Lexical Analyzer Generator Report

Mostafa Hamdy (64)

Mahmoud Saleh (62)

Waleed Adel (74)

Diaa Ibrahim (27)

Abdalla Mahmoud (35)

- ## **NFA Class :**

1. map<string , State*>regular_expression;

   this map is used to hold all the regular expression that exist in the input file, we store the name of the regular expression and we store the first node in the regular expression, form the first node we can traverse the whole NFA graph.

2. map<string , State*>regular_definition

   same as (regular_expression) but we store only the regular definition.

3. set<string> inputSet;

   this set is used to hold all the input characters that we found in the input file, this set will be used in the next stages.

- ## **Algorithms and Techniques:**

- PostFix :
  - we used this technique to transform the given expression to a postfix expression to make it easy for evaluating

  - string postfix(string expression);

this method takes a string that represents the expression , and returns a string that represents the postfix.
we use stack to hold the operators , every time we encounter an operator we push it in the stack but first we check that :

1. the element at the top of the stack has priority higher than the operator we are trying to push, if this happens then we have to pop all the operator that have greater priority or until we find ("(")and then push the new operator.
2. if the operator is (")") then we have to pop all the elements in the stack until we find (")")

- the priorities are :

  * (highest), then concatenation, then +(lowest).

- Evaluating postFix expression:

    - this technique is used to evaluate the postfix expression that we generated earlier, we use stack but this time the stack holds the operands

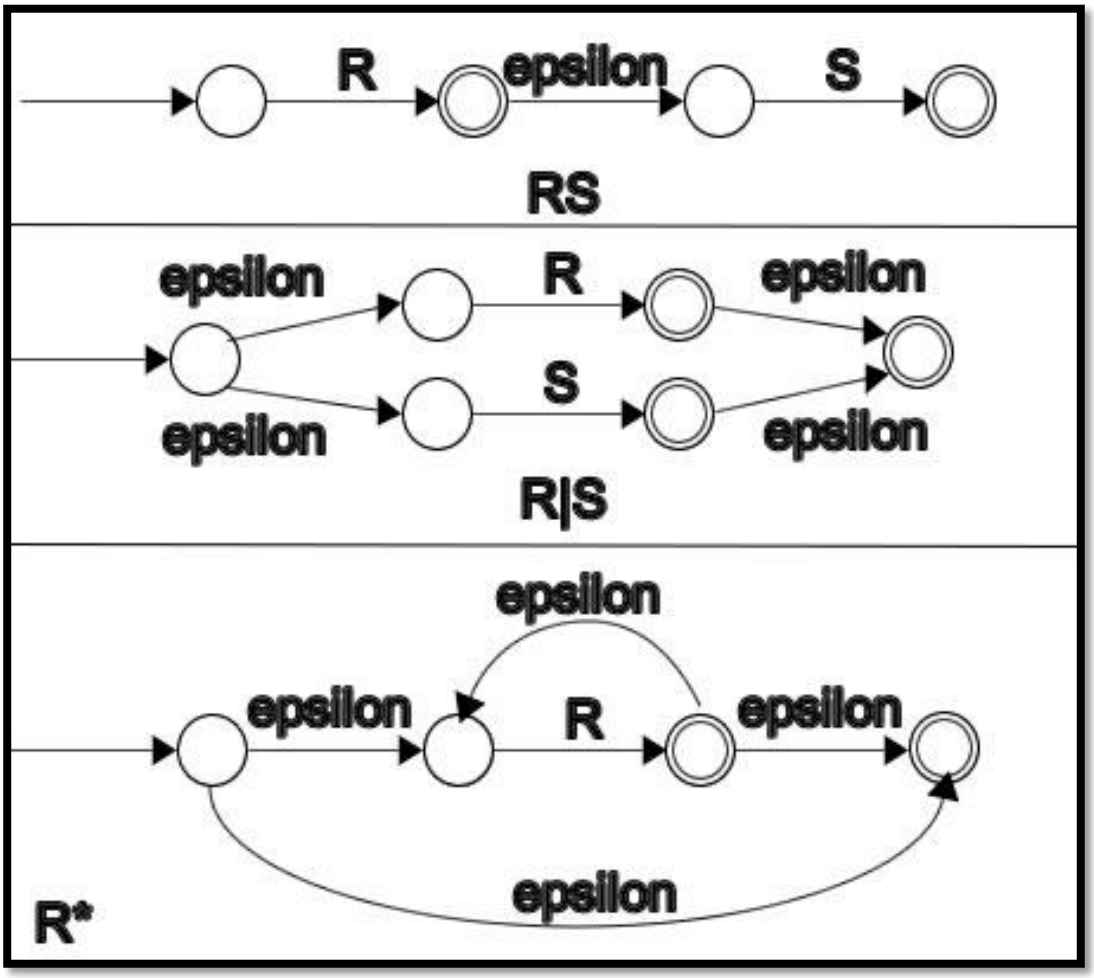
        - when we encounter an operator we do the following:

            1. if the operator is * (or +), we need to pop one element from the stack
            2. if the operator is |(or concatenation) , we need to pop two elements form the stack

- BFS(State* start)
  we used the BFS algorithm to traverse the NFA graphs.
  and print all the states

in the picture below we show how the operations are implemented :

1. Concatenation.
2. OR
3. Kleene closure



Converting NFA to DFA

(Subset Construction Algorithm):

Data Structure

1- vector<State*>DFAtable;

This table contains all DFA state we created through the algorithm to be minimized after that.

2- set<string>inputSet;

This set contains all possible inputs.

Algorithm

Mainly, we have two function that we used in this algorithm :

1- Epsilon Clousure :

void epsilonClosure(set<State*>states,set<State*>&result);

This function takes a set of states and returned also as a parameter a set of states as the result which are the reached states from the input states with epsilon input. This function works as the following:

1- Intialize the result with the input states set.

2- Push all the states of input states onto the stack.

3- While (the stack is not empty){

4- Pop the top of the stack.

5- Get all states that reached from this state with input of -1(refer to epsilon).

6- Add the reached states to the result if they are not already existed.

7- Push the reached states to the stack.

2- Move Transition:

void moveTransition(set<State*>states,string input,set<State*>&result);

This function takes a set of states and returned also as a parameter a set of states as the result which are the reached states from the input states with the specified input. This function works as the following:

1- For each state of the input set

2- Get all states that reached from this state with the specific input determined in the parameter.

3- Add the reached states to the result if they are not already existed.

Then using those two function we can peform our algorithm to get the DFA:

1- We get the starting state of the DFA by performing the Epsilon Clousure function on the starting state of the NFA.

2- For each new DFA state, perform the following for each input character:

3- Perform move to the newly created state.

4- Create new state by taking the Epsilon closure of the result .

5- For each newly created state, perform step 2.

6-Accepting states of DFA are all those states, which contain at least one of the accepting states from NFA.

Parse From Java File

Simulation

used Data Structure :

1 - set<string>input Scope :

- contains all possible input characters.

2-vector <State>dfaTable :

- contains all states of the minimized DFA graph .

3- set<string>kwards :

- contains all the key Wards specified in the input File .

4- set<string>punctuation :

- contains all the punctuations specified in the input File .

5- set<string>symbolTable :

- used to contain all the identifiers that will be in the Java program file .

Algorithm:

First , read from the Java file line by line then split each line by the separator (space) into array of words then parse each word character by character .

Algorithm of parsing :

we have four variables :

1 - **char** my_character : store the current character .

2 - String token : store the current string that we read form the word until now .

3- State state : contain the current state from the minimized DFA graph (vector dfaTable) .

4- String finalState : store the Type of the **class** of the last Accepting state (ex : id , num , relop , ...)

First , Read the current character and check **if** it belongs to the scope of the language or not by the set of strings (input Scope) :

A - **if** it an '**\n** ' character :

1 - check the token String and finalState string , **if** the token not empty and finalState not empty then :

- **if** our token until now is a key Ward from (set of strings kWards) , then print the token string .

- 1_2 - **if else** , print the type of **class** that

  stored in the finalState string .

- **if** the type of **class** was an identifier then insert it into the symbolTable set .

2- **return** to the startState .

B - **if** it a punctuation (from the set of strings (punctuation))

- exactly the same of the point 1 in the

  above .

- print the punctuation mark .

- **return** to startState .

C- **if** it doesn't belong to the scope of the language :

1 - check the token String and finalState string , if the token not empty and finalState not empty then:

- if our token until now is a key Ward from (set of strings kWards) or is a punctuation from (set of strings (punctuation)) , then print the token string .

- if else , print the type of class that stored in the finalState string .

- if the type of class was an identifier then insert it into the symbolTable set .

2 - print an error message .

3 - substring the entire word from the character that comes after the current character to the end of word and start parse this substring .

D- if it belongs to the scope of the language :

1- get the next state in the DFA graph .

- if there is no next state :

- if the current state was the starting state

  - print an error

  - substring the entire word from the character that comes after the current character to the end of the word and starting pares this substring recursively .

- if the current state is not accepting state :

  - if the finalState is empty then print an error .

  - if the type of class in the finalState string is an identifier then, insert it into the symbolTable .

- substring the entire word from the current character to the end of the word and starting pares this substring recursively .

- if the current state is an accepting state :

- check the token String and finalState string, if the token not empty and finalState is not empty then :

- if our token until now is a key Ward from (set of strings kWards) , then print the token string .

- if else , print the type of class that stored in the finalState string .

- if the type of class was an identifier then insert it into the symbolTable set .

- return to starting state and get the nextState by the current character .

- if the nextState is an accepting state then store the class type of this state in startState string .

2- if there is a next state then make this our current state (State state)

- check if it is an accepting state then

store the type of the class in finalState string .

Assumption:

example : " 123abs"

our assumption : this string is a number followed by a identifier . not a num then error .

our output :

num >> 123

id >> abs

justification :

1 - if we consider this string is an error then we will consider : "while(" is an error too .however that string is key ward followed by an punctuation .

2 – we look forward to the longest accepting so we will accept " abs " as an identifier .

Stream of Tokens For The Example Test Program:

int
id
,
id
,
id
,
id
;
while
(
id

relop
num
)
{
id
assign
id
addop
num
;
}

Dfa Minimization:-

Data structures used:
vector of vector <state>

This vector to hold the whole workspace of states every vector in the containing vector represents a set that contains some states that are going to the same output when the same input is applying, And then breaking every small vector into more smaller vectors until we can't break it or it become one state.

HashMap

I used a hash map to identify between states that are going to some output when applying some input
how it works:
first get the state then see what edges that are going out of it then concatenate them in

one string with the type of that state then push the hashing string into the map as a key with the vector containing the states in the value of that key then we get all the states grouped by what they are going to.

Algorithm:-

First group all states into two groups one containing the accepted states and the other containing the non accepting states then send to the recursive method which see every state in every vector and hash every state depending on its going out states and its type in hashing map that collects between the hashing string and the vector containing the states that has the same hashing function. then send the new division between the states again into the recursive method until no division could be happen then we will create the new minimized states and send it to the simulation step.