Volume 3 Issue IX, September 2015
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

# Methods of Regular Expression

Neha[1], Abhishek Sharma[2]
[1]M.Tech, [2]Assistant Professor
Department of Cse, Shri Balwant College of Engineering &Technology, Dcrust University

***Abstract - Regular expressions are used to represent certain set of string in algebraic manner. Regular expressions are widely used in the field of compiler design, text editor, search for an email- address, grep filter of unix, train track switches, pattern matching ,context switching and in many areas of computer science. The demand of converting regular expression into finite automata and vice versa motivates research into some alternative so that time taken for above is minimized. For conversion of deterministic finite automata to regular expression, several techniques like Transitive closure method, Brzozowski Algebraic method and state elimination method have been proposed.***
***Keywords: DFA, NDFA, Regex, Automata, fn, ht***

## I. INTRODUCTION

Regular expressions is a form of pattern matching that you can apply on textual content. Take for example the DOS wildcards **?** and ***** which you can use when you're searching for a file. That is a kind of very limited subset of RegExp. For instance, if you want to find all files beginning with **"fn"**, followed by 1 to 4 random characters, and ending with **"ht.txt"**, you can't do that with the usual DOS wildcards. RegExp, on the other hand, could handle that and much more complicated patterns. Regular expressions are, in short, a way to effectively handle data, search and replace strings, and provide extended string handling. Often a regular expression can in itself provide string handling that other functionalities such as the built-in string methods and properties can only do if you use them in a complicated function or loop.

### *A. RegExpression Syntax*

There are two ways of defining regular expressions in JavaScript — one through an object constructor and one through a literal. The object can be changed at runtime, but the literal is compiled at load of the script, and provides better performance. The literal is the best to use with known regular expressions, while the constructor is better for dynamically constructed regular expressions such as those from user input. In almost all cases you can use either way to define a regular expression, and they will be handled in exactly the same way no matter how you declare them.

### *B. Declaration*

Here are the ways to declare a regular expression in JavaScript. While other languages such as PHP or VBScript use other delimiters, in JavaScript you use forward slash (/) when you declare RegExp literals.

| Syntax | Example |
|---|---|
| RegExp Literal | |
| /pattern/flags; | var re = /mac/i; |
| RegExp Object Constructor | |
| new RegExp("pattern","flags"); | var re = new RegExp(window.prompt("Please input a regex.","yes\|yeah"),"g"); |

### *C. Regular Expression*

A regular expression (RE) is a pattern that describes some set of strings. Regular expression over a language can be defined as:
Regular expression for each alphabet will be represented by itself. The empty string (ϵ) and null language (ϕ) are regular expression denoting the language {ϵ} and {ϕ} respectively.
If E and F are regular expressions denoting the languages L (E) and L (F) respectively, then following rules can be applied recursively.

Union of E and F will be denoted by regular expression E+F and representing language L (E) U L (F).
Concatenation of E and F denoted by EF and representing language L (E*F) = L (E) * L (F).
Kleene closure will be denoted by E* and represent language (L (E))*.
Any regular expression can be formed using 1-2 rules only.

*D. Flags*

| Flag | Description |
|---|---|
| Global Search | |
| g | The global search flag makes the RegExp search for a pattern throughout the string, creating an array of all occurrences it can find matching the given pattern. |
| Ignore Case | |
| i | The ignore case flag makes a regular expression case insensitive. For international coders, note that this might not work on extended characters. |
| Multiline Input | |
| m | This flag makes the beginning of input (^) and end of input ($) codes also catch beginning and end of line respectively. JavaScript1.5+ only. |

## II. CONVERSION OF DFA TO RE

Kleene proves that every RE has equivalent DFA and vice versa. On the basis of this theoretical result, it is clear that DFA can be converted into RE and vice versa using some algorithms or techniques. For converting RE to DFA, first we convert RE to NFA(Thomson Construction) and then NFA is converted into DFA(Subset construction).For conversion of DFA to regular expression, following methods have been introduced.
Transitive closure method
Brzozowski Algebraic method
State elimination method

*A. Transitive Closure Method*

Kleene's transitive closure method [2] defines regular expressions and proves that there is equivalent RE corresponding to a DFA. Transitive closure is the first mathematical technique, for converting DFAs to regular expressions. It is based on the dynamic programming technique. In this method we use $R^k_{ij}$ which denotes set of all the strings in $\Sigma^*$ that take the DFA from the state $q_i$ to $q_j$ without entering or leaving any state higher than $q_k$. There are finite sets of $R^k_{ij}$ so that each of them is generated by a simple regular expression that lists out all the strings.

*B. Brzozwski Algebraic Method*

Brzozowski method is a unique approach for converting deterministic finite automata to regular expressions. In this approach first characteristic equations for each state are created which represent regular expression for that state. Regular expression equivalent to deterministic finite automata is obtained after solving the equation of $R_s$ (regular expression associated with starting state $q_s$).

*C. State Elimination Method*

The state removal approach is widely used approach for converting DFA to regular expression. In this approach, states of DFA are removed one by one until we left with only starting and final state, for each removed state regular expression is generated. This newly generated regular expression act as input for a state which is next to removed state. The advantage of this technique over the transitive closure method is that it is easier to visualize. This technique is described by Du and KO, but a much simpler approach is given by Linz. First, if there are multiple edges from one node to other node, then these are unified into a single edge that contains the union of inputs. Suppose from q1 to q2 there is an edge weighted ‘a’ and an edge weighted ‘b’, those would be unified into one

edge from q1 to q2 that has the weight (a + b). If there are n accepting states, take union of n different regular expressions.

## III. PROBLEM STATEMENT

Regular expressions, also known as regex, are commonplace in computing. They are generally used for matching or replacing strings of text, such as when searching documents for words, filtering email for spam, or searching the web. They feature in most major programming languages and are central to the working of parsers such as Lex. More formally, they can define a class of languages known as regular languages. For regular expressions to be usable by computers they are converted to various types of finite automata. It is this process of conversion that we will investigate here. The types of finite automata we will deal with are:

*DFA (deterministic finite automata)*

*NFA (nondeterministic finite automata)*

*GNFA (general nondeterministic finite automata)*

The steps we will perform to create them are:

*Conversion of regular expression to NFA*

*Conversion of NFA to DFA*

*Conversion of DFA to minimal DFA*

*Conversion of DFA to regular expressions*

This last step is to demonstrate that our implementation of the algorithms is correct – if the final regular expression describes the same language as the first in each test instance it will be a strong indicator of success.
It became clear early in the investigation that the most sensible method to reach our goal was to use the following:

Thompson's Algorithm

Subset Construction

DFA minimisation by removal of dead, inaccessible and redundant states from DFA

Conversion of DFA to GNFA and removal of states from GNFA

## IV. IMPLEMENTATION

The source code for a Java applet that, upon the user entering a regular expression can perform the following:

*Convert regular expression to NFA*

*Convert NFA to DFA*

*Minimise DFA*

*Convert DFA back to regular expression via GNFA*

### *A. Classes*

The applet consists of six classes. Mostly they were straightforward to implement.
*1) RegexAutomaton.Class (The Main Class):* It creates the GUI, deals with input and creates any Regex objects as required which it

then adds to its database.

*2) Regex.Class (An Individual Regex):* It contains the terms for the original regular expression, trees of Node objects for the NFA, DFA and minimised DFA, a two dimensional array representing the GNFA, and the terms for the final regular expression. It also contains the algorithms for converting one to another.

*3) Node.Class (An Abstract Parent Class Of The NfaNodeAnd DfaNode Classes):* It contains the basic commonality of Nodes – node number, indicators of initial and final states.

*4) DfaNode.Class (A Subclass Of The Node Class):* It implements a DFA node. In addition to the sections inherited from its parent class it contains a vector for DfaNodes moved to from this DfaNode and functions to iterate through DfaNodes.

*5) NfaNode.Class (A Subclass Of The Node Class):* It is similar to the DfaNode class but with the different transition table according to NFA requirements.

*6) IllegalRegexTermsException Class:* An exception class thrown when illegal regular expression terms are provided to the application, e.g. two pipes in a row, a regex that starts with a closing bracket, and so on.

*B. Functions*

The key functions used in above classes are explained below:

*1) GetStatement ()*:This function takes care of most of the parsing. It checks for symbols or sequences of symbols. The various combinations of symbols it looks for are clearly commented in the code. Each term is converted to an NfaNode object, or a number of NfaNode objects, and is linked to the NfaNode objects created so far.

*2) DoUnion ()*: If getStatement () function encounters a union it parses the left hand part itself and then calls this function to deal with the right hand side of the union.

*3) ConvertNFAtoDFA ()* : It converts the graph of NfaNode objects to a graph of DfaNode objects.

*4) MinimiseDFA ()* : It copies the graph of DfaNode objects and minimises it.

*5) ConvertDFAtoRegex ()* : It converts the minimised graph of DfaNode objects back to a regex by creating and minimizing a GNFA in the form of a two dimensional array of String objects.

*6) MakeCell () :* applies Arden's rule to the GNFA matrix cells to minimize the GNFA

## V. RESULTS

Regular expression to deterministic finite automata and vice versa using heuristics proposed by various researchers. Software for conversion has been developed in java. After execution of software the GUI screen of java applet is displayed.

## VI. CONCLUSION

The state removal approach seems useful for determining regular expressions via manual inspection, but is not as straightforward to implement as the transitive closure approach and the algebraic approach. The transitive closure approach gives a clear and simple implementation, but tends to create very long regular expressions. The algebraic approach is elegant, leans toward a recursive approach, and generates reasonably compact regular expressions. Brzozowski's method is particularly suited for recursion oriented languages, such as functional languages, where the transitive closure approach would be cumbersome to implement.

## REFERENCES

[1] Dean N. Arden. Delayed-logic and finite-state machines. In Theory of Computing Machine Design, pages 1–35. U. of Michigan Press, Ann Arbor, MI, 1960.

[2] G. Berry and R. Sethi. From regular expressions to deterministic automata. TCS: Theoretical Computer Science, 48:117–126, 1987.
[3] Janusz A. Brzozowski. Derivatives of regular expressions. J. ACM, 11
[4] (4):481–494, 1964. Ding-Shu Du and Ker-I Ko. Problem Solving in Automata, Languages, and Complexity. John Wiley & Sons, New York, NY, 2001.
[5] John E. Hopcroft and Jeffery D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, Reading, MA, 1979.
[6] Richard Y. Kain. Automata Theory: Machines and Languages. Robert E. Krieger Publishing Company, Malabar, FL, 1981.
[7] S. C. Kleene. Representation of events in nerve nets and finite automata. In Automata studies, pages 3–40. Ann. of Math. Studies No. 34, Princeton University Press, Princeton, NJ, 1956.
[8] Peter Linz. An introduction to Formal Languages and Automata. Jones and Bartlett Publishers, Sudbury, MA, third edition, 2001.
[9] E. F. Moore. Gedanken experiments on sequential machines. In Automata Studies, pages 129–153. Ann. of Math. Studies No. 34, Princeton University Press, Princeton, NJ, 1956.
[10] Arto Salomaa. Jewels of Formal Language Theory. Computer Science Press, Rockville, MD, 1984.