OPTIMIZATION OF DETERMINISTIC FINITE AUTOMATON BASED PATTERN MATCHING IN LEXICAL ANALYSIS OF COMPILER

Vaikunth Pai*.
**Dept. of Information Technology, SrinivasInstitute of Management Studies, Pandeshwar, Mangalore-01, India, vpaistar@yahoo.com*

ABSTRACT

A compiler is a program that reads a program written in one language - the source language - and translates it into an equivalent program in another language - the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. A phase is an independent task in the compilation process, which transforms the source program from one representation to another.

The process of compilation starts with the first phase called lexical analysis. In this phase the input is scanned completely in order to identify the tokens. The token structures are recognized with the help of some diagrams. These diagrams are known as finite automata and to construct finite automata, regular expressions are used. These diagrams can be translated into a program for identifying tokens. The first goal of this research is to implement and optimize pattern matchers constructed from regular expressions for lexical phase of the compilation process. It will be suitable for inclusion in a Lex compiler because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA along the way.

The second goal of this research is to minimize the number of states of any DFA, so it can be used to reduce the size of a DFA-based pattern matcher.

Keywords: DFA, NFA, FA, Lex, Regular Expression

Introduction:

Lexical analysis is a process of recognizing tokens from the source program. The process of recognition can be accomplished by building a model called Finite State Machine (FSM) or Finite Automaton (FA). The characteristics of token can be represented by mathematical notation called Regular Expression.

Motivation to this research is space and time requirements for determining whether an input string x is in the language denoted by a regular expression r using recognizers constructed from deterministic finite automata and nondeterministic finite automata and an implementation of lexical analyzer that avoids constructing the intermediate NFA explicitly so that it affect the space and time requirements of a DFA-based pattern matcher simulator.

Optimization of DFA-Based Pattern Matching:

In this section, we present concept that have been used to implement and optimize pattern matchers constructed from regular expressions. It is suitable for inclusion in a Lex compiler because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA along the way and minimizes the number of states of any DFA, so it can be used to reduce the size of a DFA-based pattern matcher.

Important States of an NFA

Let us call a state of an NFA important if it has a non-epsilon out-transition. The subset construction uses only the important states in a subset T when it determines epsilon-closure(move(T, a)), the set of states that is reachable from T on input a. The set move(s, a) is nonempty only if state s is important. During the construction, two subsets can be identified if they have the same important states, and either both or neither include accepting states of the NFA.

When the subset construction is applied to an NFA obtained from a regular expression, we can exploit the special properties of the NFA to combine the two constructions. The combined construction relates the important states of the NFA with the symbols in the regular expression. Thompson's construction builds an important state exactly when a symbol in the alphabet appears in a regular expression. For example, important states will be constructed for each a and b in (alb)*abb.

Moreover, the resulting NFA has exactly one accepting state, but the accepting state is not important because it has no transitions leaving it. By concatenating a unique right-end marker # to a regular expression r, we give the accepting state of r a transition on #, making it an important state of the NFA for r# In other words, by using the augmented regular expression (r)# we can

forget about accepting states as the subset construction proceeds; when the construction is complete, any DFA state with a transition on # must be an accepting state.

We represent an augmented regular expression by a syntax tree with basic symbols at the leaves and operators at the interior nodes. We refer to an interior node as a cat-node, or-node, or star-node if it is labeled by a concatenation, I, or * operator, respectively. Figure 3.1 shows a syntax tree for an augmented regular expression with cat-nodes marked by dots.

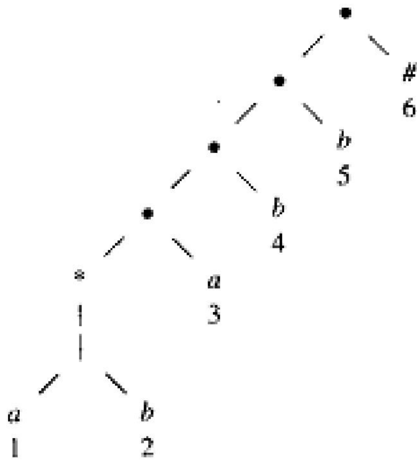

Figure 1: Syntax tree for (alb)*abb#

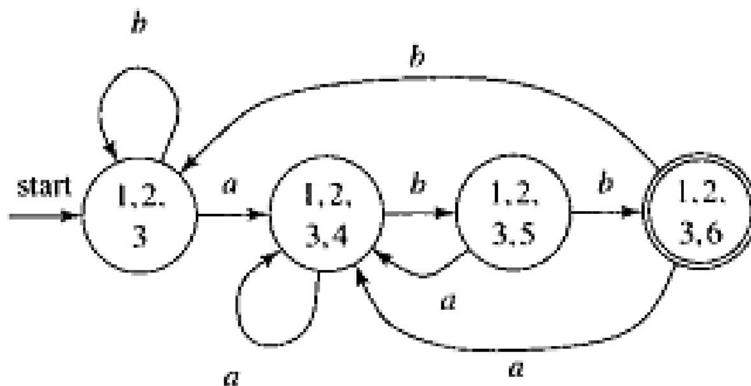

Figure 2: DFA accepting (alb)*abb#

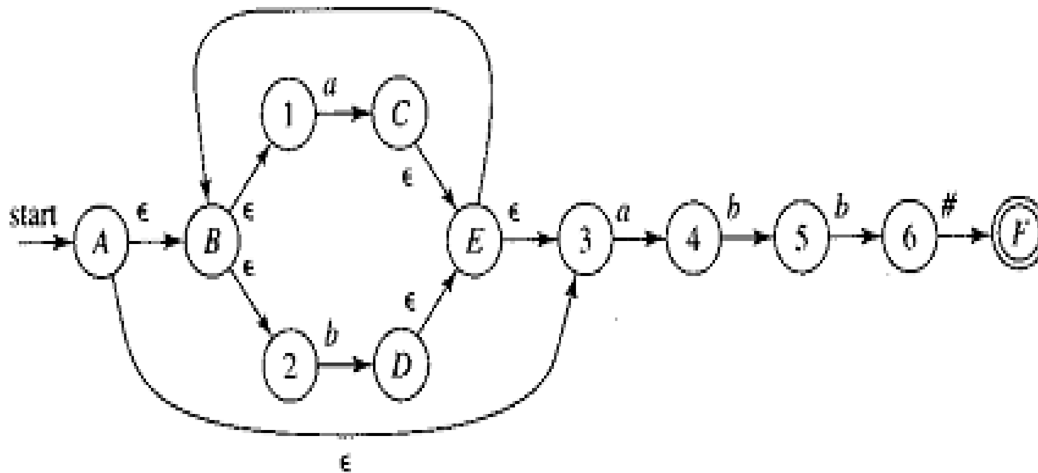

Figure 3: NFA accepting (alb)*abb#

Leaves in the syntax tree for a regular expression are labeled by alphabet symbols or by epsilon. To each leaf not labeled by epsilon we attach a unique integer and refer to this integer as the position of the leaf and also as a position of its symbol. A repeated symbol therefore has several positions. Positions are shown below the symbols in the syntax tree of Figure 1. The numbered states in the NFA of Figure 3 correspond to the positions of the leaves in the syntax tree in Figure 1. It is no coincidence that these states are the important states of the NFA. Non-important states are named by upper case letters in Figure 3. The DFA in Figure 2 can be obtained from the NFA in Figure 3 if we apply the subset construction and Identify subsets containing the same important states.

From a Regular Expression to a DFA:

In this section, we show how to construct a DFA directly from an augmented regular expression (r)#. We begin by constructing a syntax tree T for (r)# and then computing four functions: nullable, firstpos, lastpos and followpos, by making traversals over T. Finally, we construct the DFA from followpos. The functions nullable, firstpos, and lastpos are defined on the nodes of the syntax tree and are used to compute followpos which is defined on the set of positions.

The notion of a position matching an input symbol will be defined in terms of the function fallowpos positions of the syntax tree. If i is a position, then followpos(i) is the set of positions j such that there is input string... cd ... such that i corresponds to this occurrence of c and j to this occurrence of d.

In Figure 1 followpos(1) = {1, 2, 3}. The reasoning is that if we see an a corresponding to position 1, then we have just seen an occurrence of alb in the closure (a | b)*. We could next see

the first position of another occurrence of a | b, which explains why 1 and 2 are in followpos(1). We could also next see the first position of what follows (a | b)*, that is, position 3.

In order to compute the function followpos, we need to know what positions can match the first or last symbol of a string generated by a given subexpression of a regular expression. If r* is such a subexpression, then every position that can be first in r follows every position that can be last in r. Similarly, if rs is a subexpression, then every first position of s follows every last position of r.

At each node n of the syntax tree of a regular expression, we define a function firstpos(n) that gives the set of positions that can match the first symbol of a string generated by the subexpression rooted at n. Likewise, we define a function lastpos(n) that gives the set of positions that can match the last symbol in such a string.

In order to compute firstpos and lastpos, we need to know which nodes are the roots of subexpressions that generate languages that include the empty string. Such nodes are called nullable, and we define nullable (n) to be true if node n is nullable, false otherwise.

We can now give the rules to compute the functions nullable, firstpos, lastpos, and followpos. For the first three functions, we have a basis rule that tells about expressions of a basic symbol, and then three inductive rules that allow us to determine the value of the functions working up the syntax tree from the bottom; in each case the inductive rules correspond to the three operators, union, concatenation, and closure

The first rule for nullable states that if n is a leaf labeled epsilon, then nullable(n) is surely true. The second rule states that if n is a leaf labeled by an alphabet symbol, then nullable(n) is false. In this case, each leaf corresponds to a single input symbol, and therefore cannot generate epsilon. The last rule for nullable states that if n is a star-node with child cl then nullable(n) is true, because the closure of an expression generates a language that includes epsilon.

As another example, the fourth rule for firstpos states that if n is a cat-node with left child c1, and right child c2 and if nullable (c1) is true, then

firstpos(n) = firstpos(c1) U firstpos(c2)

otherwise, firstpos(n) = firstpos (c1). What this rule says is that if in an expression rs, r generates epsilon, then the first positions of s "show through" r and are also first positions of rs; otherwise, only the first positions of r are first positions of rs. The reasoning behind the remaining rules for nullable and firstpos are similar.

The function followpos(i) tells us what positions can follow position i in the syntax tree. Two rules define all the ways one position can follow another.

1. If n is a cat-node with left child c1 and right child c2, and i is a position in lastpos(c1), then all positions in firstpos(c2) are in followpos(i).

2. If n is a star-node, and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Figure 4 shows the values of firstpos and lastpos at all nodes in the tree of Figure 1; firstpos(n) appears to the left of node n and lastpos(n) to the right. For example, firstpos at the leftmost leaf labeleda is {1}, since this leaf is labeled with position 1. Similarly, firstpos of the second leaf is {2}, since this leaf is labeled with position 2. By the third rule in table 3.1, firstpos of their parent is {1, 2}.

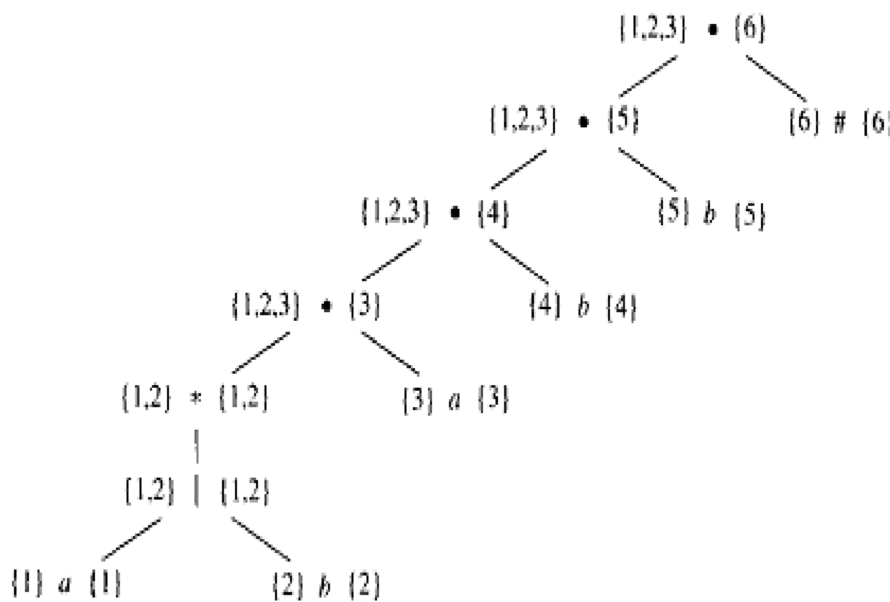

Figure 4: firstpos and lastpos for nodes in syntax tree for (a | b)*abb#.

The node labeled * is the only nullable node. Thus, by the if-condition of the fourth rule, firstpos for the parent of this node (the one representing expression (a | b)*a) is the union of {1, 2} and {3}, which are the firstpos's of its left and right children. On the other hand, the else-condition applies for lastpos of this node, since the leaf at position 3 is not nullable. Thus, the parent of the star-node has lastpos containing only 3.

Let us now compute followpos bottom up for each node of the syntax tree of Figure 3.4. At the star-node, we add both 1 and 2 to followpos(1) and to followpos(2) using rule (2). At the parent of the star-node, we add 3 to followpos(1) and followpos(2) using rule (1). At the next cat-node, we add 4 to followpos(3) using rule (1). At the next two cat-nodes we add 5 to followpos(4) and 6 to followpos(5) using the same rule. This completes the construction of followpos. Table 1 summarizes followpos.

Node	Followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-

Table 1: The function followpos

Minimizing the Number of States of a DFA:

An important theoretical result is that every regular set is recognized by a minimum-state DFA that is unique up to state names. In this section, we show how to construct this minimum-state DFA by reducing the number of states in a given DFA to the bare minimum without affecting the language that is being recognized. Suppose that we have a DFA M with set of states S and input symbol alphabet~. We assume that every state has a transition on every input symbol. If that were not the case, we can introduce a new "dead state" d, with transitions from d to d on all inputs, and add a transition from state s to d on input a if there was no transition from s on a.

We say that string w distinguishes state s from state t if, by starting with the DFA M in state s and feeding it input w, we end up in an accepting state, but starting in state t and feeding it input w, we end up in a nonaccepting state, or vice versa.

Our algorithm for minimizing the number of states of a DFA works by finding all groups of states that can be distinguished by some input string. Each group of states that cannot be distinguished is then merged into a single state. The algorithm works by maintaining and refining

a partition of the set of states. Each group of states within the partition consists of states that have not yet been distinguished from one another, and any pair of states chosen from different groups have been found distinguishable by some input.

Initially, the partition consists of two groups: the accepting states and the nonaccepting states. The fundamental step is to take some group of states, say A = {s1, s2,….,sk} and some input symbol a, and look at what transitions states s1, s2,….,sk have on input a. If these transitions are to states that faIl into two or more different groups of the current partition, then we must split A so that the transitions from the subsets of A are all confined to a single group of the current partition. Suppose, for example, that s1 and s2 go to states t1 and t2 on input a, and t1 and t2 are in different groups of the partition. Then we must split A into at least two subsets so that one subset contains s1 and the other s2 Note that t1 and t2 are distinguished by some string w, so s1 and s2 are distinguished by string aw.

We repeat this process of splitting groups in the current partition until no more groups need to be split.

Performance Analysis:

Time-Space Tradeoffs

Given a regular expression r and an input string x, we now have two methods for determining whether x is in L(r). One approach is to construct an NFA N from regular expression r. This construction can be done in O(| r |) time, where | r | is the length of r. N has at most twice as many states as | r |, and at most two transitions from each state, so a transition table for N can be stored in O(| r |) space. We can then use NFA simulation Algorithm to determine whether N accepts x in O(| r | X | x |) time. Thus, using this approach, we can determine whether x is in L (r) in total time proportional to the length of r times the length of x. This approach has been used in a number of text editors to search for regular expression patterns when the target string x is generally not very long.

A second approach is to construct a DFA from the regular expression r by applying Thompson's construction to r, and then the subset construction. Implementing the transition function with a transition table, we can use DFA simulation Algorithm on input x in time proportional to the length of x, independent of the number of states in the DFA. This approach has often been used in pattern-matching programs that search text files for regular expression patterns. Once the finite automaton has been constructed, the searching can proceed very rapidly, so this approach is advantageous when the target string x is very long.

There are, however, certain regular expressions whose smallest DFA has a number of states that is exponential in the size of the regular expression. For example, the regular expression

(alb)*a(alb)(alb) . . . (alb), where there are n - I (a I b)'s at the end, has no DFA with fewer than 2n states. This regular expression denotes any string of a's and b's in which the nth character from the right end is an a. It is not hard to prove that any, DFA for this expression must keep track of the last n characters it sees on the input; otherwise, it may give an erroneous answer. Clearly, at least 2n states are required to keep track of all possible sequences of n a's and b's. Fortunately, expressions such as this do not occur frequently in lexical analysis applications, but there are applications where similar expressions do arise.

A third approach is to use a DFA, but avoid constructing all of the transition table by using a technique called "lazy transition evaluation." Here, transitions are computed at run time but a transition from a given state on a given character is not determined until, it is actually needed. The compute transitions are stored in a cache. Each time a transition is about to be made, the cache is consulted. If the transition is not there, it is computed and stored in the cache. If the cache becomes full, we can erase some previously computed transition to make room for the new transition.

Table 2 summarizes the worst-case space and time requirements for determining whether an input string x is in the language denoted by a regular expression r using recognizers constructed from nondeterministic and deterministic finite automata. The "lazy" technique combines the space requirement of the NFA method with the time requirement of the DFA approach. Its space requirement is the size of the regular expression plus the size of the cache; its observed running time is almost as fast as that of a deterministic recognizer. In some applications, the "lazy" technique is considerably faster than the DFA approach, because no time is wasted computing state transitions that are never used.

AUTOMATAN	SPACE	TIME
NFA	O(|r|)	O(|r| * |x|)
DFA	O(2|r|)	O(|x|)

Table 2: Space and time taken to recognize regular expressions

Conclusion:

In this work, we present concepts that have been used to implement and optimize pattern matchers constructed from regular expressions. The first algorithm is suitable for inclusion in a

Lex compiler because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA along the way.

The second algorithm minimizes the number of states of any DFA, so it can be used to reduce the size of a DFA-based pattern matcher. The algorithm is efficient; its running time is O(n log n), where n is the number of states in the DFA.

References:

1. Aho, A. V., and J. D. Ullman: Principles of Compiler Design, Reading, Mass.: Addison-Wesley, 1977.
2. Aho, A. V., and J. D. Ullman: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Englewood Cliffs, N.J.: Prentice-Hall, 1972.
3. Barrett, W. A., and J. D. Couch: Compiler Construction: Theory and Practice, Chicago: Science Research Associates, Inc., 1979.
4. Gries, D.: Compiler Construction for Digital Computers, New York: John Wiley & Sons. 1971.
5. Hopcroft, J. E., and J. D. Ullman: Formal Languages and Their Relation to Automata, Reading. Mass.: Addison-Wesley, 1969.
6. Hopcroft, J. E, and J. D. Ullman: Introduction to Automata Theory, Languages, and Computation, Reading, Mass.: Addison-Wesley, 1979.
7. Johnson, S. C.: "Yacc: Yet Another Compiler-Compiler," Computing Services Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.
8. Lesk, M. E, and E. Schmidt: "Lex-A Lexical Analyzer Generator," Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N. J., 1975.
9. Lewis, P. M. II, D. J. Rosenkrantz, and R. E Steams: Compiler Design Theory, Reading. Mass.: Addison-Wesley, 1976.