ECOTE final project

Semester: Summer 2020/2021

Author: Paweł Borsukiewicz, 301106

Subject: ECOTE

Write a program reading regular expressions, then constructing NFA using Thompson algorithm, converting NFA into DFA and checking if input strings are generated by this expression (working on DFA).

I. General overview and assumptions

The aim of the project is to create a program that takes as an input RE in a form of a string. Given string is checked if it is correct. In case of some mistakes such as unequal number of left and right brackets user is informed that the mistake was made.

Having confirmed that RE is correct, program generates NFA and displays it in a form of a graph. Then, similarly from NFA it generates DFA and displays it. Finally user has an opportunity to check if particular strings are generated by the RE. Check is performed on the final DFA.

II. Functional requirements

The RE is provided by the user during execution of the program. So as to make program less complex my input alphabet is a set of small letters (a-z ascii symbols 97-122) and epsilon symbol replaced by big E (ascii symbol 69) due to the fact that ε is not a part of basic ascii table.

Following operators are to be allowed: “|”, “*”, “(”, “)”. Correctness of their usage will also be checked before execution of transformation code.

RE will be transformed into postfix notation so as to facilitate the process of obtaining NFA as it more clearly states the order of operations to make. The main assumption contrary to algebraic notation, where operator is between the operands, is that in postfix notation the operator follows its operands. For instance let’s take mathematic example, in algebraic

notation we may have 2+3*4, while in postfix notation it will be converted to 234*+, where first operator is * meaning that we have to multiply, then we can add result of multiplication to 2. Similarly, REs can undergo such conversion. For example a|b* may be transformed to ab*|.

NFA will be obtained using Thompson algorithm. It exactly specifies methods of obtaining NFA subparts for "|", "*" and concatenation, hence RE will be composed of smaller parts. Such subparts will be transformed to NFA and later glued together to make one coherent NFA.

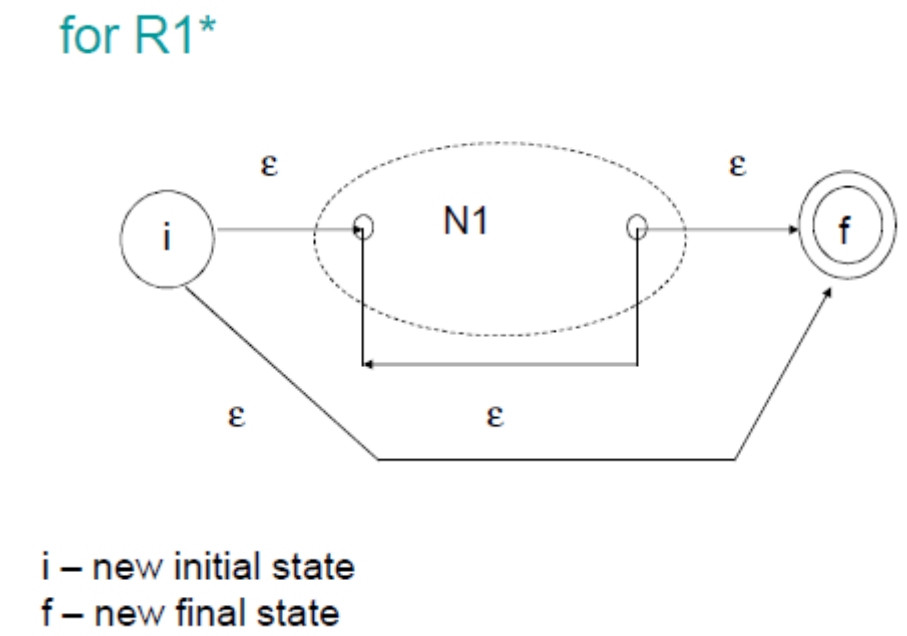

Figure 1. Thompson algorithm for R1*, taken from lecture slides

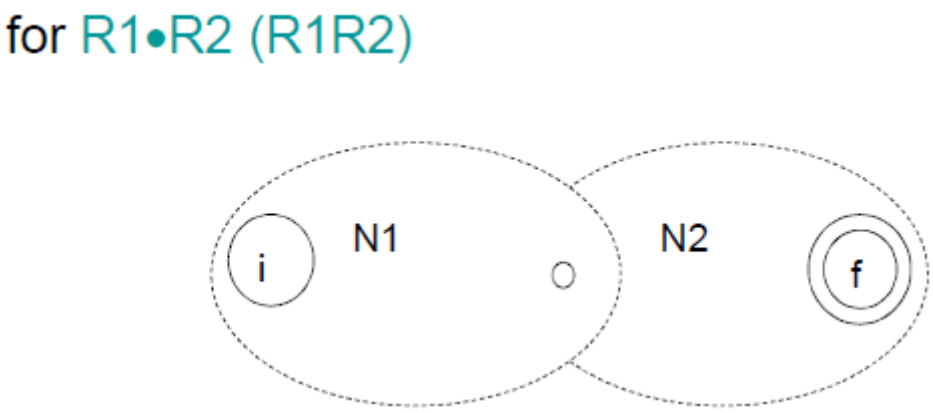

Figure 2. Thompson algorithm for R1R2, taken from lecture slides

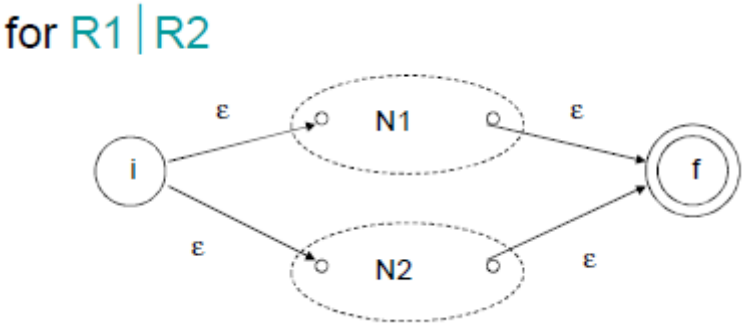

Figure 3. Thompson algorithm for R1|R2, taken from lecture slides

Having NFA, it will be converted to DFA. It is done at first by a function responsible for finding epsilon closure for a given node. Starting from initial node epsilon closure is found to create first node of DFA. Having that node, move functions for all input symbols in our RE are executed. If epsilon closure of any of move functions generates new set of nodes, it will be saved as a new DFA node and the algorithm used for initial node will be repeated on it. When all new nodes are found (there is no more possibilities when all nodes were iterated though with move functions) our DFA is complete and will be ready for test input strings.

Such input strings can be composed only of any strings. If they contain any improper symbols, simply string will be declared to be not accepted by the DFA.

III. Implementation

General architecture

Program will is written in C++ in a form of console application.

It is composed of classes storing nodes. There are separate NFA and DFA nodes, however, they share the same class. Their transition patters are slightly different. Each node is a separate object. Code is be divided into many small .h and .cpp files to make it easier to read and operate on.

Data structures

Both NFA and DFA nodes store information about possible transitions (next node and required input symbols to get there) with the difference that for NFA there will be always only one input symbol leading to the next node, while in NFA there may be more than one.

```cpp
#pragma once
#include <string>
#include <vector>

using namespace std;

class NodeMaster;

class Node {
    private:
        int NodeNumber;
        int NodeGroupIndex;                                 //easier recognition of origin
        string transition = "";                             //consecutive letters define consecutive transitions
        string transitionDFA = "";
    public:
        bool endNode = false;
        Node(int num);
        vector<Node*> nextNodes;                            //nodes and their transitions
        vector<Node*> nextNodesDFA;
        vector<int> DFANodes;
        void addNextNode(char tran, Node* next);
        void addNextNodeDFA(char tran, Node* next);
        string getTransitionAtPosition(int pos);            //get transition symbol of particular transition
        string getTransitionAtPositionDFA (int pos);
        int getTransitionAtInput(string input);
        int getNodeNumber();
        int getNodeNumberOfTransitions();
        int getNodeNumberOfTransitionsDFA();
};
```

Figure 4. Node class

Pointers to next nodes are stored in a vector, similarly transition strings – if there will be more than one possible transition in DFA string will consist of all possible inputs, else it will be single character string.

Pointers to starting node of DFA and NFA is be stored in NodeMaster class that has only one object. It also stores information about number of nodes and a set of functions to operate on nodes. Those function are for example responsible for generating e-closure, move functions, traversing graphs, appending elements to them by creating appropriate nodes for all operators in all possible configurations.

From all this information we are able to generate NFA or DFA graph and present in simplified graphical form - table. Nonetheless, it is handled by functions in .cpp files dedicated for DFA/NFA generation which only use NodeMaster's or Nodes' functions when needed.

String acceptance is checked in a function specially designed for this purpose. It checks it we are able to reach final state after inputting all input characters. If we are finishing in a node that is not an accepting state or some input symbol does not provide any transition we discover that the input is not accepted and proper message is displayed. If we reach final state, we are informed that the string indeed is accepted.

```
class NodeMaster{
    private:
        int NumberOfNodes = 0;
        int NumberOfDFANodes = 0;
        int NumberOfNodeGroups = 0;
        Node* startNode = nullptr;
        Node* endNode = nullptr;
        Node* startDFA = nullptr;
        vector<Node*> subNodeStart;
        vector<Node*> subNodeEnd;
        Node* SearchSubNode(Node* node, int index, vector<int> indexList);
        Node* SearchSubNodeDFA(Node* node, int index, vector<int> indexList);
        vector<int> RecursiveClosure(Node* node, vector<int> indexes);
    public:
        Node* getDFAstart();
        int GetNumberOfDFANodes();
        void setDFAstart(Node* node);
        void setEndNode(Node* node);
        void IncrementDFANodes(int number);
        void IncrementNodes(int number);
        void IncrementNodeGroups(int number);
        Node* GetStartNode();
        Node* GetEndNode();
        void SetStartNode(Node* newstart);
        int GetNumberOfNodes();
        void CreateStar(char trans);
        void CreateOr(char trans1, char trans2);
        void CreateAnd(char trans1, char trans2);
        void check_if_start_node_exists(Node* newnode);
        void ternimateSubNodeGeneration();
        Node* findDFAnodeATtransition(string trans, Node* curr);
        Node* getNodeWithIndex(int index);
        Node* getDFANodeWithIndex(int index);
        vector<int> getEClosure(int nodeNumber);
        vector<int> getMove(vector<int> DFAnode, char trans);
        vector<int> getEClosure(vector<int> moveNodes);
};
```

Figure 5. NodeMaster class

Input/output description

As specified before, input has to be delivered to program via console. Output consists of information if particular string can be generated by DFA in form of output to console. It specifies shortly reason when string is not accepted and it also provides transitions' order within DFA that was used to reach particular states.

```
Transition's order: A->B->C->C->C->C->C
String aaaaaa is not accepted! It doesn't end in final state

Transition's order: A->B->C->C->C->C->D->D->E->D->D->E->D->D->D->E->C->C->C->C->C->C->C->C->C->C->C->C->C->C->C->C->D->E
String aaaaabbabbabbbaaaaaaaaaaaaaaaba is accepted!

Transition's order: A->B->D
String abcde is not accepted! Invalid transition or input symbol found
```

Figure 6. Checking if input is accepted

NFA and DFA is generated in a form of table. Nodes are presented with numbers (NFA) or capital letter (DFA), transitions are also provided for each state. Start state is symbolized with 'S' and final state with 'F'.

```
Printing NFA transitions:
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
- Node 0| Transitions = 2 |   To node 1 on E To node 2 on E
- Node 1| Transitions = 1 |   To node 3 on a
- Node 2| Transitions = 1 |   To node 4 on b
- Node 3| Transitions = 1 |   To node 5 on E
- Node 4| Transitions = 1 |   To node 5 on E
- Node 5| Transitions = 2 |   To node 0 on E To node 7 on E
- Node 6| Transitions = 2 |   To node 0 on E To node 7 on E
- Node 7| Transitions = 1 |   To node 9 on b
S Node 8| Transitions = 1 |   To node 6 on a
- Node 9| Transitions = 1 |   To node 10 on a
F Node 10| Transitions = 0 |
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Figure 7. Printing NFA for a(a|b)*ba

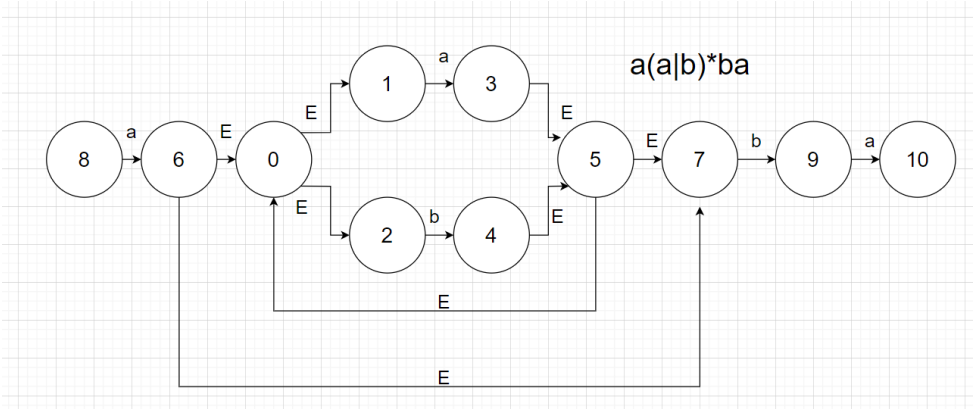

Figure 8. NFA for a(a|b)*ba

It can be observed that node numbering does not start from 1. It is due to the fact that the order of creation of NFA parts depends on postfix notation. As a|b operation is indicated to be executed first, we can observed nodes 0-5 in the middle of the graph. Then we perform * operation creating nodes 6 and 7. Concatenation with “a” is next, therefore, node 8 is created – it is a start node (I forgot to draw an arrow on figure 8.). At the end “b” and “a” are also concatenated creating nodes 9 and 10 respectively.

DFA nodes are also defined as integeres, however, to differentiate them more easily I add to them 65 and cast them to char (0->A, 1->B, etc.). Contrary to NFA, in DFA there can be more than one accepting state, which is taken into consideration.

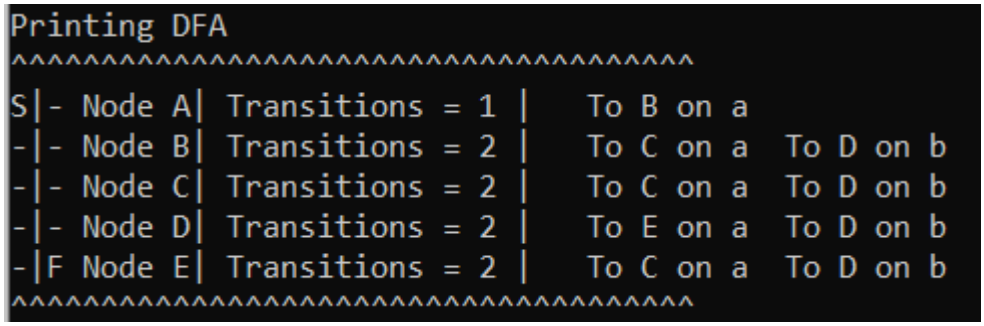

Figure 9. Printing DFA for a(a|b)*ba

Additional information regarding move, closures, postfix notation is also provided, mostly for purpose of the debugging, but it also prvides a lot of useful information.

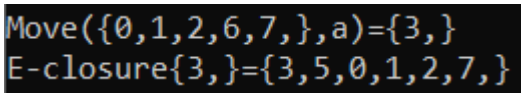

Figure 10. Move and e-closure

IV. Functional test cases

Positive:

1) RE: a|b INPUT: a *// simple check*

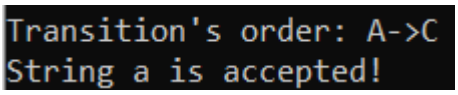

Figure 11. Test case

2) RE: (ab)* INPUT: E *//E will mean epsilon in my code, checks if * works*

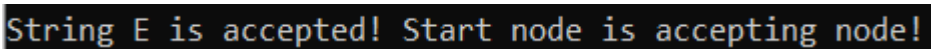

Figure 12. Test case

3) RE: (a(a|bb))|E INPUT: abb *// checks if brackets and | work*

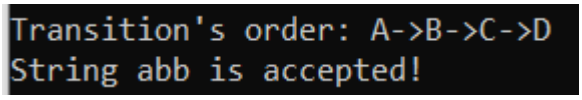

Figure 13. Test case

4) RE: a(a|b)*ba INPUT: aba *//checks more advance * version with non-E output*

```
Transition's order: A->B->D->E
String aba is accepted!
```

Figure 14. Test case

5) RE: aa*|b* INPUT: bbb *//checks combination | and **

```
Transition's order: A->C->C->C
String bbb is accepted!
```

Figure 15. Test case

Negative:

1) RE: (aaaa // *no right bracket*

```
Original RE: (aaaa
Unequal number of left and right parenthesis! Aborting!
Process returned -2 (0xFFFFFFFE)   execution time : 0.220 s
```

Figure 16. Test case

2) RE:)a*b(// *")" cannot be before "("*

```
Original RE: )a*b(
RE cannot start with )|* symbols. Aborting!
Process returned -4 (0xFFFFFFFC)   execution time : 0.224 s
```

Figure 17. Test case

3) RE: 1234 // *symbols not allowed*

```
Original RE: 1234
Only small letters and E as eps allowed!  Aborting!
Process returned -5 (0xFFFFFFFB)   execution time : 0.333 s
```

Figure 18. Test case

4) RE: a(a|b)*ba INPUT: baaa *//such input cannot be generated by this RE*

```
Transition's order: A
String baaaa is not accepted! Invalid transition or input symbol found
```

Figure 19. Test case

5) RE: a||b // *we cannot have more than one "|" in a row*

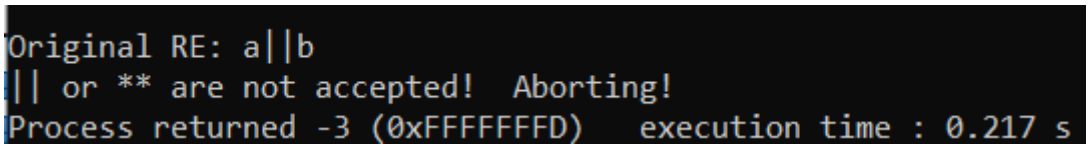

Figure 20. Test case