CS 420 - Compilers

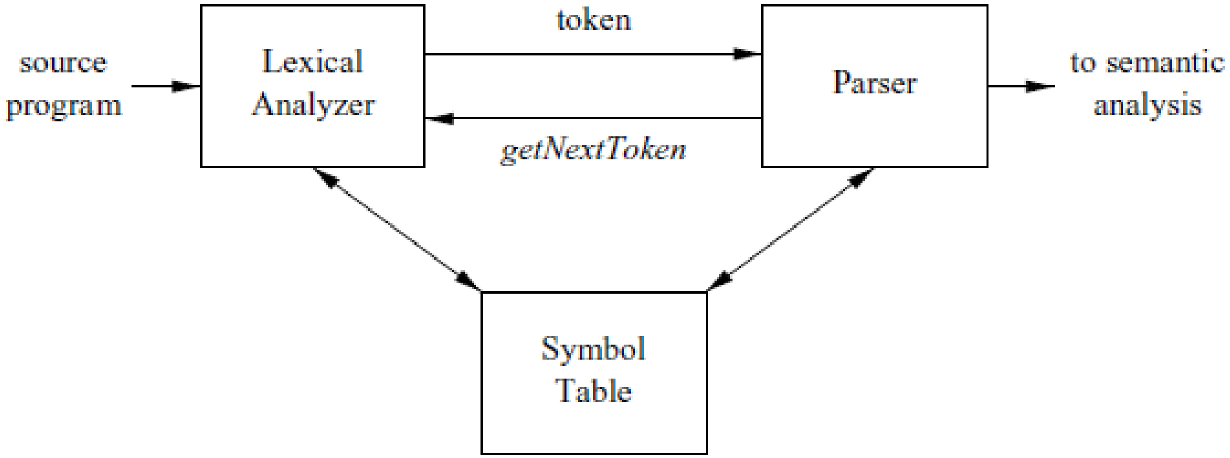
Dr. Chen-Yeou (Charles) Yu

- **From Regular Expressions to Automata (3.7)**
  - **Conversion of an NFA to a DFA (3.7.1)**
  - **Simulation of an NFA (3.7.2)**
  - **Efficiency of NFA Simulation (3.7.3) (Bypass)**
  - **Construction of an NFA from a Regular Expression (3.7.4)**
  - **…**
- **Design of a Lexical-Analyzer Generator (3.8) (TBD. In Part8)**

From Regular Expressions to Automata

- Do not forget the goal of this chapter is to understand the lexical analysis
- We had a couple of jobs to do
  - Convert the Regular Expression (RE) to NFA
  - **Convert the NFA to DFA**
- The book, introduced the “jobs to do” in,

reversed order



Conversion of an NFA to a DFA

- The general idea behind the subset construction is that each state of the constructed DFA **corresponds** to a **set** of **NFA states**.
- DFA states would be the subset of NFA states!

Conversion of an NFA to a DFA

- Algorithm: subset construction of DFA from NFA
- INPUT: An NFA, N.
- OUTPUT: A DFA, D accepting the same language as N.
- This is the example of the input NFA, N

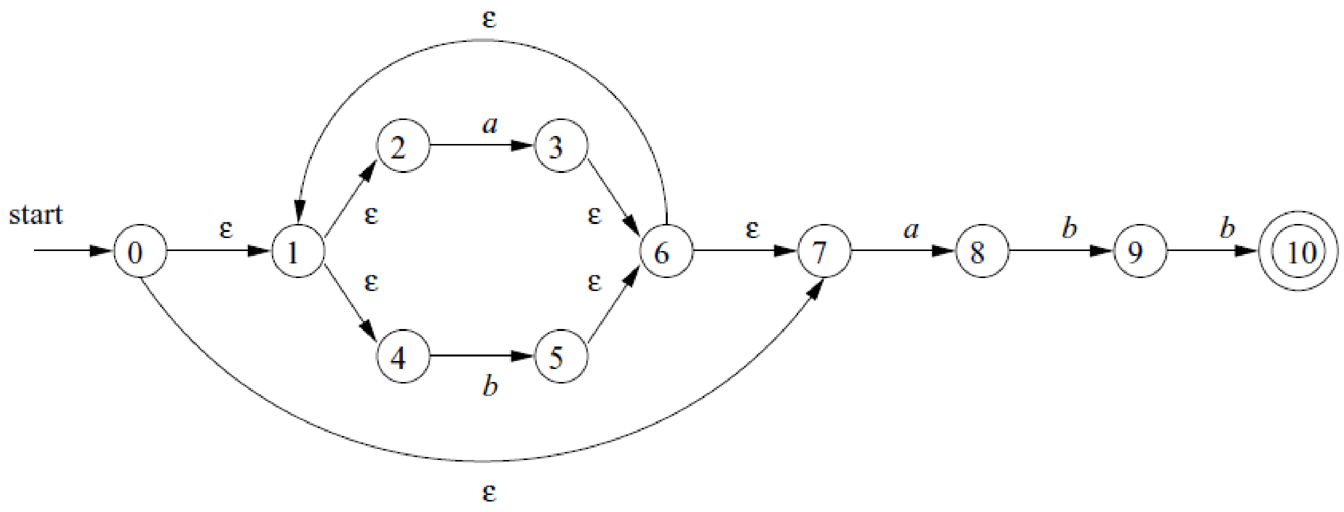


Figure 3.34: NFA N for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Conversion of an NFA to a DFA

- This is the tool (Transition table, Dtran) we are going to use in this example, the Dtran for DFA, D

NFA STATE	DFA STATE	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 4, 5, 6, 7, 10\}$	E	B	C

Figure 3.35: Transition table *Dtran* for DFA D

Conversion of an NFA to a DFA

- This is the output example of output DFA, D
  - See? The epsilons are removed!

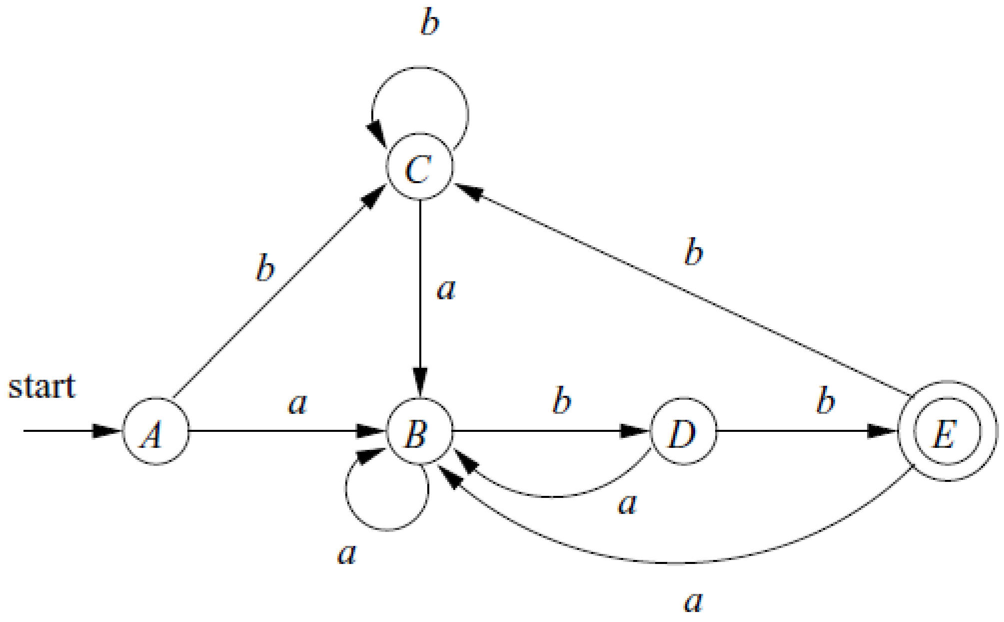


Figure 3.36: Result of applying the subset construction to Fig. 3.34

Conversion of an NFA to a DFA

- As for the detail, it involves complicated set operations.
- I will just try to by pass that because for its complexity
- You can check the book
- They do the subset construction in making use of the “epsilon-closure”.
- Instead of the conversion from NFA to DFA, we can directly run the NFA simulation itself.
- In the area of the computation, sometimes, the algorithm is very hard to prove. We can use the simulation to support our idea

Simulation of an NFA

- **INPUT**: An input string x terminated by an end-of-file character eof. An NFA *N* with start state S_0, accepting states F, and transition function move().
- **OUTPUT**: Answer “yes” if N accepts x; “no” otherwise.
- **METHOD**: The algorithm keeps a set of current states S, those that are reached from S_0 following a path labeled by the inputs read so far
  - If *c* is the next input character, read by the function nextChar(), then we first compute move(S, c) and then close that set using $\epsilon\text{-}closure()$

Simulation of an NFA

```
S = ε-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ε-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

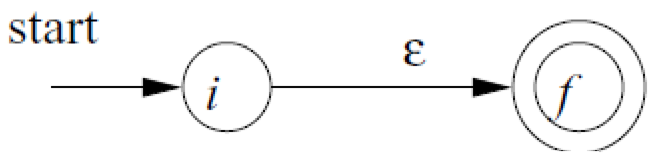
Figure 3.37: Simulating an NFA

Construction of an NFA from a Regular Expression

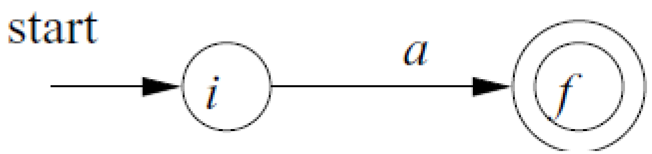
- We will just quickly go through that(not focusing onto detail)
- In the book, it is an algorithm for converting any regular expression to an NFA that defines the **same** language.
- INPUT: A regular expression *r* over alphabet Σ
- OUTPUT: An NFA *N* accepting L(*r*).
- METHOD: Begin by parsing *r* into its constituent subexpressions.
  - The rules for constructing an NFA consist of: 1) basis rules for handling subexpressions with no operators, and 2) inductive rules for constructing larger NFA's from NFA’s immediate subexpressions of a given expression

Construction of an NFA from a Regular Expression

BASIS: For expression ϵ construct the NFA



For any subexpression a in Σ, construct the NFA



Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

Construction of an NFA from a Regular Expression

- Now, we deal with the “induction” steps:

INDUCTION: Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t, respectively.

a) Suppose $r = s|t$. Then $N(r)$, the NFA for r, is constructed as in Fig. 3.40.

- This is the claim:

Since any path from i to f must pass through

either $N(s)$ or $N(t)$ exclusively, and since the label of that path is not changed by the ϵ's leaving i or entering f, we conclude that $N(r)$ accepts $L(s) \cup L(t)$, which is the same as $L(r)$. That is, Fig. 3.40 is a correct construction for the union operator.

Construction of an NFA from a Regular Expression

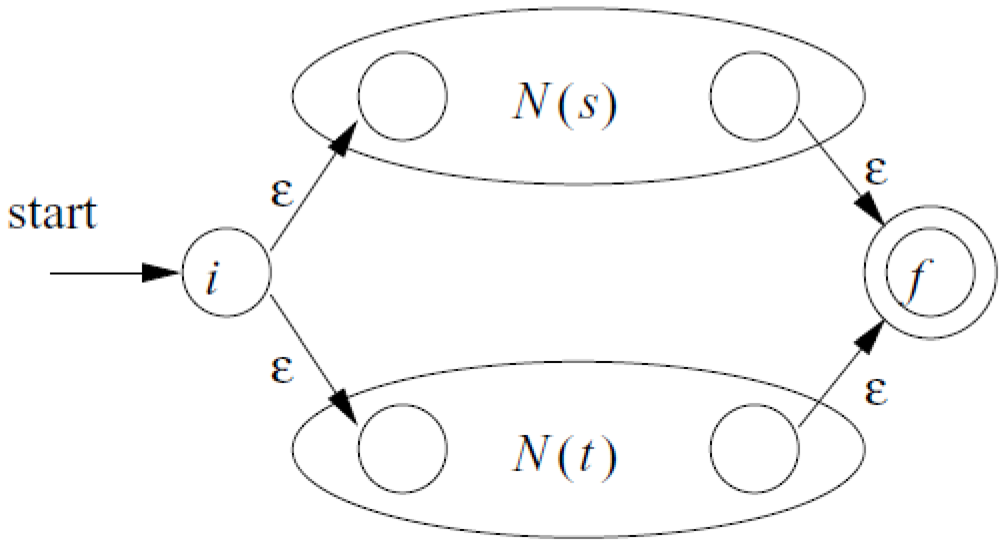


Figure 3.40: NFA for the union of two regular expressions

Construction of an NFA from a Regular Expression

b) Suppose $r = st$. Then construct $N(r)$ as in Fig. 3.41. The start state of $N(s)$ becomes the start state of $N(r)$, and the accepting state of $N(t)$ is the only accepting state of $N(r)$. The accepting state of $N(s)$ and the start state of $N(t)$ are merged into a single state, with all the transitions in or out of either state. A path from i to f in Fig. 3.41 must go first through $N(s)$, and therefore its label will begin with some string in $L(s)$. The path then continues through $N(t)$, so the path's label finishes with a string in $L(t)$.

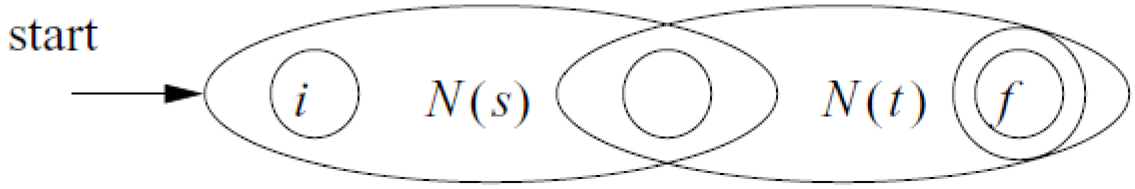


Figure 3.41: NFA for the concatenation of two regular expressions

Construction of an NFA from a Regular Expression

- Claim:

accepting states never have edges

out and start states never have edges in, so it is not possible for a path to re-enter $N(s)$ after leaving it. Thus, $N(r)$ accepts exactly $L(s)L(t)$, and is a correct NFA for $r = st$.

- Then, we deal with the * closure. Check the book if you are interested.

c) Suppose $r = s^*$. Then for r we construct the NFA $N(r)$ shown in Fig. 3.42.

Construction of an NFA from a Regular Expression

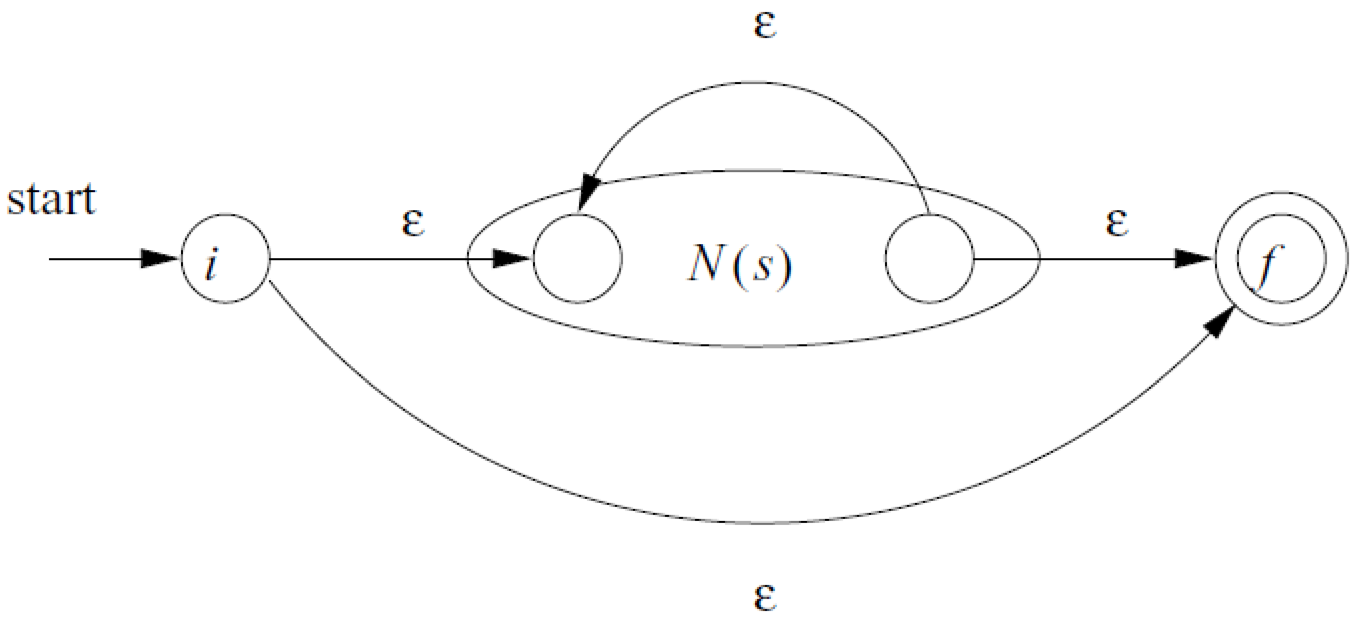


Figure 3.42: NFA for the closure of a regular expression

d) Finally, suppose $r = (s)$. Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.