



[home](#) [articles](#) [quick answers](#) [discussions](#) [features](#) [community](#)  
[help](#)

Articles » General Programming » Algorithms & Recipes » Parsers



# Writing own regular expression parser



Amer Gerzic, 14 Nov 2003

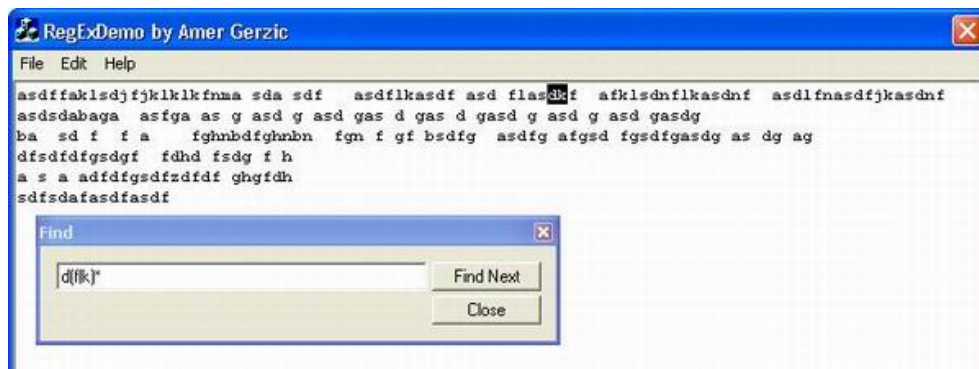
★★★★★ 4.91 (127 votes)

Rate this:

Explains principles behind writing regular expression parsers.

[Download source files - 8.48 Kb](#)

[Download demo project - 22.9 Kb](#)



## Introduction

Have you ever wondered how regular expressions work in detail? If the answer is yes then this article is right for you. I will try to guide you step by step on how to create your own mini regular expression library. The purpose of the article is NOT to give you a highly optimized, tested and great regular expression library, but to explain principals behind the pattern matching in text files. If you only want a good library and don't care how it works, you probably want Boost regex library, which you can find [here](#). I used a couple of different regular expression libraries but I must say that I am happiest with Boost regex. Obviously you decide it for yourself. There is another regular expressions article here on CP which could be found [here](#). I must admit, I did not read the article completely but it seems to me that the article focuses mostly on how to use the regular expression library provided by the author. In this article, the author uses similar technique (I could be wrong though) to create a more sophisticated library. The article you are reading right now, could be seen as a prerequisite for the article I just mentioned above.

Regular expressions are part of MS .NET library or Java SDK (if you write code in Java). As you can see, regular expressions are available in many different programming languages and technologies. Actually the article does not focus on writing the library in a specific language. I wrote the code in C++, using STL primarily because it is my favorite language/library, but the principles from the article could be applied to any language (obviously). I will try to be as language independent as possible, using pseudo code where ever possible. If you want the code in Java, please send me an email. The code provided here in this article is free (obviously) but if you like it and use it in your application, it would be great if you would give me the credit for what I deserve. Also please email me, so I can show off to my peers and/or potential employers.

## Overview

So how are we going to do it? Well before we start with coding, it is necessary to explain the mathematical background needed to fully understand the method used here in this article. I would strongly recommend to read and understand the math behind, because once we overcome the math part, the rest will be very simple. Note however that I will not have any mathematical proofs. If you are interested in proofs, please check out the references, which could be found in [References](#) section of this article. Additionally, note that the regular expression parser, which we will create here will support these three operations:

1. Kleen Closure or Star operator ("\*\*")
2. Concatenation (For example: "ab")
3. Union operator (denoted with character "|")

However many additional operators can be simulated by combining these three operators. For instance:

1.  $A^+ = AA^*$  (At least one A)
2.  $[0-9] = (0|1|2|3|4|5|6|7|8|9)$
3.  $[A-Z] = (A|B|...|Z)$ , etc.

The reason for implementing only three operators is simplicity. When I started planning to write this article, I quickly recognized that I had to limit myself in many ways. The topic is so large, that it would require a book to explain every little detail (maybe I will write it someday). As I stated above, the purpose of the article is not to equip you with a library but to introduce you to principles behind regular expressions. If you want to know more on how to use regular expressions, then you can check out the book: *Mastering Regular Expressions* - O'Reilly.

Following is the overview of the article:

1. [What is NFA?](#)
2. [What is DFA?](#)
3. [Thompson's Algorithm](#)
4. [Subset Construction Algorithm](#)
5. [DFA optimization](#)
6. [Using the results from parts above](#)
7. [Final Words](#)
8. [Reference](#)

## What is NFA?

NFA stands for **nondeterministic finite-state automata**. NFA can be seen as a special kind of final state machine, which is in a sense an abstract model of a machine with a primitive internal memory. If you want to know more about finite-state machines, please refer to [References](#) section below.

Let us look at the mathematical definition of NFA.

An NFA  $A$  consists of:

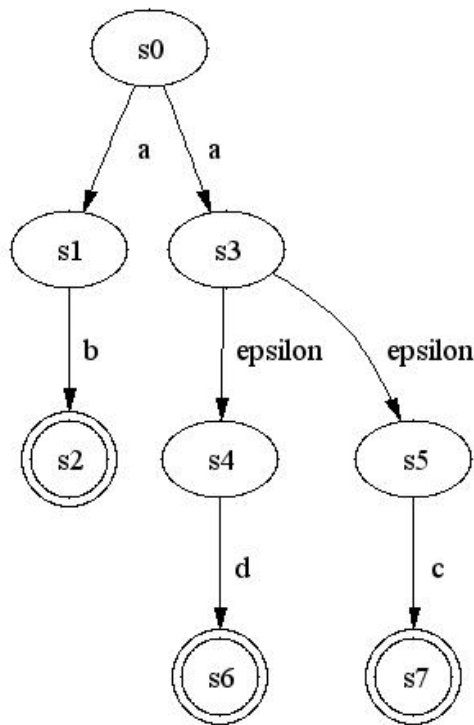
- a. A finite set  $I$  of input symbols
- b. A finite set  $S$  of states
- c. A next-state function  $f$  from  $S \times I$  into  $P(S)$
- d. A subset  $Q$  of  $S$  of accepting states
- e. An initial state  $s_0$  from  $S$

denoted as  $A(I, S, f, Q, s_0)$

If we would explain the above definition to a 12 year old, we could say that an NFA is a set  $S$  of states that are connected by function  $f$  (maybe to a smarter 12 year old). NFAs are represented in two formats: Table and Graph. For example, let us look at the table representation:

Input States	a	b	c	d	Epsilon
$s_0$	$s_1, s_3$				
$s_1$		$\{s_2\}$			
$\{s_2\}$					
$s_3$					$s_4, s_5$
$s_4$				$\{s_6\}$	
$s_5$			$\{s_7\}$		
$\{s_6\}$					
$\{s_7\}$					

An equivalent graph would be:



Looking at the table/graph above, we can see that there are special transitions called Epsilon transitions, which is one of the special features of NFA. A transition is an event of going from one state to another. Epsilon transition is a transition from one state to another on an empty string. In other words, we are going from one state to another on no character input. For example, as we can see from table/graph above, we can go on no input from **s3** to **s4** and **s5**, which means that we have a **choice**. Similarly, there is a **choice** to go from state **s0** to states **s1** or **s3** on character input **a**. Hence the name **nondeterministic**, because at some point, the path which we are able to go is not unique but we have a **choice**. The final and accepting states are drawn double circled (or enclosed in "{}" in the table), like for example **s6**. Once one of these states is reached, we have an accepting state, and hence an accepting character string.

In an NFA, like the mathematical definition defines, there is always a starting state. I used Graphviz, a great tool for drawing different kinds of graphs (see [References](#) section), for drawing of NFAs and DFAs (see later). Because Graphviz is laying out the nodes of a graph on its own, it seems to me that a starting state is always the state drawn at the top of the graph. So we will follow that convention.

## What is DFA?

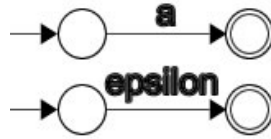
DFA stands for **deterministic finite state automata**. DFA is very closely related to NFA. As a matter of fact, the mathematical definition of NFA works for DFA too. But obviously there are differences, from which we will take advantage. One big difference is that in a DFA there are no Epsilon transitions. Additionally, for each state all transitions leading away from this state are done on different input characters. In other words, if we are in state **s**, then on input character **a**, there is a unique transition on that character from **s**. Additionally, in a DFA all states must have a valid transition on an input character. Input character here is finite set **I** of input symbols (like in mathematical definition). For example, in the above graph (under DFA), the set of symbols **I** would be **{a, b, c, d}**.

Now that we understand NFAs and DFAs, we can proceed by saying that given any NFA, there is an equivalent DFA (You are going to have to trust me on this because I don't think it is appropriate to give you the mathematical proof of this statement). As humans, generally it is easier for us to construct an NFA, as well as interpret what language an NFA accepts. But why do we need the DFAs then? Well if we think about computers, it is very hard to "teach" them to do very well educated guesses (sometimes even we can't make smart educated guesses). And this is exactly what the computer needs to do, when traversing an NFA. If we would write an algorithm, which would use an NFA to check for an accepting combination of characters, it would involve backtracking to check for choices that it did not make previously. Obviously, there are regular expression parsers which work using NFAs, but they are generally slower than those that use DFAs. This is due to the fact that a DFA has a unique path for each accepting string, so no backtracking is involved. Hence, we are going to use a DFA to check if a combination of input characters is accepted by an automata.

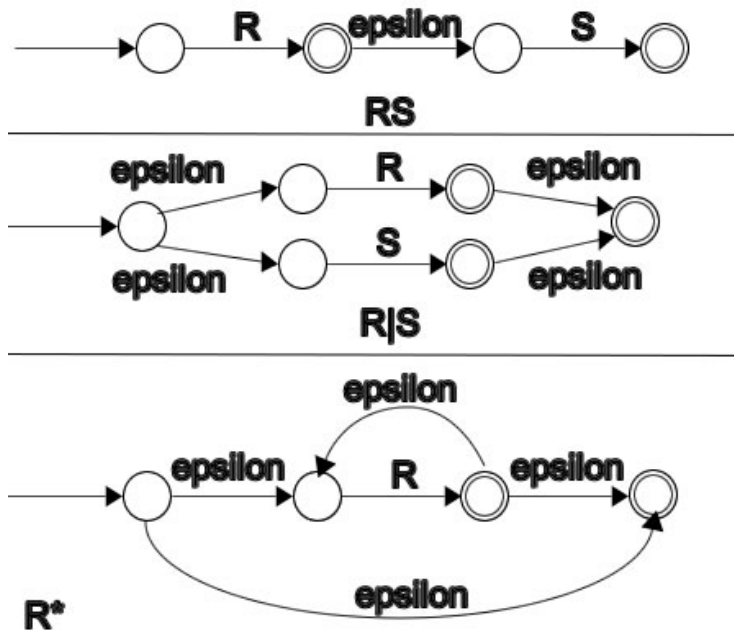
**Note:** If you really want to understand both NFAs and DFAs, I would recommend to do further reading on these topics. It is useful as an exercise to convert from one to another, to fully understand the difference and the algorithm used here to convert NFA to DFA (see later).

# Thompson's Algorithm

Now that we have all the mathematical background that we need to understand regular expressions, we need to start thinking about what is our goal. As a first step, we realize that we need a way of going from a regular expression (like  $ab^*(a|b)^*$ ) to a data structure, which will be easy to manipulate and use for pattern matching. But let us first look at the method for converting a regular expression to an NFA. Probably the most famous algorithm for doing this conversion is Thompson's algorithm. This algorithm is not the most efficient, but it ensures that any regular expression (assuming that its syntax is correct) will be successfully converted to an NFA. With the help of the basic NFA as seen from figure below, we can construct any other:



Using the basic elements above, we will construct three operations, which we would like to implement in our regular expression parser like the following:



But how do we go from something like  $(a|b)^*ab$  to the graph above? If we consider what we really need to do, we can see that evaluating regular expressions is similar to evaluating arithmetic expressions. For example, if we would like to evaluate  $R=A+B*C-D$ , we could do it like:

```
PUSH A
PUSH B
PUSH C
MUL
ADD
PUSH D
SUB
POP R
```

Here **PUSH** and **POP** are stacks and **MUL**, **ADD** and **SUB** take 2 operands from the stack and do the corresponding operation. We could use this knowledge for constructing an NFA from a regular expression. Let's look at the sequence of operations that need to be performed in order to construct an NFA from a regular expression  $(a|b)^*cd$ :

```
PUSH a
```

PUSH b  
 UNION  
 STAR  
 PUSH c  
 CONCAT  
 PUSH d  
 CONCAT  
 POP R

As we can see, it is very similar to the evaluation of arithmetic expressions. The difference is that in regular expressions the star operation pops only one element from the stack and evaluates the star operator. Additionally, the concatenation operation is not denoted by any symbol, so we would have to detect it. The code provided with the article simplifies the problem by pre-processing the regular expression and inserting a character ASCII code 0x8 whenever a concatenation is detected. Obviously it is possible to do this "on the fly", during the evaluation of the regular expression, but I wanted to simplify the evaluation as much as possible. The pre-processing does nothing else but detects a combination of symbols that would result in concatenation, like for example: `ab,a(,)a,*a,(,) (.`

PUSH and POP operations actually work with a stack of simple NFA objects. If we would PUSH symbol `a` on the stack, the operation would create two state objects on the heap and create a transition object on symbol `a` from state 1 to state 2. Here is the portion of the code that pushes a character on the stack:

[Hide](#) [Copy Code](#)

```
void CAG_RegEx::Push(char chInput)
{
    // Create 2 new states on the heap
    CAG_State *s0 = new CAG_State(++m_nNextStateID);
    CAG_State *s1 = new CAG_State(++m_nNextStateID);

    // Add the transition from s0->s1 on input character
    s0->AddTransition(chInput, s1);

    // Create a NFA from these 2 states
    FSA_TABLE NFATable;
    NFATable.push_back(s0);
    NFATable.push_back(s1);

    // push it onto the operand stack
    m_OperandStack.push(NFATable);

    // Add this character to the input character set
    m_InputSet.insert(chInput);
}
```

As we can see, the character is converted to a simple NFA and then the resulting NFA is added to the stack. `CAG_State` class is a simple class, which helps us structure the a NFA as we need it. It contains an array of transitions to other states on specific characters. Epsilon transition is transition on character 0x0. At this point, it is easy to see the structure behind NFA. An NFA (and DFA) is stored as a sequence of states (deque of `CAG_State` pointers). Each state is having as a member, all the transitions stored in a multimap. A transition is nothing else than mapping from a character to a state (`CAG_State*`). For detailed definition of the `CAG_State` class, please refer to the code.

Now back to the conversion from regular expression to NFA. Now that we know how to push the NFA onto the stack, the pop operation is trivial. Just retrieve the NFA from the stack and that's it. As I said earlier, a NFA table is defined to be a double ended queue (STL container `deque<CAG_State*>`). In this way, we know that the first state in the array is always the starting state, while the last state is final/accepting state. By preserving this order, we can quickly get the first and last states as well as append and prepend additional states when performing operations (like Star operator). Here is the code to evaluate each individual operation:

[Hide](#) [Copy Code](#)

```
BOOL CAG_RegEx::Concat ()
{
    // Pop 2 elements
    FSA_TABLE A, B;
    if (!Pop(B) || !Pop(A))
        return FALSE;

    // Now evaluate AB
    // Basically take the last state from A
    // and add an epsilon transition to the
    // first state of B. Store the result into
    // new NFA_TABLE and push it onto the stack
}
```

```

A[A.size()-1]->AddTransition(0, B[0]);
A.insert(A.end(), B.begin(), B.end());

// Push the result onto the stack
m_OperandStack.push(A);

return TRUE;
}

```

As we can see, the concatenation is popping two NFAs from the stack. First NFA is changed, so that it is now new NFA, which is then pushed on the stack. Note that we first pop second operand. This is the case because in regular expressions, the order of operands is of importance because  $AB \neq BA$  (not commutative).

[Hide](#) [Shrink](#) [Copy Code](#)

```

BOOL CAG_RegEx::Star()
{
    // Pop 1 element
    FSA_TABLE A, B;
    if(!Pop(A))
        return FALSE;

    // Now evaluate A*
    // Create 2 new states which will be inserted
    // at each end of deque. Also take A and make
    // a epsilon transition from last to the first
    // state in the queue. Add epsilon transition
    // between two new states so that the one inserted
    // at the begin will be the source and the one
    // inserted at the end will be the destination
    CAG_State *pStartState = new CAG_State(++m_nNextStateID);
    CAG_State *pEndState = new CAG_State(++m_nNextStateID);
    pStartState->AddTransition(0, pEndState);

    // add epsilon transition from start state to the first state of A
    pStartState->AddTransition(0, A[0]);

    // add epsilon transition from A last state to end state
    A[A.size()-1]->AddTransition(0, pEndState);

    // From A last to A first state
    A[A.size()-1]->AddTransition(0, A[0]);

    // construct new DFA and store it onto the stack
    A.push_back(pEndState);
    A.push_front(pStartState);

    // Push the result onto the stack
    m_OperandStack.push(A);

    return TRUE;
}

```

Star operator pops a single element from the stack, changes it according to the Thompson's rule (see above) and then pushes it on the stack.

[Hide](#) [Shrink](#) [Copy Code](#)

```

BOOL CAG_RegEx::Union()
{
    // Pop 2 elements
    FSA_TABLE A, B;
    if(!Pop(B) || !Pop(A))
        return FALSE;

    // Now evaluate A|B
    // Create 2 new states, a start state and
    // a end state. Create epsilon transition from
    // start state to the start states of A and B
    // Create epsilon transition from the end
    // states of A and B to the new end state
    CAG_State *pStartState = new CAG_State(++m_nNextStateID);
    CAG_State *pEndState = new CAG_State(++m_nNextStateID);
    pStartState->AddTransition(0, A[0]);
    pStartState->AddTransition(0, B[0]);
    A[A.size()-1]->AddTransition(0, pEndState);
    B[B.size()-1]->AddTransition(0, pEndState);

    // Create new NFA from A
    B.push_back(pEndState);
    A.push_front(pStartState);
    A.insert(A.end(), B.begin(), B.end());

    // Push the result onto the stack
}

```

```

m_OperandStack.push(A);

return TRUE;
}

```

Finally, the union pops two elements, makes the transformation and pushes the result on the stack. Note that here we have to watch for the order of the operation.

Finally, we are now able to evaluate the regular expression. If everything goes well, we will have a single NFA on the stack, which will be our resulting NFA. Here is the code, which utilizes the above functions.

[Hide](#) [Shrink](#) [Copy Code](#)

```

BOOL CAG_RegEx::CreateNFA(string strRegEx)
{
    // Parse regular expresion using similar
    // method to evaluate arithmetic expressions
    // But first we will detect concatenation and
    // insert char(8) at the position where
    // concatenation needs to occur
    strRegEx = ConcatExpand(strRegEx);

    for(int i=0; i<strRegEx.size(); ++i)
    {
        // get the character
        char c = strRegEx[i];

        if(IsInput(c))
            Push(c);
        else if(m_OperatorStack.empty())
            m_OperatorStack.push(c);
        else if(IsLeftParanthesis(c))
            m_OperatorStack.push(c);
        else if(IsRightParanthesis(c))
        {
            // Evaluate everything in parenthesis
            while(!IsLeftParanthesis(m_OperatorStack.top()))
                if(!Eval())
                    return FALSE;
            // Remove left paranthesis after the evaluation
            m_OperatorStack.pop();
        }
        else
        {
            while(!m_OperatorStack.empty() && Presedence(c, m_OperatorStack.top()))
                if(!Eval())
                    return FALSE;
            m_OperatorStack.push(c);
        }
    }

    // Evaluate the rest of operators
    while(!m_OperatorStack.empty())
        if(!Eval())
            return FALSE;

    // Pop the result from the stack
    if(!Pop(m_NFATable))
        return FALSE;

    // Last NFA state is always accepting state
    m_NFATable[m_NFATable.size()-1]->m_bAcceptingState = TRUE;

    return TRUE;
}

```

Function `Eval` is actually evaluating the next operator on the stack. Function `Eval()` pops the next operator from the operator stack, and using the `switch` statement, it determines which operation to use. Parenthesis are treated as operators too, because they determine the order of evaluation. The function `Presedence(char Left, char Right)` determines the precedence of two operators and returns `TRUE` if precedence of `Left` operator  $\leq$  precedence of `Right` operator. Please check out the code for implementation.

## Subset Construction Algorithm

Now that we know how to convert any regular expression to an NFA, the next step is to convert NFA to DFA. At first, this process seems to be very challenging. We have a graph with zero or more Epsilon transitions, and multiple transitions on single character and we need an equivalent graph with no Epsilon transitions and a unique path for each accepted sequence of input characters. Like I said, it seems to be very challenging, but it is really not. Mathematicians actually already solved that problem for us, and then using the results, computer scientists created the Subset Construction Algorithm. I am not sure whom to give credit here but the Subset Construction

Algorithm goes like this:

First, let us define 2 functions:

- **Epsilon Closure**: This function takes as a parameter, a set of states **T** and returns again a set of states containing all those states, which can be reached from each individual state of the given set **T** on Epsilon transition.
- **Move**: Move takes a set of states **T** and input character **a** and returns all the states that can be reached on given input character from all states in **T**.

Now using these 2 functions, we can perform the transformation:

1. The start state of DFA is created by taking the **Epsilon closure** of the start state of the NFA
2. For each new DFA state, perform the following for each input character:
  - i. Perform **move** to the newly created state
  - ii. Create new state by taking the **Epsilon closure** of the result (i). Note that here we could get a state, which is already present in our set. This will result in a set of states, which will form the new DFA state. Note that here from one or many NFA states, we are constructing a single DFA state.
3. For each newly created state, perform step 2.
4. Accepting states of DFA are all those states, which contain at least one of the accepting states from NFA. Keep in mind that we are here constructing a single DFA state from one or many NFA states.

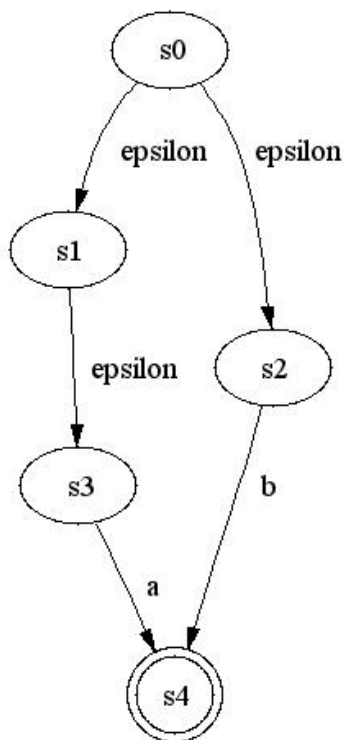
Simple enough? If not, then read further. Following is the pseudo code found on the pages 118-121 of the book "Compilers - Principles, Techniques and Tools" by Aho, Sethi and Ullman. The algorithm below is the equivalent to the algorithm above but expressed in a different way. First, let's define the **Epsilon Closure** function:

Hide Copy Code

```
S EpsilonClosure(T)
{
    push all states in T onto the stack
    initialize result to T
    while stack is not empty
    {
        pop t from the stack
        for each state u with an edge from t to u labeled epsilon
        {
            if u is not in EpsilonClosure(T)
            {
                add u to result
                push u onto the stack
            }
        }
    }
    return result
}
```

Basically, what this function does is, goes through all the states in **T** and checks what other states can be reached from these on no input. Note that each state can reach at least one state on no input, namely itself. Then the function goes through all these resulting states and checks for further transitions on no input. For example, let us look at the following:





If we would call Epsilon transition on a set of states  $\{s_0, s_2\}$  the resulting states would be  $\{s_0, s_2, s_1, s_3\}$ . This is because from  $s_0$ , we can reach  $s_1$  on no input, but from  $s_1$ , we can reach  $s_3$  on no input, so from  $s_1$  we can reach  $s_3$  on no input.

Now that we know how the Epsilon transition works, let us look at the pseudo code to transform an NFA to a DFA:

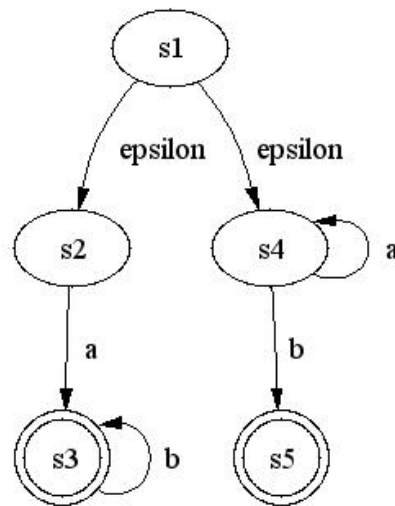
[Hide](#) [Copy Code](#)

```

D-States = EpsilonClosure(NFA Start State) and it is unmarked
while there are any unmarked states in D-States
{
    mark T
    for each input symbol a
    {
        U = EpsilonClosure(Move(T, a));
        if U is not in D-States
        {
            add U as an unmarked state to D-States
        }
        DTran[T,a] = U
    }
}
  
```

Finally the  $DTran$  is the DFA table, equivalent to the NFA.

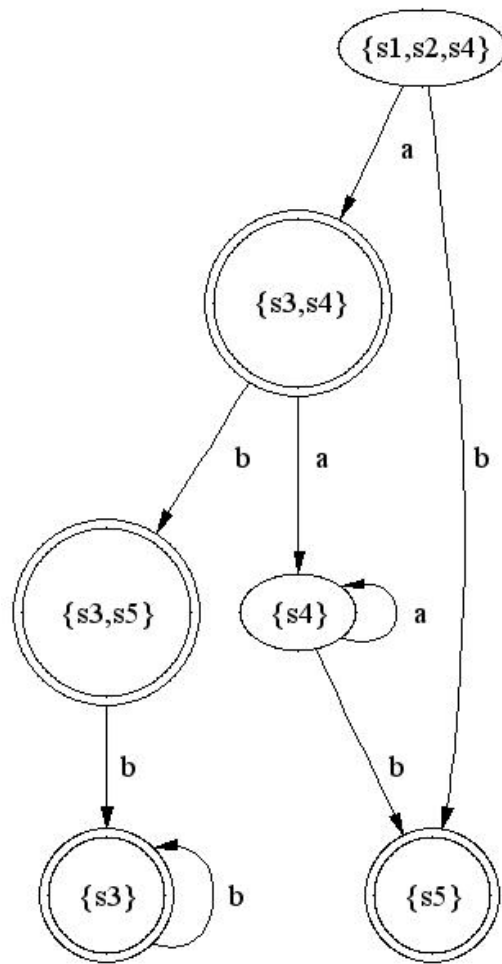
Before we go to the next step, let us convert an NFA to a DFA by hand, using this process. If you want to master this process, I would strongly suggest that you perform more similar transformations using this method. Let's convert the following NFA to its corresponding DFA using subset construction algorithm:



Using the subset construction algorithm, we would do following (Each newly created state will be **bolded**):

1. Create start state of DFA by taking epsilon closure of the start state of NFA. This step produces the set of states:  $\{s1, s2, s4\}$
2. Perform `Move('a',  $\{s1, s2, s4\}$ )`, which results in set:  $\{s3, s4\}$
3. Perform `EpsilonTransition( $\{s3, s4\}$ )`, which creates a new DFA state:  $\{s3, s4\}$
4. Perform `Move('b',  $\{s1, s2, s4\}$ )`, which results in set:  $\{s5\}$
5. Perform `EpsilonTransition( $\{s5\}$ )`, which creates new DFA state:  $\{s5\}$
6. **Note:** Here we must record 2 new DFA states  $\{s3, s4\}$  and  $\{s5\}$ , together with DFA starting state  $\{s1, s2, s4\}$ . Also we must record transition on character `a` from  $\{s1, s2, s4\}$  to  $\{s3, s4\}$  and on character `b` from  $\{s1, s2, s4\}$  to  $\{s5\}$ .
7. Perform `Move('a',  $\{s3, s4\}$ )`, which returns:  $\{s4\}$
8. Perform `EpsilonTransition( $\{s4\}$ )`, with result:  $\{s4\}$
9. Perform `Move('b',  $\{s3, s4\}$ )`, which results in set:  $\{s3, s5\}$
10. Perform `EpsilonTransition( $\{s3, s5\}$ )` with result:  $\{s3, s5\}$
11.  $\{s3, s4\} \rightarrow \{s4\}$  on `a`
12.  $\{s3, s4\} \rightarrow \{s3, s5\}$  on `b`
13. Perform `Move('a',  $\{s5\}$ )`, which returns an empty set, so we don't need to check Epsilon transitions
14. Perform `Move('b',  $\{s5\}$ )`, which returns an empty set, so forget it.
15. Perform `Move('a',  $\{s4\}$ )`, which returns:  $\{s4\}$ . But this is **not** a new state, so forget it. However we must record the transition:
16.  $\{s4\} \rightarrow \{s4\}$  on `a`
17. Perform `Move('b',  $\{s4\}$ )` which returns:  $\{s5\}$
18. Perform `EpsilonTransition( $\{s5\}$ )` which returns:  $\{s5\}$  (not new, but we must record transition)
19.  $\{s4\} \rightarrow \{s5\}$  on `b`
20. Perform `Move('a',  $\{s3, s5\}$ )` which returns an empty set, so forget it.
21. Perform `Move('b',  $\{s3, s5\}$ )` which produces:  $\{s3\}$
22. `EpsilonTransition( $\{s3\}$ )` produces:  $\{s3\}$ , a **NEW DFA state**
23.  $\{s3, s5\} \rightarrow \{s3\}$  on `b`
24. `Move('a',  $\{s3\}$ )` is an empty set
25. `Move('b',  $\{s3\}$ )` is  $\{s3\}$  which is not new but transition must be recorded!
26.  $\{s3\} \rightarrow \{s3\}$  on `b`

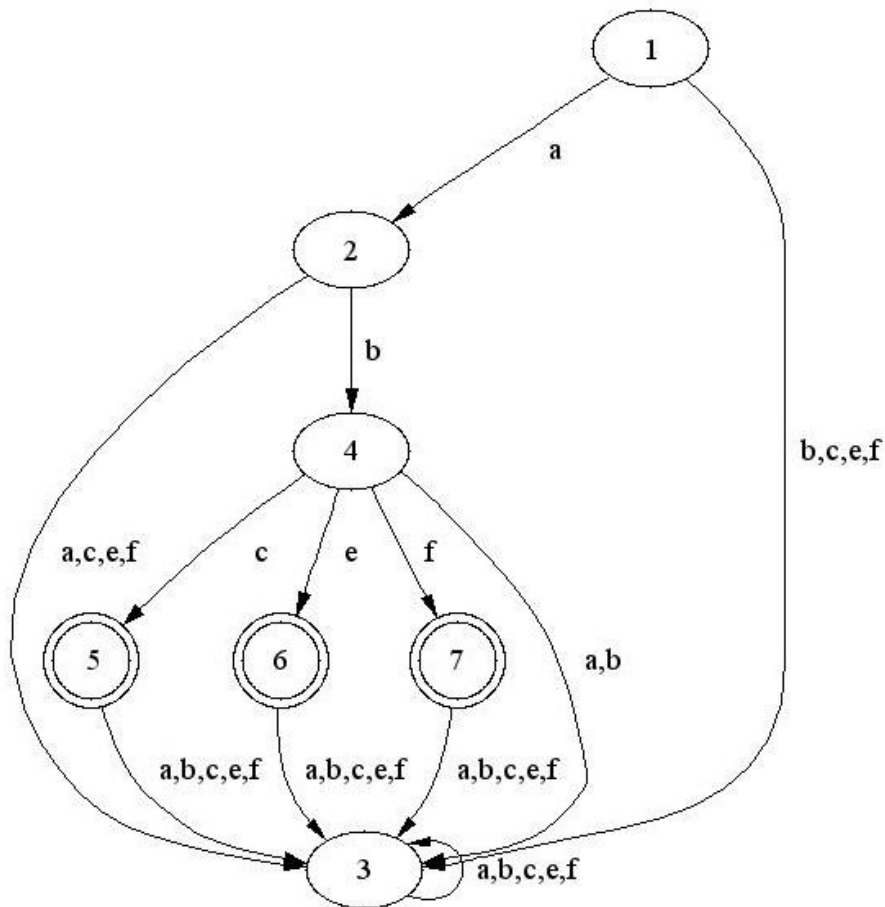
There are no new states, so we are done. Following is the drawing of the DFA:



The starting state is  $\{s1, s2, s4\}$ , because that is  $\text{EpsilonClosure}(\text{Starting state of NFA})$ . The accepting states are  $\{s5\}$ ,  $\{s3, s4\}$ , and  $\{s3, s5\}$  because they contain  $s3$  and/or  $s5$ , which are accepting states of the NFA.

## DFA Optimization

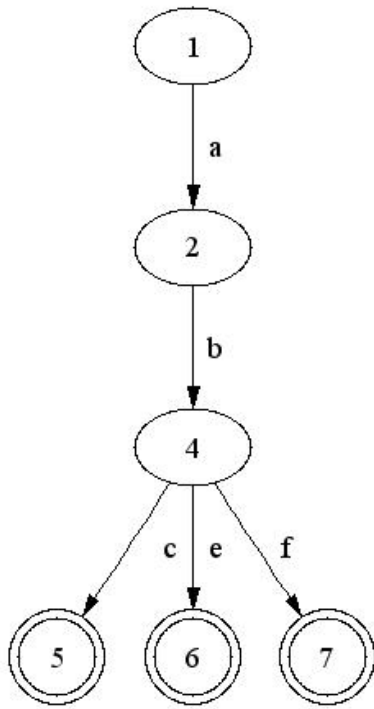
Now that we have all the knowledge to convert a regular expression into an NFA and then convert NFA to an equivalent DFA, we actually could stop at this point and use it for patterns matching. Originally when I planned to write this article, in order to keep it as simple as possible showing only principles, DFA optimization was not taken into account. But then it occurred to me that, first of all for large regular expressions, we are creating very large NFAs (by the nature of Thompson's algorithm), which in turn occasionally creates complex DFAs. If we would search for patterns, this might slow us considerably down, so I decided to include the optimization as a part of the regular expression parser. The optimization here is not a complicated one. So let's look at the following example:



If we look at this DFA, we notice that state 3 is first of all not a final state. Additionally we notice that there are no outgoing edges from this state except for the loop. In other words, once we get into state 3, there is no chance to get to an accepting state. This is due to the fact that a DFA, besides the fact that it has a unique path for each accepting string and does not contain the Epsilon transitions, it also must have a transition on all input characters from a particular state (Here all input characters mean, the set of possibly accepting input characters. For example:  $a|b|c$ , the set of input characters here is  $\{a, b, c\}$ ). Here is where we abuse the math a little bit, in order to make a DFA simpler. By deleting the state 3, our DFA becomes simpler, and it still accepts same set of patterns. In this case, our DFA is not anymore exactly a DFA. If you are asking yourself, why is this important, well the answer is: It is not! At least for us! We will use this very basic optimization mechanism to delete all the states with such characteristics and so we will obtain a smaller and compacter DFA for pattern matching. To summarize, we will delete states (and all transitions leading to these states from other states) with following characteristics:

1. State is not an accepting state
2. State does not have any transitions to any other different state.

So the result is following:



The DFA above, definitely seems to be smaller than the the previous one. I will still call this a DFA, despite the fact that it is not really a DFA.

## Using the results from parts above

Finally we are ready to use all of the parts from above, to match some text patterns. Once we have the DFA, all we need to do is to take an input character and run it against the starting state. Here is the pseudo code to do that:

Hide Shrink ▲ Copy Code

```

Find(string s)
{
    for each character c in s
    {
        for each active state s
        {
            if there is transition from s on c
            {
                go to the next state t
                if t is accepting state
                {
                    record the pattern
                }
                mark t as active
            }
            mark s as inactive
        }

        if there is transition from starting state on c
        {
            go to the next state s
            if s is accepting state
            {
                record the pattern
            }
            mark s as active
        }
    }
}
  
```

The code above can be found in the `Find(...)` function of the regular expression class. To keep track of active states, I use a linked list, so I can quickly add and delete states that are active/inactive respectively. After all characters are processed, all results are stored in a vector, which contains pattern matches. Using the functions `FindFirst(...)` and `FindNext(...)`, you can traverse through the results. Please refer to the documentation of the `CAG_RegEx` class for information on how to use the class. Also, at this point, I have to stress that the demo program loads the complete file into the rich edit control and then when searching is done, it

stores it into a string, passing it as an argument to the `FindFirst` function. Depending on your RAM size, I would avoid loading of huge files, because it could take a lot of time to copy the data from one string to another, because of the use of virtual memory. Like I said earlier, the program is designed to show the principles behind pattern matching in text files. Depending on time, future releases might incorporate a more complete regular expression parser that searches through files of any size and delivers the results in different ways.

At this point, for the completeness of the article, I must note that there is a way of converting a regular expression directly into a DFA. This method is not explained here yet, but if time permits, it will be in future articles (or article updates). Additionally, there are different ways of constructing an NFA from regular expressions.

## Final Words

Well, that's it! I hope you enjoyed reading the article as much as I enjoyed writing it. Please use the demo code in any kind of applications, but give me the credit where deserved. If you want to build a more complete library, using the demo code presented here, please send me a copy of your additions.

**Note:** The class `CAG_RegEx` contains two functions `SaveDFATable()` and `SaveNFATable`, which in debug mode save the NFA and DFA to `c:\NFATable.txt` and `c:\DFATable.txt` respectively. As the names already reveal, these are NFA and DFA tables. Additionally, the class has functions `SaveNFAGraph()` and `SaveDFAGraph()`, which in debug mode create 2 files `c:\NFAGraph.dot` and `c:\DFAGraph.dot`. These files are simple text files, containing the instructions for drawing these graphs using Graphviz (Check out the reference 4 below).

## References & Tools Used

1. "Discrete Mathematics" - Richard Johnsonbaugh (Fifth Edition)
2. "Discrete Mathematics and Its Applications" - Kenneth H. Rosen (Fourth Edition)
3. "Compilers - Principles, Techniques and Tools" - Aho, Sethi and Ullman
4. Graphviz from ATT (Tool for drawing of any kind of graphs). You can find it [here](#).

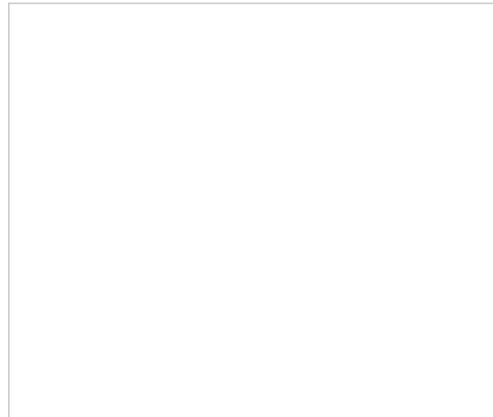
## License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

## Share


EMAIL



## About the Author



### Amer Gerzic

President Infinity Software Solutions, LLC.  
United States 

Originally from Bosnia and Herzegovina, but lived for 6 years in Germany where I did majority of education, then moved to US, where I live since 1999. I like programming, computers in general, but also Basketball, Soccer, Tennis, and many other things. Masters graduate from Grand Valley State University in CIS and working as a full time software developer. Please visit my website [www.amergerzic.com](http://www.amergerzic.com)

# Comments and Discussions

You must <a href="#">Sign In</a> to use this message board.		
Search Comments <input type="text"/>		<input type="button" value="Go"/>
<input checked="" type="checkbox"/> Profile popups	Spacing <input type="button" value="Relaxed"/>	Noise <input type="button" value="Medium"/>
Layout <input type="button" value="Normal"/>		Per page <input type="button" value="50"/>
		<input type="button" value="Update"/>
First Prev Next		
<b>Message Removed</b>	Member 11343411	5-Jan-15 5:15
<b>Java Code Please :)</b>	ryanjerskine	25-Oct-14 16:28
<b>Regular expression</b>	Suffian Nizami	23-Oct-14 8:38
<b>Plz mail me the java code</b>	Member 11045397	29-Aug-14 3:56
Re: Plz mail me the java code	Amer Gerzic	16-Sep-14 4:38
<b>Rgex to NFA implementation in JAVA</b>	madhur mehta	18-Jul-14 18:00
Re: Rgex to NFA implementation in JAVA	Amer Gerzic	16-Sep-14 4:38
<b>Lots of thanks..</b>	Member 10255374	10-Sep-13 6:16
Re: Lots of thanks..	Amer Gerzic	16-Sep-14 4:39
<b>Errors in the Thompson's Algorithm [modified]</b>	mnicky	9-Dec-12 5:27
Re: Errors in the Thompson's Algorithm	Amer Gerzic	9-Dec-12 10:59
<b>Counting</b>	Kuzayfa	5-Dec-12 21:33
Re: Counting	Amer Gerzic	6-Dec-12 3:10
<b>Awesome</b>	Aaron_Redmond	9-Mar-11 8:10
Re: Awesome	Amer Gerzic	12-Aug-11 5:07
<b>My vote of 1</b>	mbue	21-Jan-11 5:32
<b>Bad code</b>	mbue	21-Jan-11 5:31
<b>Algorithm for comparing regular expressions</b>	SonOfPirate	20-Aug-09 12:41
Re: Algorithm for comparing regular expressions	Amer Gerzic	21-Aug-09 3:21
<b>Thanks so much</b>	langtugacon	2-Mar-09 6:19
Re: Thanks so much	Amer Gerzic	2-Mar-09 6:32
Re: Thanks so much	Amer Gerzic	2-Mar-09 6:43
<b>C# implementation</b>	Mizan Rahman	5-Aug-08 3:20
Re: C# implementation	Amer Gerzic	5-Aug-08 3:34

<b>C# porting [modified]</b>	<b>Ugo Moschini</b>	<b>22-Jul-08 22:02</b>
Re: C# porting	Amer Gerzic	23-Jul-08 5:00
<b>C# Porting [modified]</b>	<b>Ugo Moschini</b>	<b>21-Jul-08 22:41</b>
Re: C# Porting	Amer Gerzic	23-Jul-08 5:00
Re: C# Porting	Ugomos	23-Jul-08 8:03
Re: C# Porting	Amer Gerzic	23-Jul-08 8:08
Re: C# Porting	Ugomos	23-Jul-08 8:26
Re: C# Porting	Amer Gerzic	23-Jul-08 8:31
Re: C# Porting	Ugomos	23-Jul-08 8:55
<b>Wild card support</b>	<b>Mizan Rahman</b>	<b>21-Jul-08 6:03</b>
Re: Wild card support	Amer Gerzic	22-Jul-08 2:59
Re: Wild card support [modified]	Mizan Rahman	22-Jul-08 4:52
Re: Wild card support	Amer Gerzic	22-Jul-08 6:43
Re: Wild card support	Mizan Rahman	23-Jul-08 2:08
Re: Wild card support	Amer Gerzic	23-Jul-08 4:58
Re: Wild card support [modified]	Mizan Rahman	23-Jul-08 22:31
Re: Wild card support	Amer Gerzic	24-Jul-08 5:51
Re: Wild card support	Mizan Rahman	24-Jul-08 22:32
Re: Wild card support	Amer Gerzic	25-Jul-08 3:17
<b>bug report</b>	<b>apen2007</b>	<b>25-May-08 20:23</b>
Re: bug report	Amer Gerzic	27-May-08 3:03
Re: bug report	apen2007	27-May-08 3:55
Re: bug report	Amer Gerzic	27-May-08 3:57
<b>NFA to DFA</b>	<b>john rohin</b>	<b>12-Apr-08 19:55</b>
Re: NFA to DFA	Amer Gerzic	14-Apr-08 3:09
<b>[Message Deleted]</b>	<b>S.M.H. Oloomi</b>	<b>4-Mar-08 4:33</b>
Last Visit: 31-Dec-99 19:00    Last Update: 26-Apr-15 5:07 <a href="#">Refresh</a> <a href="#">1</a> <a href="#">2</a> <a href="#">3</a> <a href="#">4</a> <a href="#">5</a> <a href="#">6</a> <a href="#">7</a> <a href="#">Next »</a>		

General  
 News  
 Suggestion  
 Question  
 Bug  
 Answer  
 Joke  
 Rant  
 Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.