COMPILERS
LEXICAL ANALYZER GENERATOR
28/3/2020

DATA STRUCTURES

1. Node

Node is a class used to represent states in both NFA and DFA and it consists of:

- A String to save name of the state.
- An Integer to save value of the priority of this state if it's final or zero it it's not final.
- Vector of pointers of edges which contain all edges that go out from this state.

2. Edge

Edge is a class used to represent transitions between states in both NFA and DFA and it consists of:

- A Pointer to destination node.
- Two characters: start and end alphabets that are the range of the allowed input transitions.
- A Set of characters that are disallowed to be the input of a transition by this edge.

3. NFA

NFA is a class to represent the NFA as a graph and it has:

- Two pointers on start and end nodes of the NFA graph.
- Set of pointers on nodes for all final states in this NFA.

4. DFA

DFA is a class to represent the DFA as a graph and it has:

- A Pointer to the start state node.
- A map its key is pointer on the DFA state and its value is another map, this map of key character “the input symbol” and the value is a pointer to a node “2nd DFA state which accessed by that input symbol”.

5. We defined the following helping enums:

- `LexicalType`: ***`RegularExpression`***, ***`Keyword`***, ***`Punctuation`***, ***`RegularDefinition`***.
- `LexicalTermType`: ***`Operation`***, ***`CharGroup`***, ***`WORD`***, ***`parenthesis`***, ***`EPSILON`***.

ALGORITHMS

1. Build NFA from RE

To convert regular expressions to NFA; Thompson’s construction algorithm is used.

So, there some main procedures to do this construction.

1. *Read the Lexical Rules from the file and parsing them using **ReadLexicalRulesFile** class.*
2. *Convert the parsing Lexical Rules to Postfix expressions using **LexicalRuleBuilder** class.*
3. *Build the NFA for each Postfix Lexical Rule using **Builder** class.*

   This class applies the Thompson’s construction algorithm Rules such as

```
NFA* buildLetterRecognizer(char letter);
NFA* buildEPSRecognizer();
NFA* buildAlphabetRecognizer(char startAlphabet , char endAlphabet);
NFA* buildORRecognizer(NFA* recognizer1, NFA* recognizer2);
NFA* buildCombineRecognizer(NFA* recognizer1, NFA* recognizer2);
NFA* buildANDRecognizer(NFA* recognizer1, NFA* recognizer2);
NFA* buildClosureRecognizer(NFA* recognizer);
NFA* buildPositiveClosureRecognizer(NFA* recognizer);
```

2. Converter from NFA to DFA

To convert from NFA to DFA, Subset Construction Algorithm is used.
So there is one main function to convert it and two other sub functions.

- *Closure: It takes a set of nodes and return a set of all nodes that can be reached by epsilon transition.*

```
/// Set of Nodes "NFA states" reachable from some NFA state in nodes "NFA states" on ε-transitions alone
set<Node *> Converter::closure(set<Node *> nodes) {
    int i = 0;
    set<Node *>::iterator itr;
    while (nodes.size() != i) {
        i = 0;
        for (itr = nodes.begin(); itr != nodes.end(); itr++)   // loop on all nodes
        {
            i++;
            vector<Edge *> edges = (*itr)->getEdges();
            Edge *x;
            for (auto &edge : edges)   // loop on edges of each node
            {
                x = edge;
                if (x->isEPSTransition()) {
                    nodes.insert(x->getDestination());
                }

            }
        }
    }
    return nodes;
}
```

- *Move: it takes a set of nodes and a char symbol and return a set of all nodes that can be reached by this symbol transition.*

```
/// Set of Nodes "NFA states" to which there is a transition on input char symbol from some state s in nodes.
set<Node *> Converter::move(const set<Node *>& nodes, char symbol){
    set<Node *>::iterator itr;
    set<Node *> res;
    for (itr = nodes.begin(); itr != nodes.end(); itr++)   // loop on all nodes
    {
        vector<Edge *> edges = (*itr)->getEdges();
        Edge *x;
        for (auto &edge : edges)   // loop on all edges of each node
        {
            x = edge;
            if (x->isAcceptSymbol(symbol))
            {
                res.insert(x->doTransition(symbol));
            }
        }
    }
    return res;
}
```

- *Convert: It takes a NFA and a set of alphabets used and return a DFA.*
  while (there is an unmarked DFA state in mark) {
  mark DFA state = true;
  for (each input symbol c in alphabet) {

```
        u = closure(move(T, a));
        D_table[DFA state, c] = u;
        if (U is not in DFA states) {
            add u as unmarked state to DFA states;
        }
      }
    }
```

So while building D_table, DFA graph is being built to.

3. DFA Minimizer

After converting to DFA we need to minimize the resulted DFA.

There is main function that uses other five sub-functions to minimize the DFA and get the minimized table.

- *DFAMinimize():*
    - *it takes the DFA before minimizing.*
    - *Divide the DFA into two partitions (Final and non-Final).*
    - *Put the final and non-final into vector of vectors of nodes (partitions) then send them to Minimize function.*

```
// minimizer function
void Minimizer::DFAMinimize(DFA *dfa) {

    this->partitions.clear();
    // get dfa before minimizing
    DFAStates = dfa->getDTable();
    // loop on these states to differentiate between final and non final
    // put the final and non final in two vectors
    vector<Node*> FinalStates;
    vector<Node*> NonFinalStates;
    map <Node* , map<char , Node*>>::iterator itr;
    for (itr = DFAStates.begin(); itr != DFAStates.end(); itr++){
        if (itr->first->isFinalState()){
            FinalStates.push_back(itr->first);
        } else {
            NonFinalStates.push_back(itr->first);
        }
    }

    // the final and non final put them in queue of vector of nodes
    partitions.push_back(NonFinalStates);
    partitions.push_back(FinalStates);

    // minimize the table
    Minimize(partitions);
    dfa->setDTable( dTable: DFAStates);
}
```

- *Minimize():*
    - *Recursive function that takes partitions and make minimization on it.*
    - *Uses three sub functions (containState, areStatesUnique, and updateTable).*
    - *Create temporary vector of nodes, loop on nodes of the partitions and check if the temporary contains this state. If yes, then skip.*
    - *Else, add thus state to the temporary then compares with the other states if they are unique. If they are unique, then add the second state to the temporary and after ending the loop send the temporary to subfunction updateTable. If they are not unique, check again if they have the same next state under the same inputs.*
    - *Check if the partitions from the resulted updated table equal the partitions before updating. If yes, then return the result. Else, do recursion.*

```
/*
 * minimize the results
 */
vector<vector<Node *>> Minimizer::Minimize(const vector<vector<Node *>>& partitions) {

    vector<vector<Node*>> res;

    for (vector<Node*> p:partitions){
        if (p.size() > 1){
            for (int i = 0; i < p.size(); i++){

                // if already in partition
                if (containState(res, p.at(i))){
                    continue;
                }
                vector<Node*> temp;
                temp.push_back(p.at(i));

                for (int j = i + 1; j < p.size(); j++){
                    Node* s1 = p.at(i);
                    Node* s2 = p.at(j);
```

```
            if ((!areStatesUnique(partitions, s1, s2)) && DFAStates.find(s1) != DFAStates.end() && DFAStates.find(s2) != DFAStates.end()){

                if(DFAStates.at(s1) == DFAStates.at(s2)){
                    if(!s1->isFinalState()) {
                        temp.push_back(s2);
                    } else if(s1->getName() == s2->getName()){
                        temp.push_back(s2);
                    }
                }
            }
        }
        res.push_back(temp);
        updateTable(temp);
    }
  }else{
    // when group contain one state only
    res.push_back(p);
  }
}
// if already minimized
if (partitions.size() == res.size()){
  return res;
}

return Minimize(res);
```

- *containState():*
    - *Takes vector of vectors of nodes and Node.*
    - *Check if partitions contain this Node.*

```
/*
 * return true if partition contain state
 */
bool Minimizer::containState(const vector<vector<Node*>>& partitions, Node* state) {

    for (const vector<Node*>& p:partitions){
        for (Node* N:p){
            if (N == state){
                return true;
            }
        }
    }
    return false;
}
```

- *areStatesUnique() & containedBySamePartition():*
    - *Takes vector of vectors of nodes and two Nodes.*
    - *Check if they are in the same partition by sending these attributes to containedBySamePartition().*

```
/*
 * return true if two states are unique
 */
bool Minimizer::areStatesUnique(const vector<vector<Node*>>& partitions, Node *state1, Node *state2) {
    for (const vector<Node*>&p:partitions){

            if (!containedBySamePartition(partitions, state1, state2)){
                return true;
            }
    }
    return false;
}

/*
 * return true if two states are in the same partition
 */
bool Minimizer::containedBySamePartition(const vector<vector<Node *>>& partitions, Node *State1, Node *State2) {

    for (vector<Node*> p:partitions){
        if ((std::find(p.begin(), p.end(), State1) != p.end()) && (std::find(p.begin(), p.end(), State2) != p.end())){
            return true;
        }
    }
    return false;
}
```

- *uddateTable():*
    - *Takes temporary vector of nodes to remove the similar states in the same group and update the table to be minimized.*

```
/*
 * update table of the DFAStates
 */
void Minimizer::updateTable(vector<Node *> temp) {
    if (temp.size() > 1){
        Node* state = temp.at( n: 0);
        for (int i = 1; i < temp.size(); i++){
            DFAStates.erase(temp.at(i));
        }
        for (pair<Node *, map<char, Node *>> row:DFAStates){
            for (pair<char, Node*> c:row.second){
                for(int i = 1; i < temp.size(); i++){
                    if (temp.at(i) == c.second){
                        DFAStates.at(row.first).at(c.first) = state;
                    }
                }
            }
        }
    }
}
```

4. Backtracking in Scanner

We used Backtracking technique in Scanner that is the Scanner starts moving with the input in the graph until reaching an invalid state or the input ends then it backtracks until *reaching the last acceptance state according to Maximal Munch.*

```
vector<pair<string, string>> Scanner::scanWord(string &word) {
    vector<pair<string, string>>tokens{};
    Node* startState = this->recognizer->getStartState();
    Node* currentState = startState;
    Node* nullState = new Node( name: "null",  isFinal: false);
    Node* finalState = nullState;
    int first = 0;
    int last = 0;
    for(int i=0;i<word.size();i++){
        currentState = this->recognizer->move(currentState, word.at(i));
        if(currentState->isFinalState()){
            last = i;
            finalState = currentState;
            if(i == word.size() - 1){
                string s = word.substr(first,  n: last - first + 1);
                tokens.emplace_back(s,finalState->getName());
                break;
            }
        }else if(currentState->getName() == "null" && finalState != nullState){
            string s = word.substr(first,  n: last - first + 1);
            tokens.emplace_back(s,finalState->getName());
            first = last + 1;
            i = last;
            finalState = nullState;
            currentState = startState;
        }
    }
    return tokens;
}
```

THE RESULTANT TRANSITION TABLE FOR THE MINIMAL DFA

Fully spreadsheet is found here for the resultant transition table:

https://docs.google.com/spreadsheets/d/1rfcjfy03df_5y6WCKP1_zNSzLUGQYwlkRFo_NKmKJcs/edit?usp=sharing

states\input	!	&	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<
;	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
relop	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
:	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
num	null	null	null	null	null	null	null	null	h	null	num	num	num	num	num	num	num	num	num	num	null	null	r
addop	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
(	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
*	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
,	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
mulop	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
)	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
A	B	C	(	)	*	addop	,	addop	null	mulop	num	num	num	num	num	num	num	num	num	num	:	;	r
B	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
C	null	logop	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
relop	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
assign	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
id	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r
Z	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
{	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
}	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	null	r
if	null	null	null	null	null	null	null	null	null	null	id	id	id	id	id	id	id	id	id	id	null	null	r

STREAM OF TOKENS

```
int sum , count , pass , mnt;
while ((pass != 10 && count <= sum) || pass != mnt)
{
pass = pass + 1 ;
}
```

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

Lexical_Analyzer_Generator

```
int
id
,
id
,
id
,
id
;
while
(
(
id
relop
num
logop
id
relop
id
)
logop
id
relop
id
)
{
id
assign
id
addop
num
;
}
```

ASSUMPTIONS

1. Priorities of Accepted States:

We assumed that accepted states have the following priorities from high to low (punctuation >> keywords >> regular expression) which is followed by most of the popular programming languages.

2. All Reserved character must be escaped:

Any reserved character should always be escaped before usage, otherwise it would be considering it as an operation if suitable.

Reserved characters are {‘+’, ‘-’, ‘*’, ‘|’, ‘(‘, ‘)’}

3. Grouping Characters:

Characters are grouped by their numerical values, so saying ‘a-z’, would lead to considering any character between the numeric values of a to z to be included. That would lead to us rejecting any character group where the ending character has a numeric value less than the starting character.

4. Regular Expressions Definition:

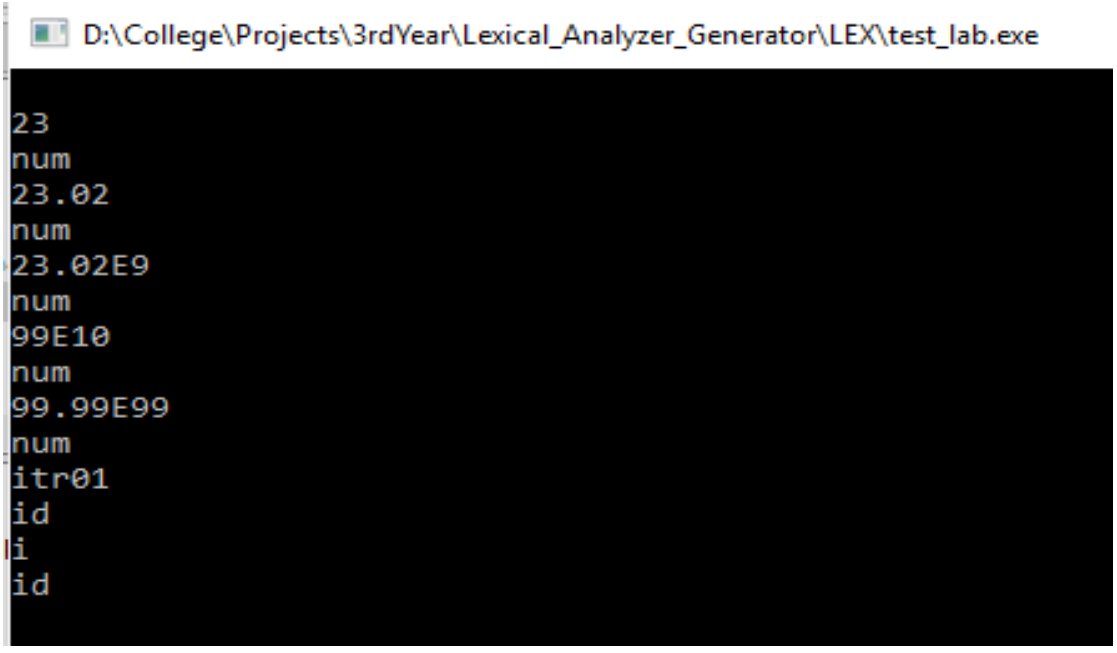
Regular definitions can be used in multiple regular expressions or definitions, however regular expressions can’t be reused to define another regular expression or definition. Also, a regular definition will only be replaced if it was already defined before being referenced, or it will consider it a normal word.

BONUS

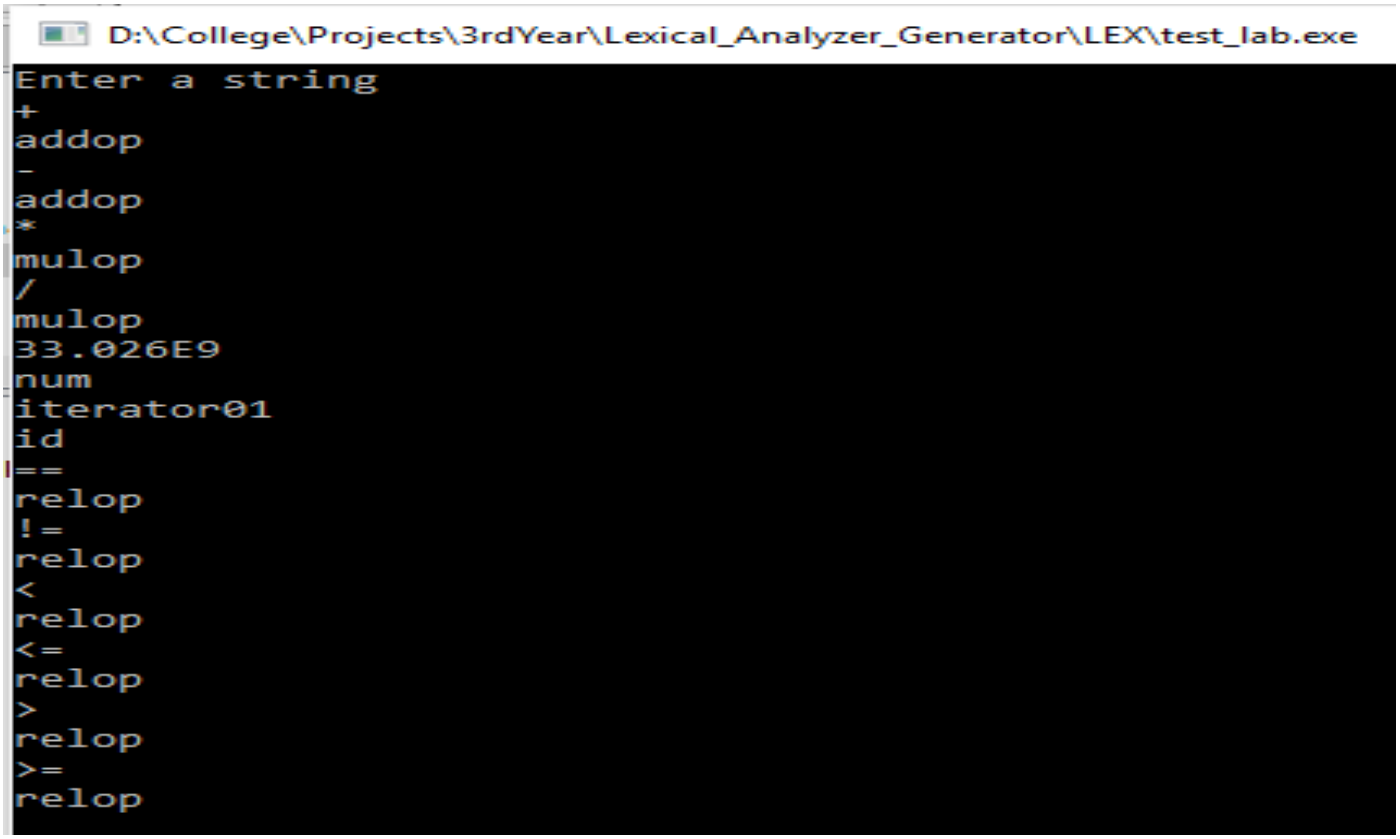
- **CODE**

```
%{
#include<stdio.h>
%}

%%
[a-z|A-Z]([a-z|A-Z|0-9])* {printf("id\t");}
[0-9]+(.[0-9]+(L|E[0-9])?)? {printf("num\t");}
"=" {printf("assign\t");}
"=="|"!="|">"|">="|"<"|"<=" {printf("relop\t");}
"+"|"-" {printf("addop\t");}
"*"|"/" {printf("mulop\t");}
%%

int yywrap()
{
return 1;
}

main()
{
printf("Enter a string\n");
yylex();
}
```

- **EAMPLE 1**



- **EAMPLE 2**



TEAM MEMBERS

Name	ID
Arsany Atef Abdo	10
Kirellos Malak Habib	35
Michael Said Beshara	38
Yomna Gamal El-Din Mahmoud	63