Transmutation of Regular Expression to Source Code Using Code Generators

Rashmi Sinha\textsuperscript{1}, Ashish Dewangan\textsuperscript{2}

\textsuperscript{1}Department of Computer Science & Engineering, Chhattisgarh Swami Vivekanand Technical University
Rajnandgaon, Chhattisgarh, India

\textsuperscript{2}Department of Electronics & Telecommunication Engineering, Chhattisgarh Swami Vivekanand Technical University
Durg, Chhattisgarh, India

Abstract—Because of the great possibility to make mistakes while dealing with the Automata, and programming and resolving problems within automata theory as it is a complex process, time consuming and still the results may not be reliable, various tools technologies and approaches are present today which significantly reduce complexity and increase productivity of dealing with regular languages in a very short time. This paper throws light on approaches which are in use of dealing with regular languages which significantly...

Keywords—Regular Expression, Finite Automata, Code Generators.

I. INTRODUCTION

Implementing finite automata for regular expression is a process that takes too much time, depending on the regular expression complexity. This project gives introduction about the processes and algorithms which is applied in implementing conversion of finite automata into Java Code. Process takes as input a regular expression or any automaton and converts it into an equivalent nondeterministic and deterministic finite automaton, then automatically generates Java Source Code with the implementation of the DFA for the given RE.

According to Ullman \cite{1} the theory of Finite Automata is a mathematical theory of algorithms that is important in computer science. In later times the algorithms were classified into two approaches. The first classification was according to their running time, and the second classification was according to the types of memory used in implementation, whereas the simplest algorithms are the ones that can be implemented by finite automatons. Finite automatons were first introduced by Stephen Kleene in the 1950s, who found some important applications of them in computer science as in the design of computer circuits and in the lexical analysers in compilers.

To create a Code Generator for Regular Expressions, following are the steps to be followed \cite{11}:

- Creation of a grammar to define input regular expressions.
- Converting regular expression into non deterministic finite automaton.
- Turning NFA into an equivalent deterministic finite automaton.
- Generating Java Source Code that equivalently represents the deterministic finite automaton.

A. Regular Expression

Regular expressions, also known as regex or regexp, provide a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

In Java regular expressions can use with the java.util.regex package, for the following common scenarios as examples:

- Simple word replacement
- Email validation
- Removal of control characters from a file
- File Searching

To develop regular expressions, ordinary and special characters are used:

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>SPECIAL CHARACTERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$/</td>
<td>^</td>
</tr>
<tr>
<td>+</td>
<td>?</td>
</tr>
<tr>
<td>.</td>
<td>[</td>
</tr>
<tr>
<td>\</td>
<td></td>
</tr>
</tbody>
</table>

B. Finite Automata

According to Aho & Ullman \cite{1} a finite automaton is a graph-based way of specifying patterns. A finite automaton, FA, provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state. It can also be given a formal mathematical definition. Finite automata are used for pattern matching in text editors, for compiler lexical analysis.

It is also possible to prove that given a language L there exist a unique (up to isomorphism) minimum finite state automaton that accepts it, i.e. an automaton with a minimum set of states. An example of how an automaton looks like is shown in Figure 1.
C. Code Generator

The code generator takes as input the reference source code from the previous project, uses the regular expressions specified in the component parameterization to find all of the relevant places in the code, and substitutes those places with the information of the project-specific module configuration. The result is a new source code module that can be directly incorporated into the current project. The code generator can also modify the source code of the current project to better integrate the desired module.[9]

![Fig. 1 Finite Automation Example](image)

- **Reference source code**
- **Current Source Code**
- **Specific Component Configuration**
- **Parameterizing**
- **Code Generation Framework**
- **Modified Source Code**

![Fig. 2 Overview of the code generator](image)

II. BACKGROUND

Traditionally, a parser and translator generator tool that is ANTLR (ANother Tool for Language Recognition) that uses LL (*) (left to right parser and constructs a leftmost derivation of the sentence) parsing had been used. It takes input a formal language grammar that specifies a language and as output it generates source code that recognizes all the strings for the given input grammar.

The ANTLR language is specified using the context-free grammars. It can create lexers, parsers and AST’s (Abstract Syntax Tree). The role of lexer is to quantify the meaningless stream of characters into discrete groups that have meaning when processed by the parser. The important task of the lexer is to look at the entire program being compiled, and breaking it up into tokens.

The AST is a special kind of tree that that represents the abstract syntactic structure of the code, we say abstract because it does not represent every detail from the code (real syntax). AST’s contains additional information’s implicitly that makes it easier to translate to a target language. Creating an abstract syntax tree is the most important part of a language translation process. AST can have an arbitrary number of subtrees, which by themselves are ASTs. [7]

![Fig. 3 The Abstract Syntax Tree for “(ac|b)*” regular expression](image)

A. Thompson, McNaughton and Yamada Algorithm

Thompson is the basic algorithm used previously, in which algorithm parses the input string (RE) using the bottom-up method, and constructs the equivalent NFA. The final NFA is built from partial NFA’s, it means that the RE is divided in several subexpressions, in our case every regular expression is shown by a common tree, and every subexpression is a subtree in the main common tree. [11]

The idea of the McNaughton and Yamada algorithm is that it makes diagrams for sub-expressions in a recursive way and then puts them together.

The translation to NFA from an RE is done by using a combination of both algorithms. The conversion is done as follows: the regular expression is converted into an Antlr Tree, then using a depth first search method we traverse the tree and start to build the NFA graph.

Each subtree is converted into a subgraph then it is concatenated (merged or joined) to the main graph. The expressionList’s child node can be one or more EXPRESSION nodes. The booleanOr is followed by another
The subset algorithm works as follows. Sometimes states on D get D simple N. This exponential growth of the states is not a concern for being shown in ANTLR. Most states of N into A the situation is not so bad as it looks in theory, because most of NFAs when converted, the number of states in DFA does not grow much at all, in fact sometimes it is fewer than in NFA. The subset algorithm works as follows:

1. The start state of DFA are all states of the NFA that can be reached with empty transition
2. For each new state of DFA do:
   - For each character of alphabet move to all reached states in that character perform an empty closure for the set of states we got (the result from empty closure can be a new state or an already existing State).
3. If at least one of the states from the result is accepted state in NFA it is accepted state in DFA as well.

For this algorithm to be clearer below is an example of transforming an NFA into an equivalent DFA. The regular expression is: a*c|bc. The corresponding NFA is shown in Figure below:

Fig. 3 NFA for Regular Expression a*c|bc

The start state of the DFA is the set of states reached in empty transition from start state of NFA, that is {0, 1, 4}. The alphabet of the NFA is: {a, b, c}.

Fig. 4 DFA for Regular Expression a*c|bc

The equivalent DFA is shown in Figure 4. Note that all states of DFA are named as follows: we keep track of each new state created, for example the first state created that is the start state has number 0, and then on each new state we increment this number per one, for example:

0= {0, 1, 4}, 1= [4], 2= [10], 3= [13].

III. CODE GENERATION APPROACHES

A. The If-Else Approach

The if-else approach is an easy way of implementing a DFA automaton, such a way works by creating an if statement for each input and the state of the automaton. The automaton has two methods, update and matches. The update method moves the current state of the automaton.
into another state depending on which input character of the input string is next read. When there is no more characters to be read from the input string the matches method is called, which simply checks if the current state of the automaton is one of the accepted states. The dead synonymously trap state represents a node when there is no transition from one state to another in the given input character, and when the automaton visits this state it never leaves it, so it mean that the given input string will never match. There must be a main method that starts this automaton, and breaks the input string into characters, that can be done by reading the input string character by character, and for each character the update method of the automaton is called.

The algorithm on how the automaton changes from one node to another is described with the pseudocode below:

1. Iterate over successors of the currentState.
2. Iterate over all edges between the currentState and the successors if the edge is the same as the character input.
3. return true
4. after each successor has been iterated an no edge has been found change the current state to the trap state and return false.

After the end of the input string has been reached, the matches’ method is called, that simply checks if the description of the current node is Accepted or Initial & Accepted.

B. The Graph Approach

There are two possible ways of representing graphs in computer sciences, adjacency matrix and adjacency list. Since graph representation is irrelevant for this project, instead of creating new graph representation, grail library will be used while representing directed graphs and while discussing the algorithm that describes the moves of the automaton. This means that when the graph output source is used the grail library must be imported (built to path) to run this program. To represent the automaton the first thing we have to do is build that automaton, so we should create for each state of the DFA a node in the Java class, and for each arc between states and edge should be created. All the nodes and edges are created in the buildGraph() method, where each node and edge is labelled.

First, a global SetBasedDirectedGraph is created, together with all kind of state definitions:
- State - that is the node represented using DirectedNodeInterface,
- A dead state - that is useful for non-existing edges between two nodes,
- CurrentState - that keeps track of the current position in the automaton, and
- Edge - that is represented using DirectedEdgeInterface

The algorithm on how the automaton changes from one Node to another is described with the pseudocode below:
1. Iterate over successors of the currentState.
2. Iterate over all edges between the currentState and the successors.
3. If the edge is the same as the character input.
4. Return true.
5. After each successor has been iterated an no edge has been found change the current state to the trap state and return false.

IV. COMPARING THE APPROACHES OF CODE GENERATION

The following sections describe each comparison criteria in detail should be used.

A. Transitions

The transitions in if-else is done with the help of two methods, update and matches. The update method moves the current state of the automaton into another state depending on which input character of the input string is next read. When there is no more characters to be read from the input string the matches method is called, which simply checks if the current state of the automaton is one of the accepted states.

Whereas in Graph Approach, the transitions are represented by connecting two nodes with an edge, that means that the source node is connected with the target node by an arc, so that with input character the a transition is made.

B. Work done in Implementing

Implementing a DFA automaton using the if-else approach is an easy way, such a way works by creating an if statement for each input and the state of the automaton.

For a to represent the automaton the first thing we have to do is build that automaton, so we should create for each state of the DFA a node in the Java class, and for each arc between states and edge should be created. There are two possible ways of representing graphs in computer sciences, adjacency matrix and adjacency list. Since graph representation is irrelevant for this project, instead of creating new graph representation, grail library will be used while representing directed graphs and while discussing the algorithm that describes the moves of the automaton. This means that when the graph output source is used the grail library must be imported (built to path) to run this program.

C. Error Checking

If-else approaches require making a template for each file in the reference source code. This process is error prone, since developers may introduce unwanted changes in the templates that do not correspond to the original source code. Moreover,
debugging of those templates requires that developers first generate code using the templates, compile that code, execute it, determine whether the generated code behaves consistently or not with the reference source code.

In contrast, the Graph approach does not require to manually creating new files for each reference source file. With minor modifications, the reference source code can be used to create a generator. Therefore, the risk of having an incorrect representation of the reference source code is reduced.

D. Maintaining

The evolution of a code generation framework is associated to the creation of a new generator or the modification of an existing one.

For the if-else approach, whenever a new generator is required, developers must repeat the process of creating automatons for each reference source file. If a generator needs to be modified, developers must modify the corresponding templates, generate code, compile and execute it, to ensure it realizes the required changes.

For the graph approach whenever a new generator is required, developers must perform minor modifications to the reference source files, and create the required regular expressions substitutions. If an existing generator needs to be modified, developers only need to modify the reference source code, compile and test it.

Table III summarizes the comparison between the approaches. The symbol ‘+’ indicates that the corresponding approach is better than the other according to the corresponding criteria. The symbol ‘-’ indicates that the approach is worse than the other. The symbol ‘=’ indicates that both approaches satisfy the criteria similarly.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>If-Else</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitions</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Work done in Implementing</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>Error Checking</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>Maintaining</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

CONCLUSIONS

This paper presented approaches for code generation and compared it with the traditional if-else based approach. The approach of regular expression substitutions, while it has a difficult implementation effort, it offers a better maintainability.

Regular Expression Code Generator is interactive software for visualizing finite automata, and converting these automatons in Java executable Source Code in a quick, fast and effective way.

Acceleo (cross-platform / Eclipse Java) is a Model to Text tool that has the same aim as the Regular Expression, Code Generator, the aim of helping developers to write the code in a faster way, a tool that is implemented in Java which as input takes user defined EMF based models (UML, Ecore, user defined metamodels) and generates any textual language as output. Jottraca is another software tool that generates any code into Java or JSP code, this kind of tools is called source-to-source generators. Other than some of the code generator tools, RegExCodeGen is platform independent software.

Generating a more optimized code in cases when range intervals are used will increase the maintainability and productivity of the generated source code, and will decrease the complexity even more.

ACKNOWLEDGMENT

No volume of words is enough to express my gratitude towards my guide, who has been very concerned and has aided for all the material essential for the preparation of this paper. He has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented and preparation of this paper venture.

REFERENCES


ISSN: 2231-2803  http://www.internationaljournalsssrg.org  Page 791