COMPILER CONSTRUCTION (FA and Design of Lexical Analyser) Fall 2019

Lecture 9: Aug. 01, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

9.1 Finite Automata

A finite automata is like the graph we have used for recognition of tokens. However, there are small differences. The finite automata are recognizers, as they simply say "yes" or "no" about each possible string. A finite automata can be any of two types: 1. deterministic finite automata (DFA) or 2. nondeterministic finite automata (NFA). The DFA has for each state and each symbol combination, only one transition and its corresponding edge in the graph. However, in the NFA, there is no restriction about labels on their edges. A symbol can be a label of several edges emerging out of the same state, and, the ε, the empty symbol, can also be a possible symbol. Both the the DFA and NFA are capable of recognizing the same languages. The languages recognized by the automata are called *regular languages*.

A finite automata is represented by $M = (Q, \Sigma, \delta, s, F)$, where Q is set of states, Σ is set of alphabets, δ is transition function, s is start state, $F \subseteq Q$ is set of final states, and δ is defined as,

$$\delta : Q \times \Sigma \to Q.$$

We represent the DFA NFA both by *transition graph*.

Example 9.1 *A DFA for regular expression* $(a|b)^*bbb$.

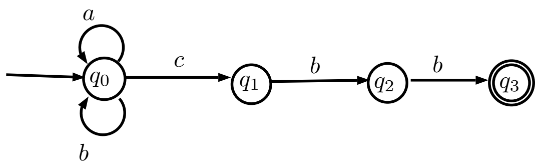

Figure 9.1: A DFA for regular expression: $(a|b)^*cbb$

The DFA is shown in Fig. 9.1. This is an abstract example, which describes all the strings ending with bbb, preceding with any number of a and b symbols in any order. The first circle q_0 with arrow entering is always taken as the start, and the state represented by double circle is called final state. The machine is represented by $M = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta, s, q_3)$. $\square$

Algorithm 1 Simulating a DFA: Input: A string terminated by **eof**. A DFA D (as transition table) with start state s_0 and accepting states F. The transition function is $new\_state = move(current\text{-}state, symbol)$, Output = "yes" if D accepts else "no"

```
1: $s = s_0$
2: $c = nextChar()$
3: while ($c <>$ eof) do
4:     $s = move(s, c)$
5:     $c = nextChar()$
6: end while
7: if ($s \in F$) then
8:     return "yes"
9: else
10:    return "no"
11: end if
```

9.2 From Regular Expressiosn to Automata

A regular expression is a notation for describing lexical analysers and other pattern-processing software. However, implementation of that software requires the simulation of a DFA or an NFA. Since an NFA has choice of making a move with ε-input, or making more that one move with same input symbol, its simulation is less forward than a DFA. Thus, there is a need of converting an NFA into a DFA that accepts the same language.

While converting a NFA to DFA, we group the states that are different in NFA but are the destination due to same transition symbol from the previous state. Also, we merge the states that are due to null (ε) transitions, through the algorithm is called subset construction algorithm.

More more details about conversion from NFA to DFA, refer to theory of computation lectures.

Simulation of an NFA The text editing programs make use of concept of constructing an NFA from a regular expression and then simulate the NFA using on-the-fly subset construction (see Algorithm 2). Note that, uppercase S symbol is used for state set. This because every transition in NFA is to a set of states. Even the start state may be a set, e.g., if there is a ε-transition from s_0 to s_1 the ε-closure of s_0 is $\{s_0, s_1\} = S_0$. Similarly, we consider the ε-closure for every state, which is merging of states to which there is transition due to ε, along with the previous state.

9.3 Design of a Lexical-analyser

In the following we will discuss about how a lexical analyser generator like Lex can be constructed. For this we need to discuss about the structure of a lexical analyser that could be generated using Lex tool. Such lexical analyser is originally in the form of a lex program, i.e., it consists of *patterns* and the *actions.* This program simulates a automaton – usually a deterministic automaton. The Fig. 9.2 demonstrates this process, i.e., generation of lexical analyser, and then using it as recognizer of tokens.

Algorithm 2 Simulating an NFA: Input: A string x terminated by **eof**. A NFA N with start state s_0 and accepting states F. The transition function is $new\ state = move(state, symbol)$, Output = "yes" if N accepts else "no"

```
1: $S = \varepsilon\text{-}(s_0)$
2: $c = nextChar()$
3: while $(c! = \textbf{eof})$ do
4:     $S = \varepsilon\text{-}closure(move(S, c))$
5:     $c = nextChar()$
6: end while
7: if $(S \cap F\ ! = \phi)$ then
8:     return "yes"
9: else
10:    return "no"
11: end if
```

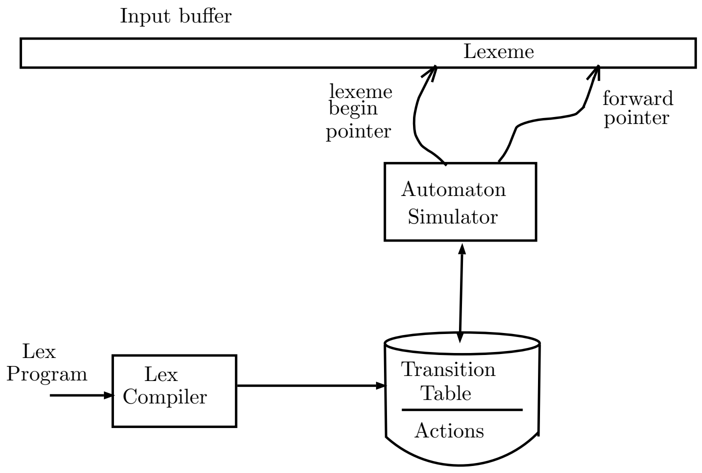

Figure 9.2: Creation of Lexical analyser

The figure comprises of components that are created from the Lex program by Lex itself. These are as follows:

1. A transition table for the finite automata,
2. The functions that are passed by the Lex to the output,
3. Actions from input program, which are fragments of code to be invoked by the FA simulator, when required.

The construction of FA we have presented in the form of algorithm 2.

In fact, given the regular expression, an NFA is constructed, which is converted into a DFA, and then DFA is minimized for number of states, so that over all code is efficient. However, all these we will not be repeating here, as their theoretical part is usually studied by the students. The curious students may study these in some Theory of Computation text.

Example 9.2 *Construction of NFA for given patterns.*

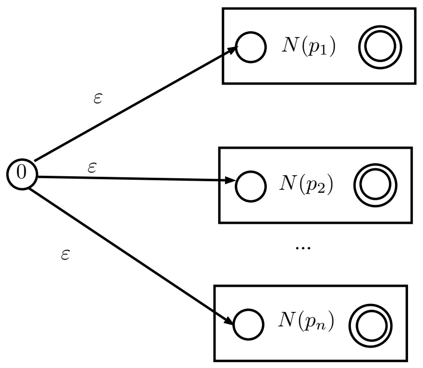

Figure 9.3: A NFA constructed from a Lex Program

We have the below given patterns and that aligns with the action

a	{action A_1 for patternp_1}
abb	{action A_2 for patternp_2}
$\mathbf{a}^*\mathbf{b}^+$	{action A_3 for patternp_3}

The actions for these patterns are construction of NFA, and that are combined to forms a single NFA. This NFA is converted into a DFA using standard algorithm making use of subset construction approach.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.