Compilers  
Lexical Analyzer Generator

28/3/2020

# data structures

## 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.

## 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.

## 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.

## 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”.

## We defined the following helping enums:

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

# Algorithms

## 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 |

## 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. * 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. * 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.   DFA \*Converter::convert(NFA \*nfa, const set<char> &alphabet) {  set<Node \*> nfaState ;  map <set<Node\*>,Node\*> ndStates;  map <Node\*,bool> mark;  map <Node\* , map<char , Node\*>> Dtable;  int i = 0;  nfaState.insert(nfa->getStart());  nfaState = closure(nfaState);  Node \* dfaState = new Node("A",0);  for (auto f:nfa->getFinalStates()) {  if (nfaState.find(f) != nfaState.end() && (f->isFinalState() > dfaState->isFinalState())) {  dfaState->setIsFinal(f->isFinalState());  dfaState->setName(f->getName());  }  }  Node\* startState = dfaState;  ndStates.insert(pair<set<Node\*>,Node\*>(nfaState,dfaState));  mark.insert(pair<Node\*,bool>(dfaState,false));  int flag = 0;  int bsize ;  do {  bsize = ndStates.size();  flag = 0;  for (const auto& itr:ndStates) // loop on all states we get  {  nfaState = itr.first;  dfaState = itr.second;  if (!mark.find(dfaState)->second) {  flag--;  if (mark.find(dfaState) != mark.end()){  mark.find(dfaState)->second = true;  }  auto \*drow = new map<char, Node \*>();  drow->clear();  map <char,Node\*>::iterator it2;  for (auto c:alphabet) // loop on each input symbol  {  drow->insert(pair<char,Node\*>(c,nullNode));  if (move(nfaState, c).empty()) {  continue;  }  set<Node \*> u = closure(move(nfaState, c));  bool found = false;  for (const auto& itr2:ndStates) {  if (compareSets(u,itr2.first)) {  if (drow->find(c) != drow->end()){  drow->find(c)->second = itr2.second;  }  Edge\* nedge = new Edge(itr2.second, c, c);  dfaState->addEdge(nedge);  found = true;  break;  }  }  if (!found) {  i++;  std::string s(1, 'A'+ i);  Node \*d = new Node(s, false);  d->setName(s);  if (drow->find(c) != drow->end()){  drow->find(c)->second = d;  }  Edge \* nedge = new Edge(d, c, c);  dfaState->addEdge(nedge);  ndStates.insert(pair<set<Node \*>, Node \*>(u, d));  mark.insert(pair<Node \*, bool>(d, false));  }  }  Dtable.insert(pair<Node \*, map<char, Node \*>>(dfaState, \*drow));  }  for (auto f:nfa->getFinalStates()) {  if (nfaState.find(f) != nfaState.end() && (f->isFinalState() > dfaState->isFinalState())) {  dfaState->setIsFinal(f->isFinalState());  dfaState->setName(f->getName());  }  }  }  } while (flag != 0 || bsize != ndStates.size());   return new DFA(startState, Dtable); } |

## DFA Minimizer

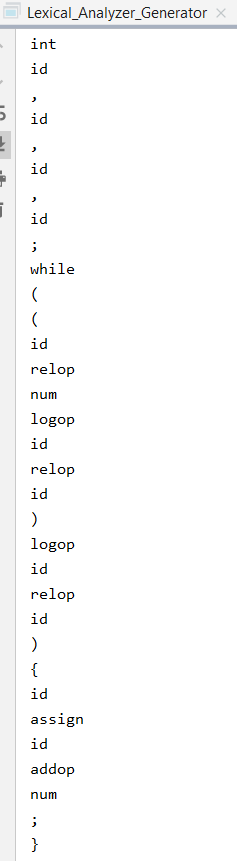
|  |  |
| --- | --- |
|  | After converting to DFA we need to minimize the resulted DFA.  There is main function that uses other five sub-functions to minimize the DFA and get the minimized table.   * DFAMinimize(): * it takes the DFA before minimizing. * Divide the DFA into two partitions (Final and non-Final). * Put the final and non-final into vector of vectors of nodes (partitions) then send them to Minimize function. * 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.      * containState(): * Takes vector of vectors of nodes and Node. * Check if partitions contain this Node. * 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(). * uddateTable(): * Takes temporary vector of nodes to remove the similar states in the same group and update the table to be minimized. |

## 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. |

# minimal dfa table

# stream of tokens



# assumptions

## 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.

## 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 {‘+’, ‘-’, ‘\*’, ‘|’, ‘(‘, ‘)’}

## 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.

## 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**

# Team members

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