Journal of Applied Sciences Research, 2(12): 1359-1362, 2006
© 2006, INSInet Publication

Regular Expression to Finite State Machine

Ayesha Hanif, Zaheer Ahmed, Muhammad Kashif Hanif, Ali Aqdas

Department of Computer Science, University of Agriculture, Faisalabad.

Abstract: It is well established fact that each regular expression can be transformed into nondeterministic finite automata with or without λ transitions. In this paper, we examine the construction of NFA from regular expression using the Thompson and Glushkov construction. We will check that how these two methods are related to each other.

Key word: Regular expression, finite state machine

INTRODUCTION

There are many strategies for building a recognizer from a regular expression, each with its own strength and weaknesses. We use regular expression for pattern matching and searching. To implement regular expressions, we usually convert them to finite state machines. Thus, the size of a finite state machine and time taken to construct it are important measures of efficiency. In addition, the simplicity of a machine is also a criterion in some situations since it can lead to simple representations. Thompson construction produces the finite state machine that has the size linear in the size of given expressions and does so in linear time. But it also gives machines that have at most two transitions into and at most two out of each state.

A second construction that is older than Thompson's construction is Glushkov construction. It is a two-step process. First, inductively compute three sets of symbols and then, second, compute the machine directly from these sets. We will check the relationship that exists between Thompson and glushkov construction methods.

Notation and terminology: We review some basic definition. A word is a string or sequence of characters over a finite alphabet ∑ . The empty word is denoted λ and the set of all words built on ∑ (λ included) is ∑*. A word x ϵ ∑* is a factor (or substring) of p if p can be written p = uxv, u, v ϵ ∑*. A factor x of p is called a suffix (respectively prefix) of p is p = ux (respectively p = xv), u ϵ ∑*.

We define also the language to denote regular expressions. Union is denoted with the infix "+", Kleene closure with "*" and concatenation simply by putting the sub expressions one after the other. Parentheses are used to change the precedence, which is normally "*",".", "+".

We inductively define a regular expression E over ∑ as follows: first, we have three base cases: E= ø, E = λ and E= a, where λ and ø denote null string and empty set and a is in ∑. Second we have four inductive cases: E = (F + G) and E = (F. G), where F and G are regular expressions, and E = (F*) and E = (F), where F is regular expression. We define the language of E inductively as follows: For a regular expression E, define L(E)= ø, if E = ø; L(E)= { λ } , if E = λ ; L (E) = {a}, if E = a; L(E) = L(F) U L(G), if E = (F + G); L(E) = L(F)L(G), if E = (F. G); L(E)=L(F*), if E=(F*); and L(E) = L(F), if E = (F).

In its more general form, a Finite State Automata (FSA)[3], which will be M denoted _ in this paper, is defined as an abstract machine consisting of:

- A set of states $Q = \{q_0,q_1,q_2,\ldots..q_n\}$ including the initial state and final state, also referred to as accepting state (in case of recognizers). Variants of this include machines having multiple start states and multiple accepting states. In this paper, a specific state visited at time t is denoted q_t.
- A set ∑ of input symbols.
- A state transition function $q_t = f(Y_t, q_{t-1})$ which takes current input event and previous state and returns the next state.

A deterministic FSA is one in which transition and output functions are deterministic, implying that the output event and next state are uniquely determined by a single input event (i.e., there is exactly one transition for each given input event and state).

A non-deterministic FSA is one where the next state depends not only on the current input event, but also on a number of subsequent input events. However, it is often possible to transform a non-deterministic FSA into a deterministic FSA, at the cost of a significant increase of the possible number of input symbols.

Corresponding Author: Muhammad Kashif Hanif, Department of computer sciences, University of Agriculture, Faisalabad.

Regular Expression to Finite State Machines: We require that a machine M obtained from regular expression E satisfies the condition that L(M) = L(E); that is, M and E are equivalent. We first check the Thompson construction and then the Glushkov method.

The Thompson Construction: Thompson designed this construction in 1968[1] to compile regular expressions into a form suitable for text searching. The construction of finite state machine has a number of properties:

- The size of the machine is at most three times the size of given regular expression.
- Each state has at most two transitions leaving it and at most two transitions entering it.
- There is one start state that is only exiting; that is, there are no transitions that enter it.
- There is one final state that is only entering; that is, there are no transitions that leave it.

The algorithm[2] for conversion in finite state machine is syntax directed in that it uses the syntactic structure of regular expression to guide the construction process. The cases in the algorithm follow the cases in the definition of a regular expression. We first show how to construct automata to recognize λ and any alphabet symbol. Then, we show how to construct automata for expressions containing an alternation, concatenation, or Kleene closure operator. As construction proceeds, each step introduces at most two new states, so the resulting NFA constructed for a regular expression has at most twice as many states as there are symbols and operators in regular expression.

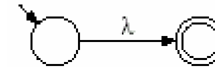
Algorithm (Thompson's Construction.) An NFA from Regular Expression:

Input: A regular expression E over an alphabet ∑.
output: An NFA N accepting L(E).

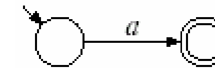
Method. We first parse E into its constituent sub expressions. Then, using rules below, we construct NFA's for each of the basic symbols in E (those that are either λ or an alphabet symbol). It is important to understand that if a symbol *a* occurs several times in E, a separate NFA is constructed for each occurrence. Then, guided by the syntactic structure of the regular expression E, we combine these NFA's inductively using rule (4) below until we obtain the NFA for entire expression. Each intermediate NFA produced during the course of construction corresponds to a sub expression of E and has several important properties: it has exactly one final state, no edge enters the start state, and no edge leaves the final state.

1. For E = ø

2. For E = λ

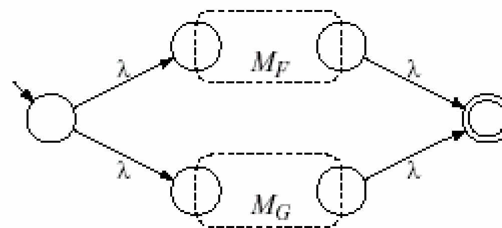


3. For an in Σ, Construct the NFA



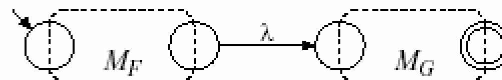
4. Suppose N(F) and N(G) are NFA's for regular expressions F and G.

a) For regular expression F+G, Construct the NFA



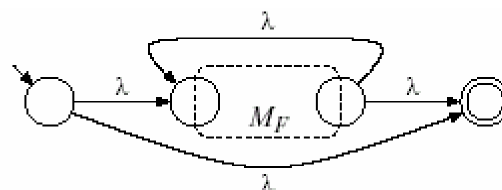
There is a transition on λ from new start state to start states of N(F) and N(G). There is a transition on λ from accepting states of N(F) and N(G) to the new accepting state. The start and accepting states of N(F) and N(G) are not start and accepting states of N(F+G). Note that any path from new start to new final state must pass through either N(F) or N(G).exclusively. Thus, the composite NFA recognizes L(F) U L(G).

b) For regular expressions E=FG, construct the composite NFA N(FG):



The start state of N(F) becomes the start state of the composite NFA and the accepting state of N(G) becomes the accepting state of the composite NFA. The accepting state of N(F) is merged with the start state of N(G); that is, all transitions from start state of N(G) becomes transitions from the accepting state of N(F). The new merged state loses its status as a start or accepting state in the composite NFA. The composite NFA recognizes L (F) L (G).

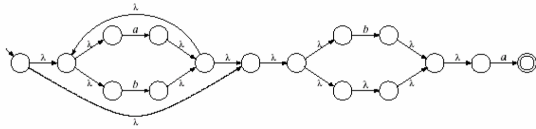
c) For the regular expression F*, construct composite NFA N(F*):



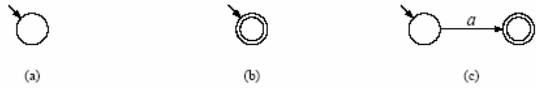
In composite NFA, we can go directly from start to final state, along an edge labeled λ, representing the fact that λ is in (L (F))*, or we can go from start to final state passing through N(F) one or more times. Clearly, the composite NFA recognizes (L (F))*.

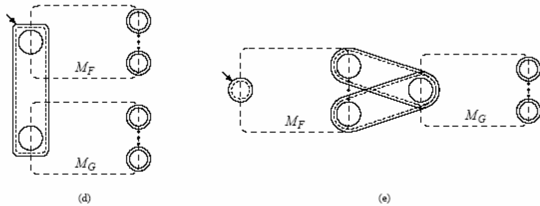
d) For the parenthesized regular expression (F), use N(F) itself as the NFA.

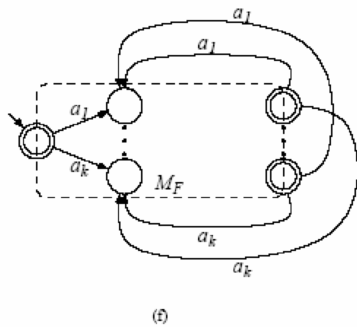
Given a regular expression E, we denote its Thompson machine by M^T_E. We give the result of Thompson construction on the regular expression (((a+b)*).((b+λ).a)).



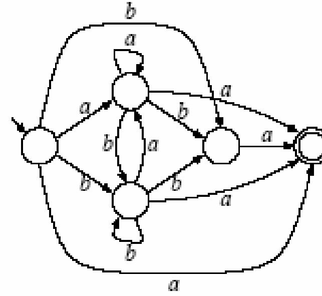
The Glushkov Construction: Champernaud[4] proved that the Glushkov construction can be made inductive in the same style as Thompson construction.







The construction of the machine M_{F+G} from M_F and M_G combines the two machines by identifying (or merging) their start states. The inductive step for F.G identifies the start state of M_G separately with each final state of M_F. The effect of identification is that each, previously final state, state in M_F has transitions to all states that M_G 's start state has transitions to and, also, M_G 's start state disappears. If M_G 's start state was a final state , then M_F 's final state remain final; otherwise, the become non final. The inductive step for F* is similar to that for product. It identifies the start state of M_F separately with each final state of M_F. The difference is that start state does not disappear, it becomes final state and all other final states remains final states. The effect of the identification is that each final state in M_F is given transitions to all states that M_F 's start state has transitions to. The Glushkov machine of previous example is:



The Glushkov machine has following properties:

- There is one start state that is only exiting.
- It has no null transitions.
- For each state p, all transitions into p have the same label.
- The size of Glushkov machine of regular expression E is, in the worst case, $O(|E|^2)$.

The Glushkov–Thompson Relationship: There are a number of issues raised by the Glushkov and Thompson constructions. The most basic is: *how are the Glushkov and Thompson machines related?* We demonstrate it by proving that every Thompson machine is a disguised Glushkov machine and vice versa.

We show that every Glushkov machine is hidden in the corresponding Thompson machine. We introduce a null-elimination transformation that removes null transitions and preserves some specific properties of the machine. When applied repeatedly to a Thompson machine to remove all null transitions, it yields corresponding Glushkov machine. we now shows the Glushkov construction of a Thompson machine. Let Γ be a new symbol that is not in Σ. Now, define the Γ expansion of a regular expression E over Σ to be a regular expression E_Γ over Σ U {Γ} that is defined inductively as follows:

$E_\Gamma = \varnothing$, if $E = \varnothing$.

$E_\Gamma = \Gamma$, if $E = \lambda$.

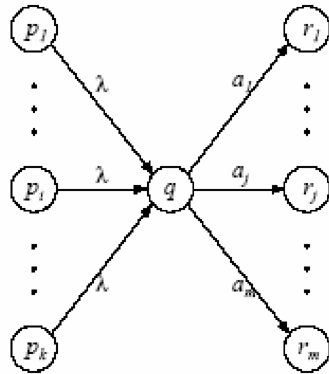
$E_\Gamma = a$, if $E = a$.

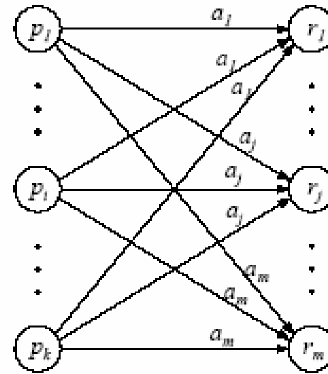
$E_\Gamma = (((\Gamma . F_\Gamma) + ((\Gamma . G_\Gamma)). \Gamma)$, if $E = (F + G)$.

$E_\Gamma = ((F_\Gamma . \Gamma) . G_\Gamma)$, if $E = (F . G)$.

$E_\Gamma = (((F_\Gamma . \Gamma)^* . \Gamma)$, if $E = (F^*)$.

$E_\Gamma = (F_\Gamma)$, if $E = (F)$.





When we apply the Glushkov construction to an empty-free expression E_Γ, the machine M_Γ that now we obtain gives the Thompson machine for E if we replace all appearances of Γ with the null string. So, for our above example the equivalent Glushkov machine is:

REFERENCES

1. Thomson, K., 1968. Regular expression search algorithm. Communications of ACM, 11: 419-422.
2. Alfred, V. Aho, Ravi Sethi and Jeffery D. Ullman., Compilers Principles, Techniques and Tools. pages 121-125, Addison Wesley Publishing Company.
3. Danial, I. and A. Cohen, Introduction to computer theory, second edition. Pages 35-36, 53-70.
4. Champernaud, J.M., 1997. From a regular expression to an automata. Rapport LIR-97.08 Laboratoire d'Informatque de Rouen, Université of de Rouen, France.
5. Dora Giammarresi, Jean-Luc Ponty, Derick Wood. A reexamination of the Glushkov and Thompson constructions.