Alexandria University CS321
Faculty of Engineering
Compilers
Computer and Systems Engineering Dept.
Third Year

Phase1: Lexical Analyzer Generator

Names	Numbers
Peter Atef	19
Radwa Adel	23
Aliaa Othman	41
Mark Philip	49

A description of the used data structures :

MAP :

first data structure to hold the NFA_table :
map< pair<int,string>, std::vector<int> > NFA_table
key : pair<int,string>
value:vector<int>
this map consists of pair of int and string and vector of string , the pair tells us number of node and class and vector consists of nodes that this node go to .

Second data structure to hold the DFA_table :
The map of the DFA table where each node paired with each class is a key and the value is the node mapped to it.
For example: (0, digit) -> 1

That means that node 0 at class digit is mapped to node 1.

map<string, vector<int>> terminals → is used to hold terminals states in DFA with string Indicate which the node accept in DFA.

map<int, vector<int>> gps → holds the the states associated with each group in partioning of states in minimization process.

Vectors :

vector <pair<string, string>> classes :
this vector consists of the class that used like digits , digit , letter

vector<int> non_terminals → is used to hold non terminals states and then use terminals and non terminals groups in minimization.

vector <pair<vector<int>,bool>> minimization_gps → hold the groups which will be minimized along the minimization process , bool indicate if the group is deleted or not and the vector represent the states in each group.

vector<pair<vector<int>, bool>> e_closure

- A vector contains the epsilon closure of every node, we use it on constructing the DFA table to get the new states.

vector<vector<int>> states

- A vector contains the new DFA states.

Explanation of all algorithms and techniques used :

infix to postfix :

convert the Regular expressions to postfix to get correct order of nfa digram this is the order of priority that used :
* + priority = 1;
& (concatenation) priority = 2;
| priority = 3;

and here is the code of it :

```
string infix2postfix(string infix)
{
   stack<char> operator_stack;

   stringstream output;

   for (unsigned i = 0; i < infix.length(); i++)
   {
      if (infix[i] == '+' || infix[i] == '&' || infix[i] == '*' || infix[i] == '|' )
      {
         while (!operator_stack.empty() && priority(operator_stack.top()) <= priority(infix[i]))
         {
            output << " "<<operator_stack.top()<<" ";
            operator_stack.pop();
         }
         operator_stack.push(infix[i]);
      }
      else if (infix[i] == '(')
      {
         operator_stack.push(infix[i]);
      }
      else if (infix[i] == ')')
      {
         while (operator_stack.top() != '(')
         {
            output << " "<<operator_stack.top()<<" ";
            operator_stack.pop();
         }
         operator_stack.pop();
      }
      else
      {
         output <<infix[i];
      }
   }

   while (!operator_stack.empty())
   {
      output << " "<<operator_stack.top()<<" ";
```

```
        operator_stack.pop();
    }

    return output.str();
}
```

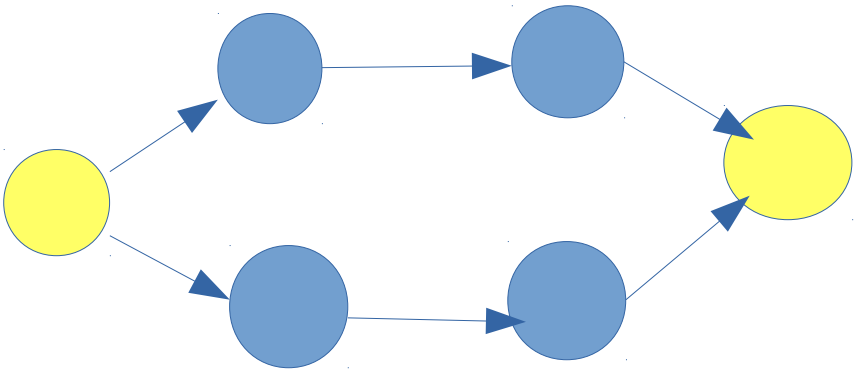
---

create NFA Table :

this function used to create nfa table void createTableNFA(char s[], string accept)
get from the postfix expression word and if it is not a + * & | then we will create a new node with this class and make this node point to next one and push it on the stack .
if we found one of this characters + * & |

```
if(str.compare("|") == 0)
        {
           // pop first element from the stack
           e1.startNode = stackOfElements.top().startNode;
           e1.lastNode = stackOfElements.top().lastNode;
           e1.grammer = stackOfElements.top().grammer;
           e1.done = stackOfElements.top().done;
           stackOfElements.pop();

           // pop second element from the stack
           e2.startNode = stackOfElements.top().startNode;
           e2.lastNode = stackOfElements.top().lastNode;
           e2.grammer = stackOfElements.top().grammer;
           e2.done = stackOfElements.top().done;
           stackOfElements.pop();

           makeOr(e1,e2); //  make the or between them }
```

we make the or between by add two more nodes (yellow nodes in the chart) and make the first one point to this to node with lamda and make the two nodes point to the second with lamda



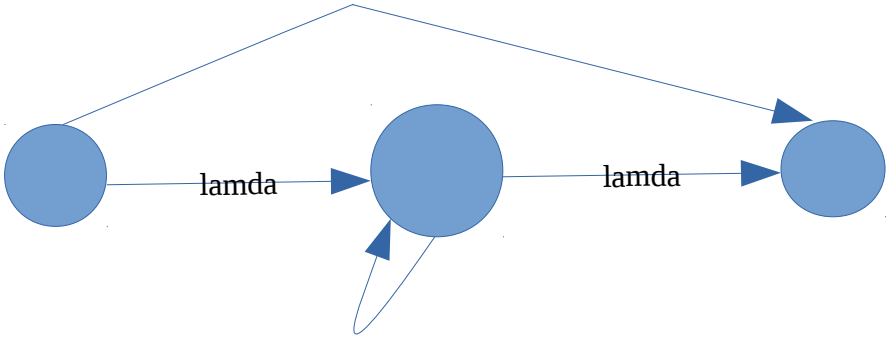
```
else if (str.compare("&") == 0)
        {
           // pop the first node from the stack
           e2.startNode = stackOfElements.top().startNode;
           e2.lastNode = stackOfElements.top().lastNode;
           e2.grammer = stackOfElements.top().grammer;
```

```
e2.done = stackOfElements.top().done;
stackOfElements.pop();

// pop the second node from the stack
e1.startNode = stackOfElements.top().startNode;
e1.lastNode = stackOfElements.top().lastNode;
e1.grammer = stackOfElements.top().grammer;
e1.done = stackOfElements.top().done;
stackOfElements.pop();

// make and
makeAnd(e1, e2);}
```

we make and by make the first node pionts to the second with lamada

```
else if (str.compare("*") == 0  && stackOfElements.size() > 0 )
    {
      // pop the element from the stack
      e1.startNode = stackOfElements.top().startNode;
      e1.lastNode = stackOfElements.top().lastNode;
      e1.grammer = stackOfElements.top().grammer;
      e1.done = stackOfElements.top().done;
      stackOfElements.pop();
      makeAstric(e1);}
```

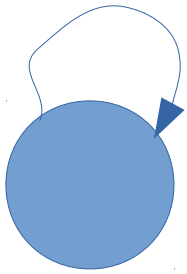
this is a unary operation so we pop one element and add more two nodes first one points to the the node with lamda and the node points to the second one with lamda and the first point to the second with lamada and the node points to itselt with lamda




```
else if (str.compare("+") == 0 && stackOfElements.size() > 0 )
```

```
{
   // pop the first element of stack
   e1.startNode = stackOfElements.top().startNode;
   e1.lastNode = stackOfElements.top().lastNode;
   e1.grammer = stackOfElements.top().grammer;
   e1.done = stackOfElements.top().done;
   stackOfElements.pop();
   // make the plus
   makePlus(e1);}
```

this a unary operation and take first element only from the stack make the node points to itself with lamda



here is the code of the createTableNFA function :

```
void createTableNFA(char s[], string accept)
{
   std::stack<element> stackOfElements;
   element e1;
   element e2;
   element e3 ;
   char *pch;

   pch = strtok (s," ");

   while (pch != NULL)
   {
      std::string str(pch);

      if(str.compare("|") == 0)
      {
         e1.startNode = stackOfElements.top().startNode;
         e1.lastNode = stackOfElements.top().lastNode;
         e1.grammer = stackOfElements.top().grammer;
         e1.done = stackOfElements.top().done;
         stackOfElements.pop();

         e2.startNode = stackOfElements.top().startNode;
         e2.lastNode = stackOfElements.top().lastNode;
         e2.grammer = stackOfElements.top().grammer;
         e2.done = stackOfElements.top().done;
         stackOfElements.pop();

         checkIfDone(e1,e2);
```

```
    makeOr(e1,e2);


    e3.startNode = globalCounter-1;
    e3.lastNode = globalCounter;
    e3.done = true;

    stackOfElements.push(e3);

}
else if (str.compare("&") == 0)
{
    e2.startNode = stackOfElements.top().startNode;
    e2.lastNode = stackOfElements.top().lastNode;
    e2.grammer = stackOfElements.top().grammer;
    e2.done = stackOfElements.top().done;
    stackOfElements.pop();

    e1.startNode = stackOfElements.top().startNode;
    e1.lastNode = stackOfElements.top().lastNode;
    e1.grammer = stackOfElements.top().grammer;
    e1.done = stackOfElements.top().done;
    stackOfElements.pop();

    checkIfDone(e1,e2);
    makeAnd(e1, e2);

    e3.startNode = e1.startNode;
    e3.lastNode = e2.lastNode;
    e3.done = true;

    stackOfElements.push(e3);

}
else if (str.compare("*") == 0  && stackOfElements.size() > 0 )
{
    e1.startNode = stackOfElements.top().startNode;
    e1.lastNode = stackOfElements.top().lastNode;
    e1.grammer = stackOfElements.top().grammer;
    e1.done = stackOfElements.top().done;
    stackOfElements.pop();

    checkIfDone(e1);
    makeAstric(e1);

    e3.startNode = globalCounter-1;
    e3.lastNode = globalCounter;
    e3.done = true;

    stackOfElements.push(e3);

}
```

```
        else if (str.compare("+") == 0 && stackOfElements.size() > 0 )
        {
           e1.startNode = stackOfElements.top().startNode;
           e1.lastNode = stackOfElements.top().lastNode;
           e1.grammer = stackOfElements.top().grammer;
           e1.done = stackOfElements.top().done;
           stackOfElements.pop();

           checkIfDone(e1);
           makePlus(e1);

           e3.startNode = e1.startNode;
           e3.lastNode = e1.lastNode;
           e3.done = true;

           stackOfElements.push(e3);

        }
        else
        {
           element temp1;
           temp1.grammer = str;
           temp1.startNode = ++globalCounter;
           temp1.lastNode = ++globalCounter;
           stackOfElements.push(temp1);
        }

        pch = strtok (NULL, " ");

     }

     // empty the stack
     e1.startNode = stackOfElements.top().startNode;
     e1.lastNode = stackOfElements.top().lastNode;
     e1.grammer = stackOfElements.top().grammer;
     e1.done = stackOfElements.top().done;
     stackOfElements.pop();

     checkIfDone(e1);

     pair <int, string> p2 (e1.lastNode, accept );
     acceptor.push_back(p2);

     starters.push_back(e1.startNode);


}
```

DFA table :

This is a recursive method which find the epsilon closure of all the nodes and put them in epsilon closure vector.

```
vector<int> find_e_closure(int state)
{
    vector<int> result
    vector<int> result2
    vector<int>  state_eclosure
    int temp
    auto it <- NFA_table.find(make_pair(state, "lamda"))
    IF(it != NFA_table.end())

        state_eclosure <- it -> second
    ELSE

        //base case, if no e-closure found return
        return result

    FOR(int i = 0 up to state_eclosure.size())
    {
        temp <= state_eclosure.at(i)
        result.push_back(temp)
        if(e_closure.at(temp).second = false)
        {
            result2 <= find_e_closure(temp)
            e_closure.at(temp).second <= true
        }
        else
        {
            result2 = e_closure.at(temp).first
        }

        result.insert(result.end(), result2.begin(), result2.end())
        e_closure.at(state).first = result
    }
    return result
}
```

This method is used to check whether the new state calculated is already found before or not.

```
int check_repeated_state(vector<int> new_state)
{
    int counter = 0
    FOR(int i = 0 up to states.size())
        IF(new_state.size() = states.at(i).size())

            FOR(int k = 0 up to new_state.size())
                FOR(int n = 0 up to new_state.size())
                    IF(new_state.at(k) = states.at(i).at(n))
                        counter++
                        break
            IF(counter = new_state.size())

                return i

            counter = 0

    return -1
}
```

The method used to construct the DFA table:

```
void consruct_DFA_table(vector<int> vect)
{
    vector<int> new_state
    vector<int> temp
    vector<int> temp2
    FOR(int j = 0 up to classes.size())
        FOR(int i = 0 up to vect.size())
            auto it <= NFA_table.find(make_pair(vect.at(i), classes.at(j).first))
            IF(it != NFA_table.end())
                temp = it -> second
            FOR(int k = 0 up to temp.size())
            {
                new_state.push_back(temp.at(k))
                temp2 <= e_closure.at(temp.at(k)).first
                new_state.insert(new_state.end(), temp2.begin(), temp2.end())
            }
            temp.clear()

        int repeated <= check_repeated_state(new_state)
        IF(repeated != -1)
            DFA_table.insert(std::pair<pair<int,string>, int>
                             (make_pair(counter,  classes.at(j).first), repeated))
            new_state.clear()
        ELSE
            IF(new_state.size() != 0)
                DFA_table.insert(std::pair<pair<int,string>, int>
                                 (make_pair(counter,  classes.at(j).first),  states.size()))
                states.push_back(new_state)
                new_state.clear()
    counter++
}
```

```
void find_DFA()
{
    FOR(int i = 0 up to globalCounter+1)
        find_e_closure(i)
    vector<int> e_closure0 <= e_closure.at(0).first
    e_closure0.push_back(0)
    states.push_back(e_closure0)
    int old_size <= states.size()
    consruct_DFA_table(states.at(0))
    int new_size <= states.size()
    int temp
    WHILE(new_size - old_size > 0)
        temp <= states.size()
        FOR(int k = up to new_size)
            consruct_DFA_table(states.at(k))
        old_size <= temp
        new_size <= states.size()
    END WHILE
}
```

Minimization of DFA :

1- find_terminals function :
in this function we iterate over all states , find the terminals state and put them in **terminals** map to use them in minimization.
2- delete_repeated_states function :
in this function we delete the repeated states in different minimization groups based on priority.
3- collect_minimization_groups function :
group all terminals and non terminals groups in **minimization_gps.**
4- minimize_DFA function :
- iterate over the **minimization_gps** vector , for each group :
- iterate over all the classes you have (ex id,digit ..etc)
- for each state in the held **minimization_gps** search the DFA_table for node from this state under the held input class .
If you find another state this state goes to with the input of the class
then push this state in output vector
else
push -1 which indicate phai state.
- call partition_class function and pass to it the group of states and the states they go to under the input class .
5- partition_classes function :
- in this function groups is partitioned based on the states they go under each input class if they are in the same group in all input cases then the original states stay in the same group but if there exist some states in other groups the original group is partitioned based on that .
- iterate over all the groups ,if you find the state in one of them , put it in **gps** vector

such that the index indicate the group number which the states belong to.
- after finishing that and finding that the given states should be in different groups iterate over gps vector and put each new partitioned group as a new group in **minimization_gps** and mark the original group as deleted by setting the boolean associated with it as false.

Lexical analyzer output :

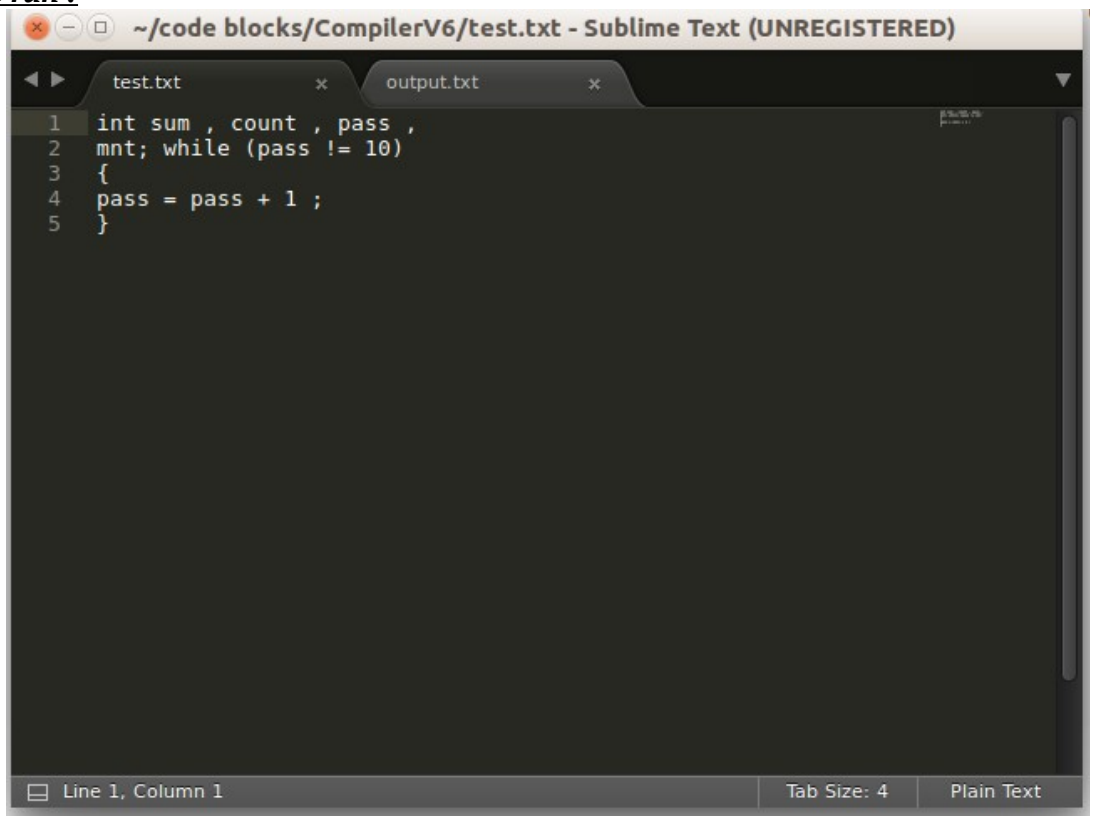
1- read the characters from file character by character
2- get the class of the read character
3- starting from the current state (0 state for first time | on reset)
4- from the current state and class number of the read character go to the next state
5- check if the current state accept any output (token)
6- if it accept continue to first point if not check for the longest accept
7- if it has no formal acceptance state the error

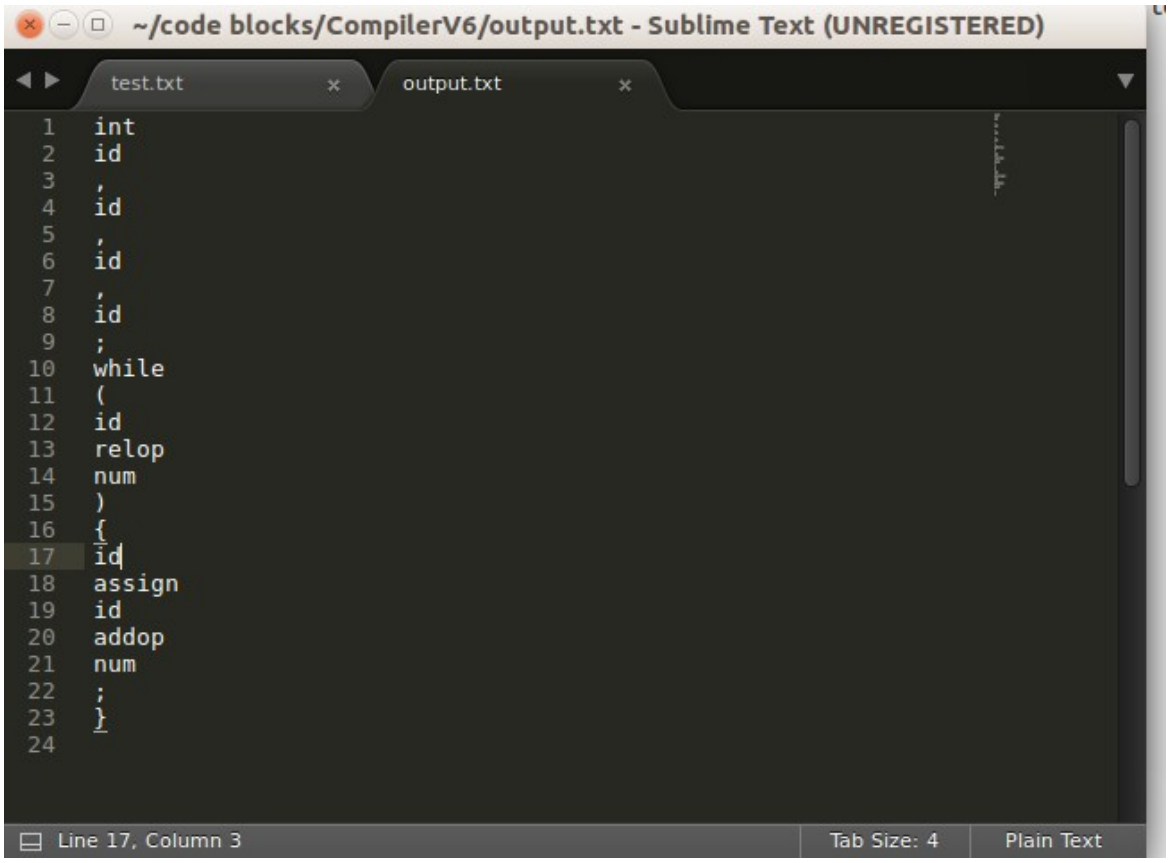
```
void runC ()
{
   ifstream inputFile("test.txt");     // open file

   char c;-
   int classNum = -1;
   int currentState = globalStart;
   vector < vector<string> > walkingAcceptor;
   int i;
   int condition = inputFile.peek();
   string currentLexim = "";

   inputFile.get(c);


   while (condition != -1)       // loop getting single characters
   {
      classNum = findClass(c);       //get class number

      if(classNum != -1)
      {

         currentState = table[currentState][classNum];

         vector<string> currentAccept;

         for( i = 0 ; i < finalAcceptor.size(); i++)
         {
            if(currentState == finalAcceptor[i].first)
            {
               currentAccept.push_back(finalAcceptor[i].second);  //ana b accept eh
            }
         }
```

```
        if(currentAccept.size() == 0)
        {

            if(walkingAcceptor.size() != 0 && currentState == numberOfStates)  // ana msh f awal
wa7da
            {

                if((walkingAcceptor.back())[0] == "id")
                {
                    sTable.push_back(currentLexim);
                }
                currentLexim="";

                cout << (walkingAcceptor.back())[0] << "\n";

                currentState = globalStart;
                walkingAcceptor.clear();

            }
            else if( currentState != numberOfStates )
            {
                currentLexim += c;
                condition = inputFile.peek();
                inputFile.get(c);
            }



        }
        else
        {
            currentLexim += c;
            walkingAcceptor.push_back(currentAccept);
            condition = inputFile.peek();
            inputFile.get(c);
        }

    }
    else
    {

        if(walkingAcceptor.size() != 0)  // ana msh f awal wa7da
        {
            currentLexim += c;
            if((walkingAcceptor.back())[0] == "id")
            {
                sTable.push_back(currentLexim);
            }
            cout << (walkingAcceptor.back())[0] << "\n";

        }
```

```
            else
            {

                if(!(c == ' '||c == '\n'||c == '\t'))
                {
                    cout << c << " error \n";
                }

                condition = inputFile.peek();
                inputFile.get(c);
            }

            currentLexim="";
            currentState = globalStart;
            walkingAcceptor.clear();


        }
    }

    if(walkingAcceptor.size() != 0)
    {
        cout << (walkingAcceptor.back())[0] << "\n";

        if((walkingAcceptor.back())[0] == "id")
        {
            sTable.push_back(currentLexim);
        }

    }

    inputFile.close();              // close file

    cout << "\n\nsTable" << "\n";
    for(int i =0;i<sTable.size();i++)
    {
         cout << sTable[i] << "\n";
    }
}
```

Sample run :


~/code blocks/CompilerV6/test.txt - Sublime Text (UNREGISTERED)
test.txt
output.txt
1 int sum , count , pass ,
2 mnt; while (pass != 10)
3 {
4 pass = pass + 1 ;
5 }
Line 1, Column 1
Tab Size: 4
Plain Text


~/code blocks/CompilerV6/output.txt - Sublime Text (UNREGISTERED)
test.txt
output.txt
1 int
2 id
3 ,
4 id
5 ,
6 id
7 ,
8 id
9 ;
10 while
11 (
12 id
13 relop
14 num
15)
16 {
17 id
18 assign
19 id
20 addop
21 num
22 ;
23 }
24
Line 17, Column 3
Tab Size: 4
Plain Text

The resultant transition table for the minimal DFA :

see attached file minimized.txt ...

Assumption made in both DFA_table and Minimized :

1 - if the node doesn't go to any other node by all input we don't put it in the map of DFA_table or Minimized_dfa_table.

2 - phai isn't included as a state in our tables.