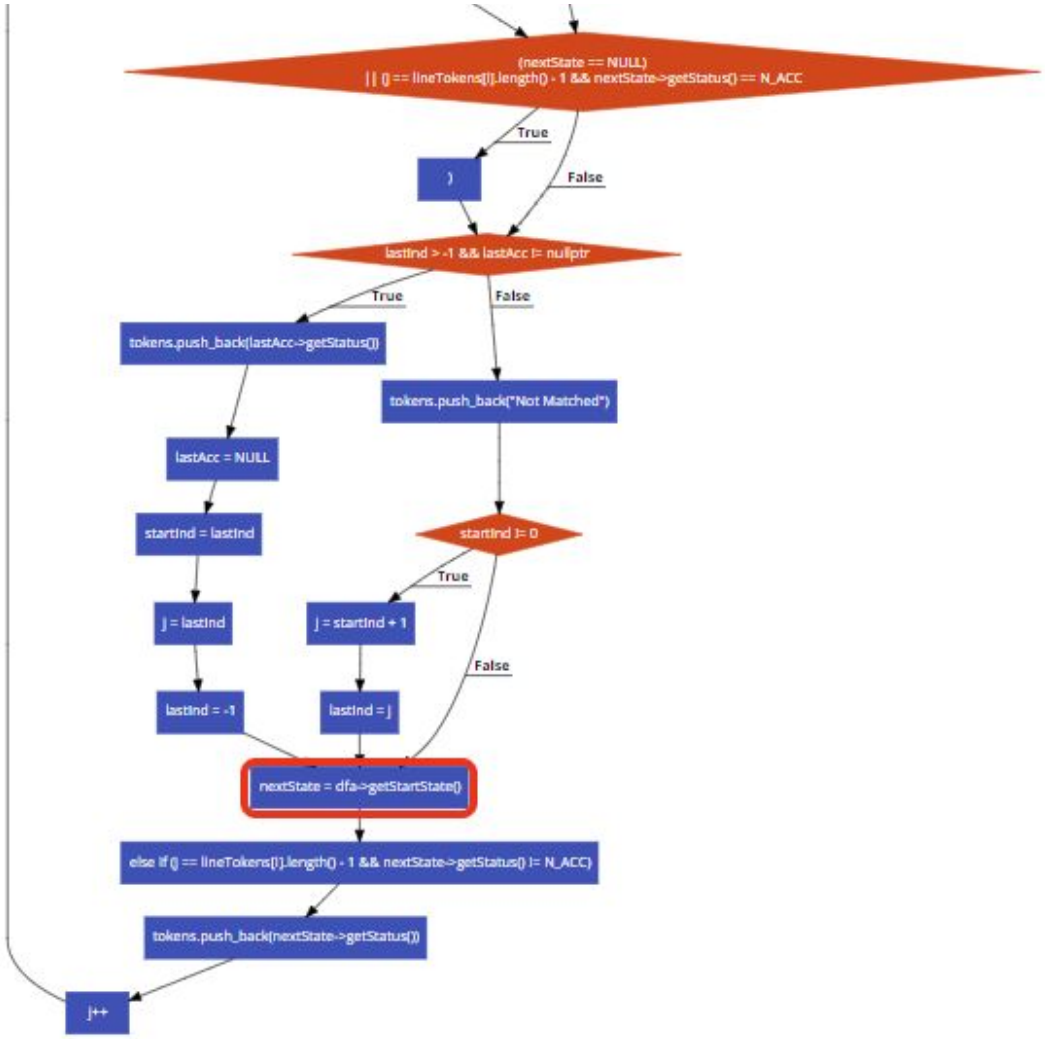
- Vector<string> tokens.

Functions

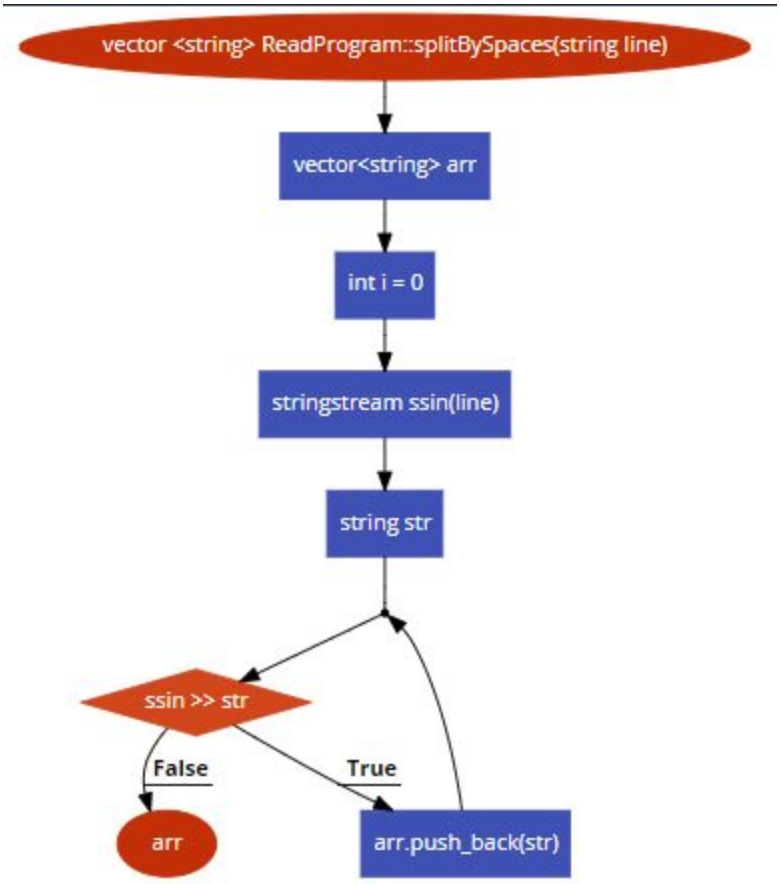
- Void readFile()



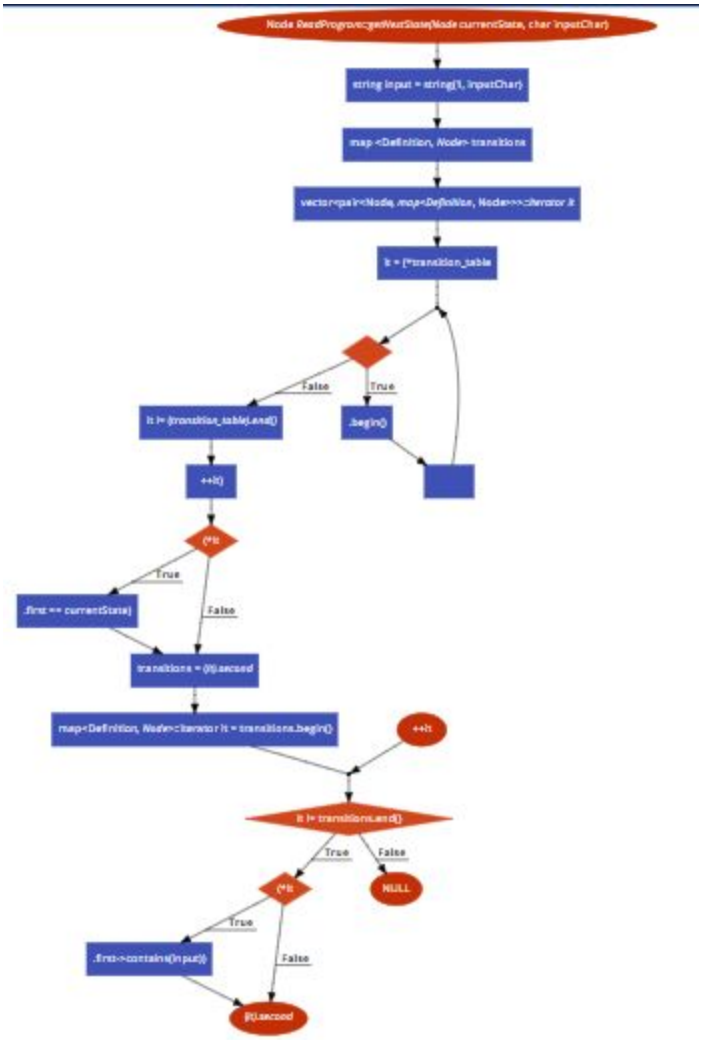

int j = 0
j < lineTokens[i].length()
True
False
nextState = getNextState(nextState, lineTokens[i].at(j))
i++
True
nextState != nullptr
True
cout<<nextState->getStatus()<<endl
False
lineTokens[i] == " "
False
nextState != NULL && nextState->getStatus() != N_ACC
True
lastAcc = nextState
False
lastInd = j
(nextState == NULL)
|| (j == lineTokens[i].length() - 1 && nextState->getStatus() == N_ACC
True


(nextState == NULL)
|| (j == lineTokens[i].length() - 1 && nextState->getStatus() == N_ACC
True
)
False
lastInd > -1 && lastAcc != nullptr
True
False
tokens.push_back(lastAcc->getStatus())
tokens.push_back("Not Matched")
lastAcc = NULL
startInd = lastInd
startInd != 0
True
j = lastInd
j = startInd + 1
False
lastInd = -1
lastInd = j
nextState = dfa->getStartState()
else if (j == lineTokens[i].length() - 1 && nextState->getStatus() != N_ACC)
tokens.push_back(nextState->getStatus())
j++

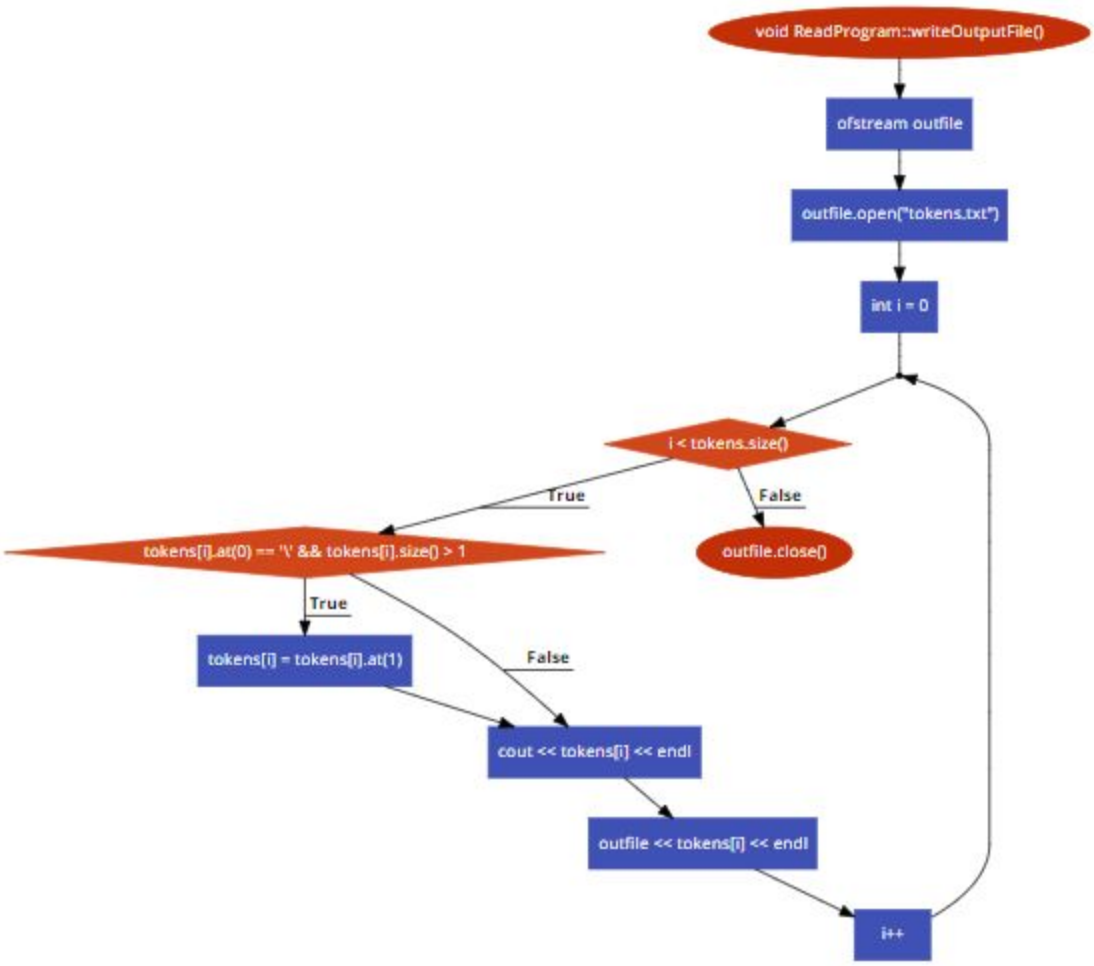
- vector<string> splitBySpaces(string line)



- Node* getNextState(Node* currentState, char inputChar)



- Void writeOutputFile()



Assumptions

- Input line is split by spaces, so only strings with continuous characters with no spaces in between could be matched or unmatched to and expression, example :
    - Digits.digits E digits will be matched to num id num.
    - If you want it to match as num it should be continuous -> digits.digitsEdigits.

Alternative designs

No alternative designs.

Design Patterns

- Singleton
    - Definitions_Table
    - Helpers
    - NFA
    - DFA
    - Template
- Factory
    - Construct_Automata
        - Takes tokens and create subgraphs which later are merged into NFA.
- Facade
    - NFA to DFA.
    - DFA to minimizer.
    - Minimizer to output generating.

Problems faced

- Structure:
    - Determining how the definition is represented.
- Parsing
    - Parsing special characters “\\”.
- Traversing Definition
- Minimization
    - Finding Efficient data structure to decrease the run time.

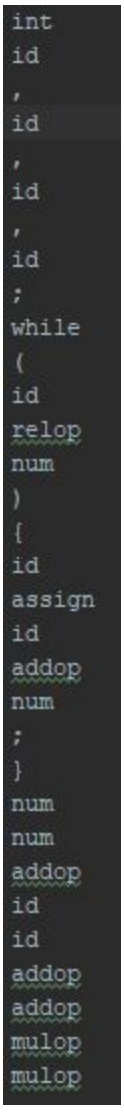
Sample runs

- Grammar :-

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

- Program :-
  -

```
int sum , count9 , pass , mnt;
while (pass != 9.9){
    pass = pass+  1 ;
}
99
20
+
for
for+-//
```

- Output :-
  - 
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
num
num
addop
id
id
addop
addop
mulop
mulop

Example 2 :- accept odd zero and even ones

- Grammar :-

```
Zero = 0
One = 1
L1: Zero (Zero Zero)*
L2 : (One One)*
```

- Program :-
  - ```
    00110
    ```

- Output :-
  - ```
    L1
    L1
    L2
    L1
    ```

- Minimized DFA :-
  - ```
    Start state 4
    Def   0 1  Accepted
    2:    3  0 <n>
    3:    3  3 <n>
    0:    3  2 <L2>
    1:    5  3 <L1>
    4:    1  2 <n>
    5:    1  3 <n>
    ```
    ```

- Example 3 :-
  - ```
    float x, y , c$;
    if ( x == 10.10E11) {
        x = x + y;
        c = x - y;
    }
    ```
  - ```
    float
    id
    ,
    id
    ,
    id
    Not Matched
    ;
    if
    (
    id
    relop
    num
    )
    {
    id
    assign
    id
    addop
    id
    ;
    id
    assign
    id
    addop
    id
    ;
    }
    ```

Example 4:-

```
for(int i;i<2;i++){

}
```

```
id
(
int
id
;
id
relop
num
;
id
addop
addop
)
{
}
```

What's Next

To ensure a one-pass compiler,the lexical analyzer should work closely with the syntax analyzer. When it reads character streams from the source code, then checks for legal tokens, it will passe the data to the syntax analyzer when it demands.

We tried to make the lexical analyser phase as modular as possible to allow small tweaks if needed.

Tasks distribution

- Aya Ashraf
    - Minimization, testing, report.

- Rowan Adel
    - Parser, output, testing, report.
- Sarah Ahmed
    - DFA, testing, report.
- Sohayla Mohammed
    - Parser, NFA, testing, report, bonus.

References

- For project
    - minimization :-
        - Compilers: Principles, Techniques, and Tools (2nd Edition)
        - tutorialspoint
- For bonus
    - Geeks for Geeks