What The Regex_Cat Dragged In

$$(a|b)^*a\underbrace{(a|b)(a|b)\cdots(a|b)}_{k-1 \text{ times}}.$$

Kyle Hamilton
COS210 - MCCC, Prof. Bostain
December 2013

Regular Expression:

From Wikipedia, the free encyclopedia

In computing, a **regular expression** (abbreviated **regex** or **regexp**) is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations. The concept arose in the 1950s, when the American mathematician *Stephen Kleene formalized the description of a regular language*, and came into common use with the Unix text processing utilities ed, an editor, and grep (global regular expression print), a filter.

Each character in a regular expression is either understood to be a metacharacter with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The **pattern sequence** itself is an expression that **is a statement in a language** designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text processing of general text files, specific textual forms, or of random input strings.

The **alphabet, Σ,** will consist of all printable (8bit) ASCII characters except the metacharacters. The meatcharacters in Regex_Cat are:

```
[ ] ( ) { } * + ? ^ $ | .
```

What's supported in Regex_Cat

Currently the most basic regular expression operations can be performed:

( ) [ ] * + ? |

To support the **POSIX Extended** standard, the following features have yet to be added:

Escapes:
\ Any special character preceded by an escape shall match itself.

Assertions:
'^' shall match the start of a line when used as the first character of an expression, or the first character of a sub-expression.

'$' shall match the end of a line when used as the last character of an expression, or the last character of a sub-expression.

Repetition:
a{n} Matches 'a' repeated exactly n times.
a{n,} Matches 'a' repeated n or more times.
a{n, m} Matches 'a' repeated between n and m times inclusive.

Wildcard:
The single character '.' when used outside of a character set will match any single character.

12 Character Classes:
[:alnum:], [:alpha:], [:blank:], [:cntrl:], [:graph:], [:print:], [:punct:], [:digit:], [:lower:], [:space:], [:upper:], [:xdigit:]
Ex: **[:alnum:]** matches Alphanumeric characters, as would: **[a-zA-Z0-9]** The latter form is supported in the current Regex_Cat.

What Regex_Cat does

1. Parse regex

A. Insert explicit concatenation operators. Regex_Cat uses a '•' which is not part of the first 128 ASCII characters. Care must be taken to compile Regex_Cat with UTF-8 encoding

B. Convert expression from infix to postfix form.

C. Save the Alphabet of the regex for later.

2. Build NFA

Convert Postfix expression to NFA

3. Convert to DFA

Use Subset construction to convert from NFA to DFA

Build DFA Transition Table

4. Simulate

Process the target string using the DFA Transition Table and Alphabet

Data Structures

ArrayList

A self expanding array. For storing data elements as well as iterating over said elements.

DoublyLinkedList

Similar to the ArrayList, but with the ability to iterate in both directions.

Stack

Great for bouncing stuff around by ‘pushing’ and ‘popping’ elements from stack to stack. The Infix to Postfix algorithm is a good example.

Directional List Graph

Used in the construction of the NFA, the graph stores **Edge** data. Each **Edge** has a **source vertex**, a **destination vertex**, and a ‘**weight**’ - in this case a character in the alphabet of the regex.

The graph is implemented as an **Adjacency List** is used, as opposed to an ‘Adjacency Matrix’, because the NFA is ‘sparse’. A sparse graph is one in which the number of edges is much less than the possible number of edges. The NFA vertices are connected by at most 2 edges, and in most cases, only 1 edge.

Directional Matrix Graph

The DFA on the other hand is a type of **Adjacency Matrix**, implemented as a **2-dimentional array**. This makes the regex engine algorithm simple and fast, with run-times of O(n). The potential trade-off is the DFA’s memory footprint caused by ‘state explosion’ - at worst, the DFA can contain 2^n - 1 number of states of the NFA (where ‘n’ is the number of states).

Algorithms

1. Infx To Postfix

Parses regular expression

2. Thompson-McNaughton-Yamada:

Constructs NFA

3. Subset Construction:

Converts NFA to DFA

Constructs Transition Table

4. Simulation Engine

Processes the target String

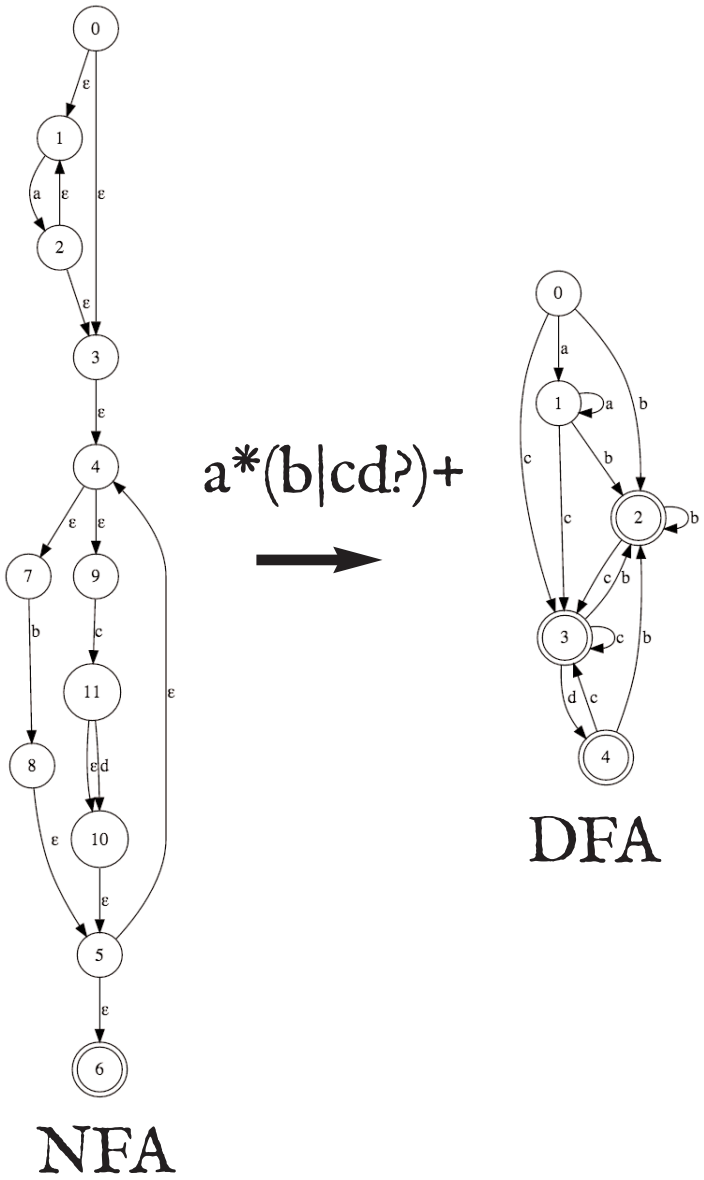


Figure: A really cool visual NFA to DFA simulator:
http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

Infix to Postfix: InfixToPostfix.java

The first step is to parse the expression into a form suitable to building the State Machine:

Since the input has no explicit concatenation operators, we first insert those. We use the '•' symbol, since it is not within the 8bit ASCII set we are supporting, hence it won't conflict with any regular charcters. We just have to careful when compiling the code to set the JVM to use UTF-8 encoding.

The next step is to convert the expression to 'Polish notation', or, **Postfix** form. The algorithm to convert an infix expression into a postfix expression uses a stack. The stack is used to hold operators. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

We also store the Alphabet of the regex in this step, so we can use it later to build the transition table, and process the target string.

Precendence

Collation-related bracket symbols [==] [::] [..]
Escaped characters \
Character set (bracket expression) []
Grouping ()
Single-character-ERE duplication * + ? {m,n}
Concatenation
Anchoring ^$
Alternation |

InfixToPostfix.java in our package:

```
private static final String OPERATORS = "[](){}*+?•^$|.";
private static final int[] PRECEDENCE = {6,6,5,5,4,4,4,4,4,3,2,2,1,1};
```

A summary of the rules:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Taken from: *http://csis.pace.edu/~wolf/CS122/infix-postfix.htm*

Infix to Postfix illustrated:

This illustration uses a + symbol for concatenation. Regex_cat uses + to denote "one or more", so our implementation uses a • as mentioned on the previous page.

A * (B + C * D) + E becomes A B C D * + * E +

current symbol	operator stack	postfix string
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
*	* (+ *	A B C
D	* (+ *	A B C D
)	*	A B C D * +
+	+	A B C D * + *
E	+	A B C D * + * E
		A B C D * + * E +

Taken from: *http://csis.pace.edu/~wolf/CS122/infix-postfix.htm*

Finite State Machine | Finite State Automaton

Taken from: *http://www.thefreedictionary.com/finite+state+machine*

A model of a computational system, consisting of a set of states (including a start state), an alphabet of symbols that serves as a set of possible inputs to the machine, and a transition function that maps each state to another state (or to itself) for any given input symbol. The machine operates by being fed a string of symbols, and moves through a series of states.

NFA (Nondeterministic Finite Automaton) vs. DFA (Deterministic Finite Automaton)

Taken from: *http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton*

A nondeterministic finite automaton (NFA), or nondeterministic finite state machine, is a finite state machine that (1) does not require input symbols for state transitions and (2) is capable of transitioning to zero or two or more states for a given start state and input symbol. This distinguishes it from a deterministic finite automaton (DFA), in which all transitions are uniquely determined and in which an input symbol is required for all state transitions. Although NFA and DFA have distinct definitions, all NFAs can be translated to equivalent DFAs using the subset construction algorithm,[1] i.e., constructed DFAs and their corresponding NFAs recognize the same formal language. Like DFAs, NFAs only recognize regular languages.

An NFA, similar to a DFA, consumes a string of input symbols. For each input symbol, it transitions to a new state until all input symbols have been consumed. Unlike a DFA, it is non-deterministic, i.e., for some state and input symbol, the next state may be one of two or more possible states.

An NFA is represented formally by a 5-tuple, (Q, Σ, Δ, q0, F), consisting of

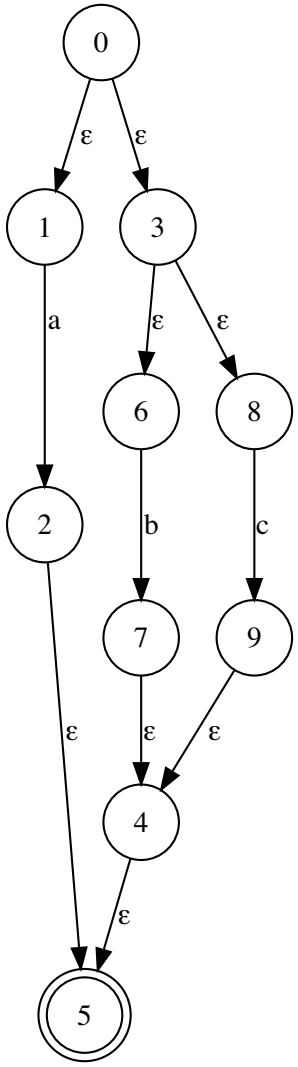
a finite set of states Q
a finite set of input symbols Σ (*the Alphabet of the regex*)
a transition relation $\Delta : Q \times \Sigma \rightarrow P(Q)$. (*the Edges*)
an initial (or start) state $q0 \in Q$
a set of states F distinguished as accepting (or final) states $F \subseteq Q$.

A DFA is also represented by a 5-tuple, (Q, Σ, δ, q0, F), with one difference.

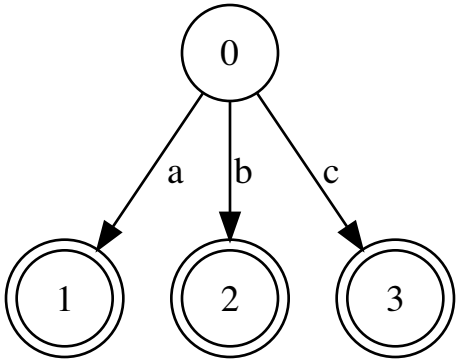
The DFA has a transition function ($\delta : Q \times \Sigma \rightarrow Q$) in place of the the transition relation Δ in the NFA.

Regular Expression: [a-c]

NFA with Epsilon States



Equivalent DFA



Regular expressions and NFAs are equivalent in power: every regular expression has an equivalent NFA (they match the same strings) and vice versa. DFAs are also equivalent in power to NFAs and regular expressions.

Thompson-McNaughton-Yamada NFA.java

The NFA for a regular expression is built up from partial NFAs for each subexpression, with a different construction for each operator, as well as epsilon states. The Start State and Accepting State of each partial NFA is used to attach the next partial NFA in the building process.

The postfix regular expression is used to build an NFA using a stack:

while(the regular expression is not exhausted)

When a symbol is encountered, an NFA corresponding to that symbol is constructed and pushed onto the stack.

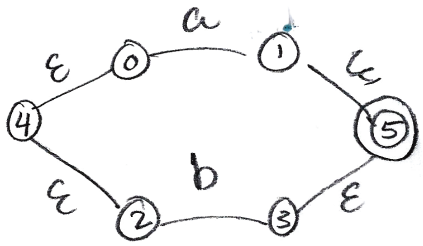
When an operator is encountered, a number of NFAs, consistent with the operator, is popped and used to construct a new NFA using the corresponding operator. A new Start state and a new Accept state is assigned.

That newly constructed NFA is then pushed onto the stack.

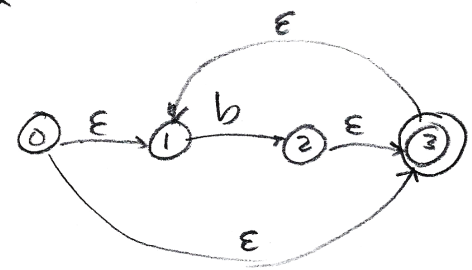
NFA construction operations

ab

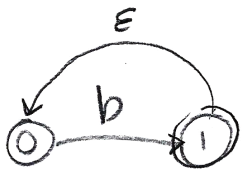
a|b



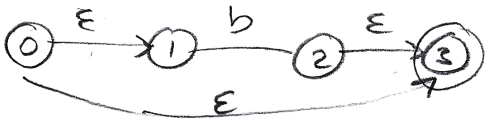
b*



b+



b?



Subset Construction: NFA to DFA

The Subset constrauction algorithm consists of the following functions:

getEclosure() calculates epslison closures of the NFA.

Each closure becomes a State *T* of the DFA, and is added to the list of DFA States *DS*.

move() assigns edges to DFA states by iterating over the e-closures.

Subset Construction:

```
while (there is an unmarked state T in DS){
        mark T
        for (each letter 'a' in the alphabet ){
                U = getEclosure(move(T,a))
                if(U is not already in DS){
                        add U to DS as an unmarked state
                }
                push transition [T,a]=U onto the dfaTransitionStack of DFA transitons
                ( implemented as an ArrayList of int[] = {T, a, U} )
        }
}
```

Subset Construction con't:

getEclosure(S):

```
start with the Start State of the NFA
push all states onto the States List S

for each state in S
        push state onto stack
        add state to eClosure list

        while(stack is not empty){
                pop stack
                if(the edge is an epsilon edge, and the destination edge doesn't already exist in the eClosure){
                        push the destination state onto the stack
                        add the destination state to the eClosure list
                }
        }
return the eClosure list
```

move(S, a)

```
for each state s in List S
        for each Edge in s in NFA
                if the weight == the letter 'a' of the Alphabet,
                        add desitnation state to list
return list
```

Subset Construction con't:

Construct the Transition Table from the list of DFA Transitions:

```
TransTable = new int[number of DFA States][length of the Alphabet];
while(!dfaTransitionStack.empty()){
        int[] trans = dfaTransitionStack.pop();
                TransTable [trans[0]][trans[1]] = trans[2];
        }
```

Sample Output:

```
NFA: [(0, 1): c], [(1, 2): ], [(2, 3): a], [(3, 7): ], [(4, 5): b], [(5, 7): ], [(6, 4): ], [(6, 0): ]
start: 6 accept: 7
NFA to DFA State map: [[6, 4, 0], [1, 2], [5, 7], [3, 7]]
DFA Transitions:
[ 1 1 3 ]
[ 0 2 2 ]
[ 0 0 1 ]
DFA Transition Table (2D array of: DFA States x Alphabet)
(-1 denotes unreachable states)
    c   a   b
   ---------------
0 | 1 | -1| 2 |
   ---------------
1 | -1| 3 | -1|
   ---------------
2 | -1| -1| -1|
   ---------------
3 | -1| -1| -1|
   ---------------
dfa final states: [2, 3]
Evaluating 'ca|b' using 'ca'
```

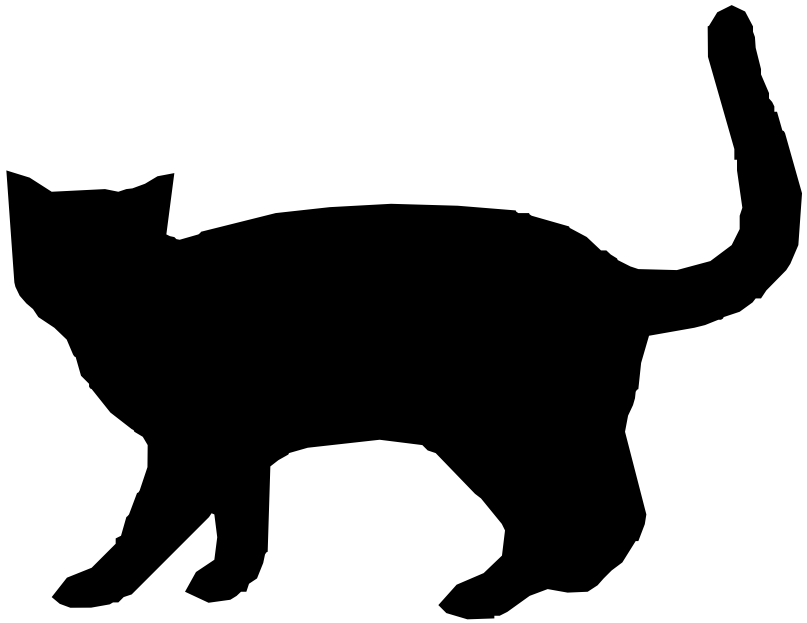
Simulation

```
// exact match:
    private boolean eatDFA(int[][] table, String lang, ArrayList<Integer> finalStates, String target) throws IOException{
        BufferedReader br = stringToBR(target);
        int c;
        int state = 0;
        while (((c = br.read()) != -1)) {
            int input = lang.indexOf((char)c);
            if(input != -1){ // only continue if input char is in the language
                if (table[state][input] == -1){
                    // unreachable state
                    return false;
                }else if ( finalStates.contains(table[state][input])  && !br.ready()){
                    return true;
                    // Table indicates that for this state, we accept the input given
                    // and we reached the end of the string
                }
            // Advance to next state.
            state = table[state][input];
            }else{
                break; // it's not even in the language, break and return false
            }
        }
        // we reached the end of the input,
        return false;
    }
```

Some helpful and some not so helpful resources

http://csis.pace.edu/~wolf/CS122/infix-postfix.htm
http://eli.thegreenplace.net/files/docs/forays/col6.html
http://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf
http://swtch.com/~rsc/regexp/regexp1.html
http://www.regular-expressions.info/posixbrackets.html
http://xlinux.nist.gov/dads//HTML/sparsegraph.html
http://binarysculpting.com/2012/02/11/regular-expressions-how-do-they-really-work-automata-theory-for-programmers-part-1/
http://binarysculpting.com/2012/02/15/converting-dfa-to-nfa-by-subset-construction-regular-expressions-part-2/
http://www.cs.hunter.cuny.edu/~saad/courses/csci493.66/hw/hw3.pdf
http://hackingoff.com/compilers/regular-expression-to-nfa-dfa
http://web.cecs.pdx.edu/~harry/compilers/slides/LexicalPart3.pdf
http://ezekiel.vancouver.wsu.edu/~cs317/nfa/RegEx/src/regex/NFA.java
https://www.clear.rice.edu/comp412/Lectures/L06LexIII-Subset-1up.pdf
http://www.cs.uaf.edu/~cs631/notes/strings/node6.html

http://www.youtube.com/watch?v=xlrw5Y8SXjA
http://www.youtube.com/watch?v=59quXH2yUFo
http://www.youtube.com/watch?v=iafR7TUZF8I
http://www.youtube.com/watch?v=fYmi2okBG40
http://www.youtube.com/watch?v=3tdePzdkYXE



- The End -