Chapter 3 Scanning – Theory and Practice

Overview

- Formal notations for specifying the precise structure of tokens are necessary
  - Quoted string in Pascal
  - Can a string split across a line?
  - Is a null string allowed?
  - Is .1 or 10. ok?
  - the 1..10 problem
- Scanner generators
  - tables
  - Programs
- What formal notations to use?

Regular Expressions

- Tokens are built from symbols of a finite vocabulary.
- We use regular expressions to define structures of tokens.

Regular Expressions

- The sets of strings defined by regular expressions are termed ***regular sets***
- Definition of regular expressions
  - $\varnothing$ is a regular expression denoting the empty set
  - λ is a regular expression denoting the set that contains only the empty string
  - A string **s** is a regular expression denoting a set containing only **s**
  - if A and B are regular expressions, so are
    - A | B (alternation)
    - AB (concatenation)
    - A* (Kleene closure)

Regular Expressions (Cont'd)

some notational convenience

P+ == PP*

Not(A) == V - A

Not(S) == V* - S

A^K == AA …A (k copies)

Regular Expressions (Cont'd)

- Some examples

Let D = (0 | 1 | 2 | 3 | 4 | ... | 9)

L = (A | B | ... | Z)

comment = -- not(EOL)* EOL

decimal = D+ · D+

ident = L (L | D)* (_ (L | D)$^+$)*

comments = ##((#| λ)not(#))* ##

Regular Expressions (Cont'd)

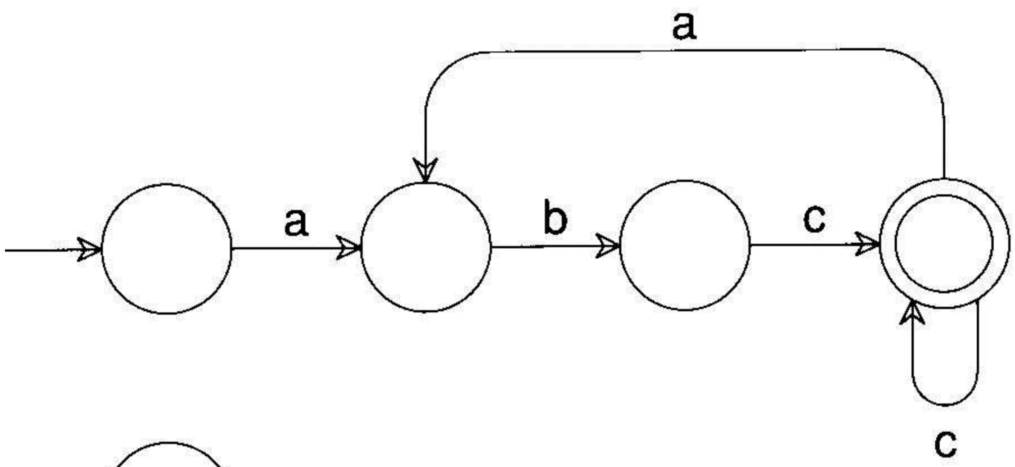
- Is regular expression as power as CFG?

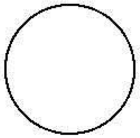
$\{ [^i]^i \mid i \geq 1 \}$

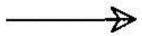
Finite Automata and Scanners

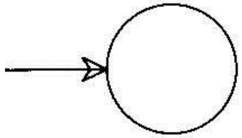
- A ***finite automaton (FA)*** can be used to recognize the tokens specified by a ***regular expression***
- A FA consists of
  - A finite set of states
  - A set of transitions (or moves) from one state to another, labeled with characters in **V**
  - A special *start* state
  - A set of *final*, or *accepting*, states

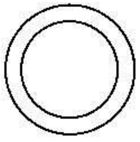
A transition diagram



 is a state

 is a transition

 is the start state

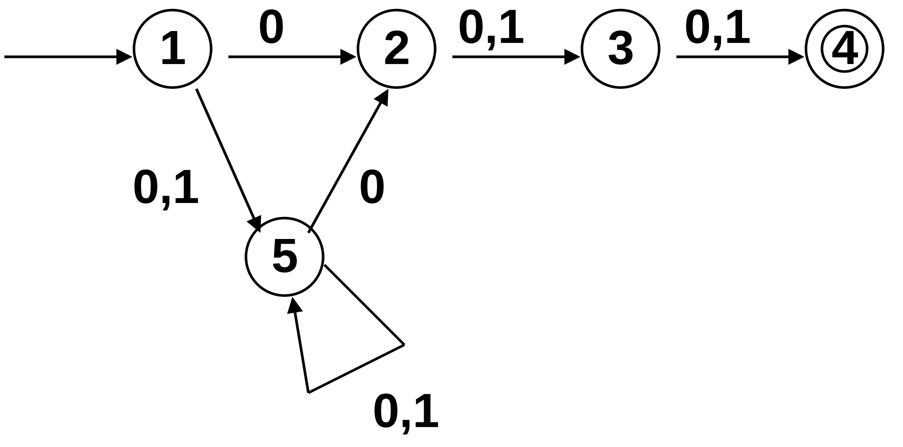
 is a final state

This machine accepts abccabc, but it rejects abcab.

This machine accepts $(abc^+)^+$.

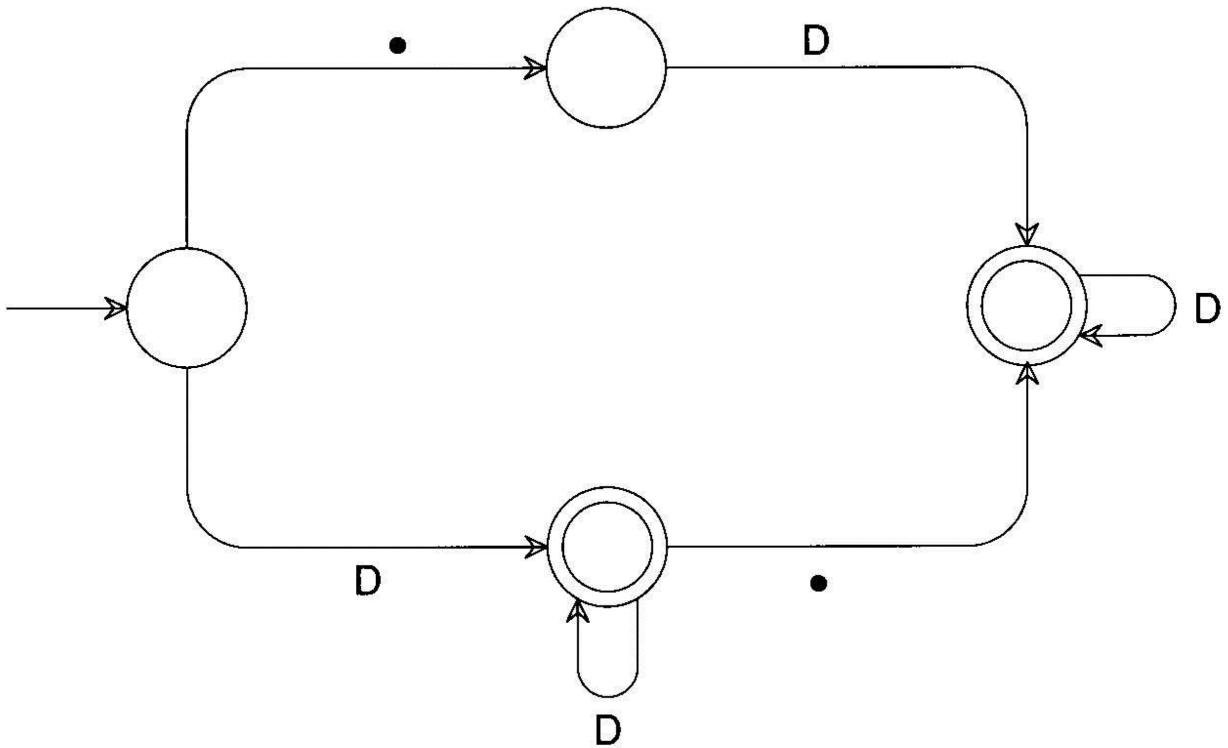
- **Example**

(0|1)*0(0|1)(0|1)



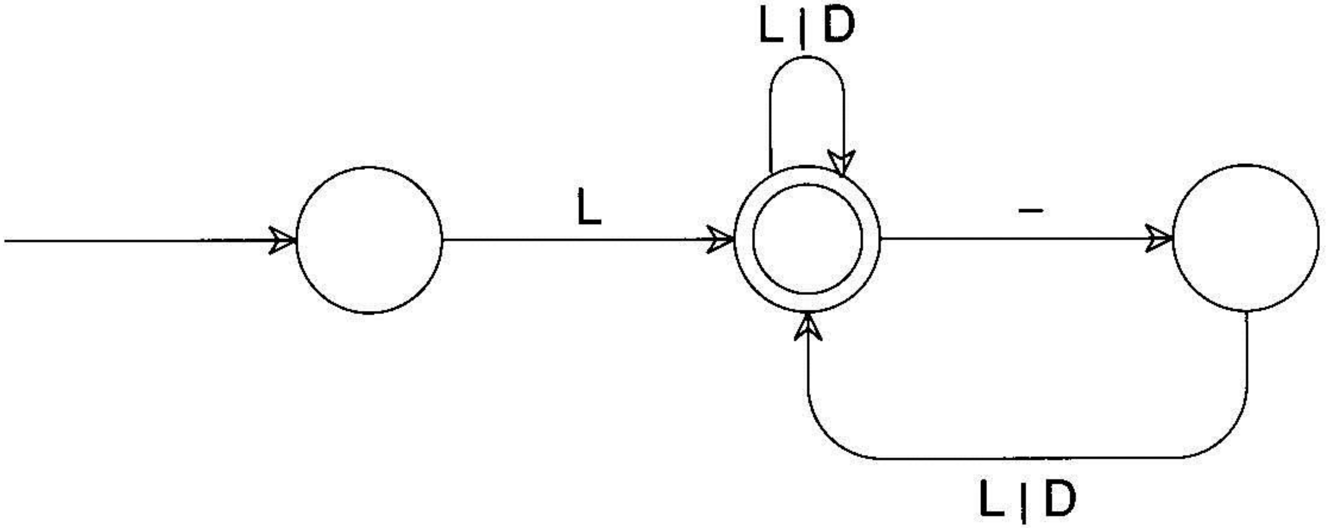
- **Example**

$$\textbf{RealLit} = (D^+(\lambda|.))|(D^*.D^+)$$



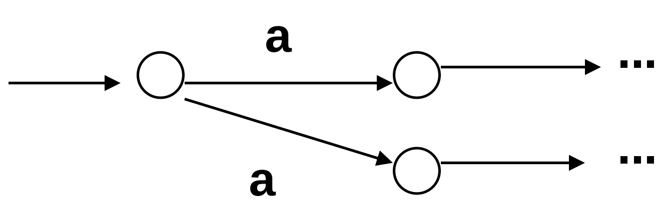
- **Example**

$$\mathbf{ID = L(L|D)^*(\_(L|D)^+)^*}$$

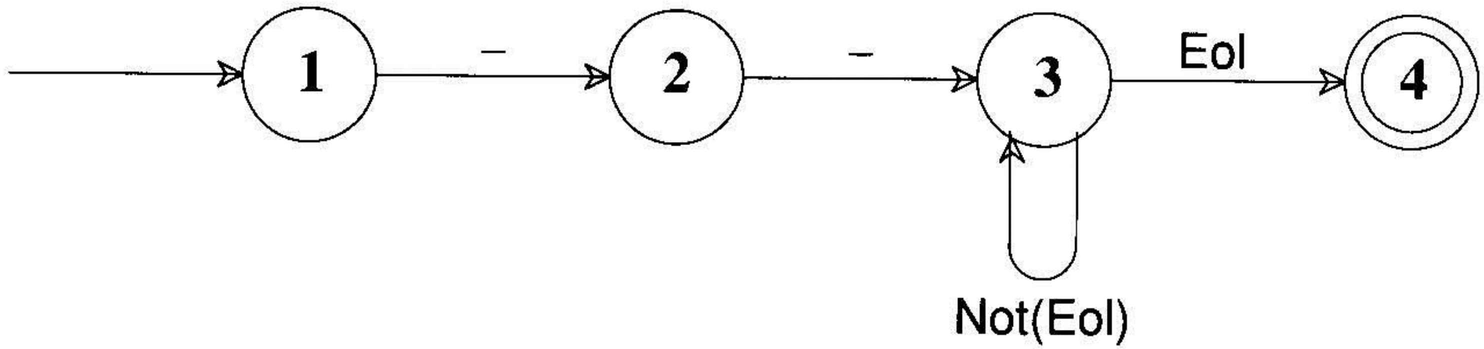


Finite Automata and Scanners

- Two kinds of FA:
  - Deterministic: next transition is unique
  - Non-deterministic: otherwise



A transition table of a DFA



The corresponding transition table is

State	Character				
	–	Eol	a	b	. . .
1	2				
2	3				
3	3	4	3	3	3
4					

Finite Automata and Scanners

- Any regular expression can be translated into a DFA that accepts the set of strings denoted by the regular expression
- The transition can be done
  - Automatically by a scanner generator
  - Manually by a programmer

```
/*
 * Note: current_char is already set to
 * the current input character.
 */
state = initial_state;
while (TRUE) {
    next_state = T[state][current_char];
    if (nextstate == ERROR)
        break;
    state = next_state;
    if (current_char == EOF)
        break;
    current_char = getchar();
}
if (is_final_state(state))
    /* Return or process valid token. */
else
    lexical_error(current_char);
```

Figure 3.1 Scanner Driver Interpreting a Transition Table

```
if (current_char == '-') {
    current_char = getchar();
    if (current_char == '-') {
        do
            current_char = getchar();
        while (current_char != '\n');
    } else {
        ungetc(current_char, stdin);
        lexical_error(current_char);
    }
}
else
    lexical_error(current_char);
/* Return or process valid token. */
```

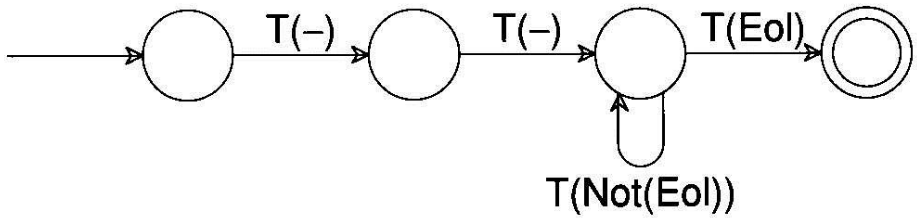
Figure 3.2 Scanner with Fixed Token Definition

Finite Automata and Scanners

- Transducer
  - **We may perform some actions during state transition.**

$\xrightarrow{\quad a \quad}$ means save a in a token buffer

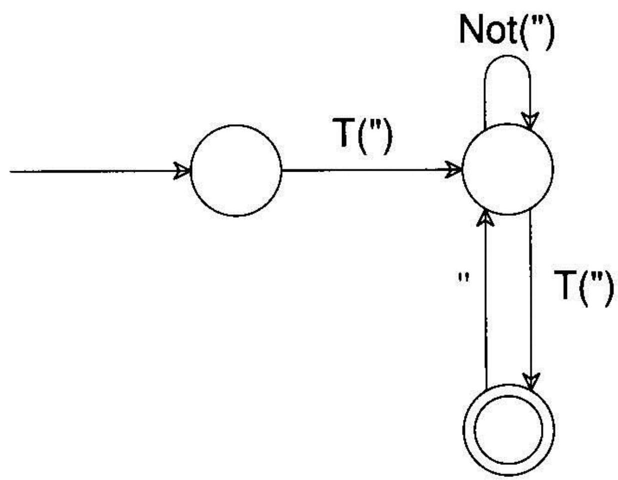
$\xrightarrow{\quad T(a) \quad}$ means don't save a (Toss it away)



A more interesting example is given by quoted strings, according to the regular expression

(" (Not(") | " ")* ")

A corresponding transducer might be



The input """Hi""" would produce output "Hi".

Practical Consideration

- Reserved Words
  - Usually, all keywords are reserved in order to simplify parsing.
  - In Pascal, we could even write

```
begin
    begin; end; end; begin;
 end
 if else then if = else;
```

- The problem with reserved words is that they are too numerous.
  - COBOL has several hundrens of reserved words!

Practical Consideration (Cont'd)

- Compiler Directives and Listing Source Lines
  - Compiler options e.g. optimization, profiling, etc.
    - handled by scanner or semantic routines
    - Complex pragmas are treated like other statements.
  - Source inclusion
    - e.g. #include in C
    - handled by preprocessor or scanner
  - Conditional compilation
    - e.g. #if, #endif in C
    - useful for creating program versions

Practical Consideration (Cont'd)

- Entry of Identifiers into the Symbol Table
- Who is responsible for entering symbols into symbol table?
  - Scanner?
  - Consider this example:

```
{ int abc;
        ...
      { int abc; }
}
```

Practical Consideration (Cont'd)

- How to handle end-of-file?
  - Create a special EOF token.
    - EOF token is useful in a CFG
- Multicharacter Lookahead
  - Blanks are not significant in Fortran
    - DO 10 I = 1,100
      - Beginning of a loop
    - DO 10 I = 1.100
      - An assignment statement **DO10I=1.100**

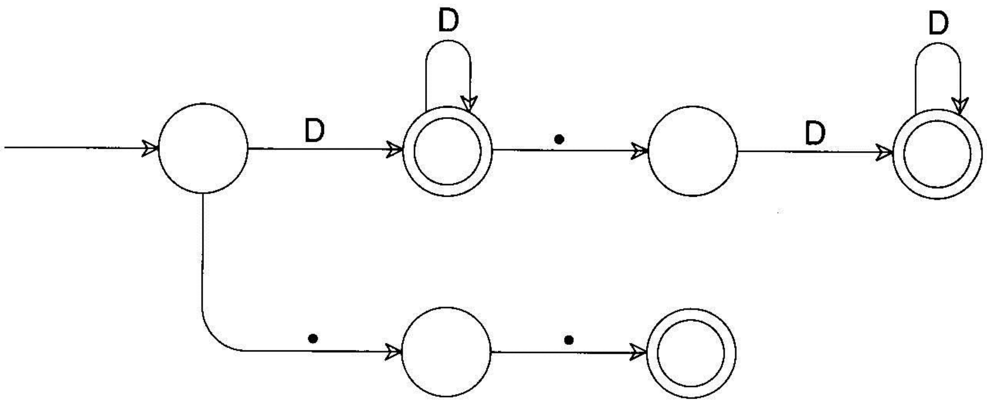
    - A Fortran scanner can determine whether the O is the last character of a DO token only after reading as far as the comma

Practical Consideration (Cont'd)

- Multicharacter Lookahead (Cont'd)
  - In Ada and Pascal
    - To scan 1..100
      - There are three token
        - » 1
        - » ..
        - » 100
      - Two-character lookahead after the 10

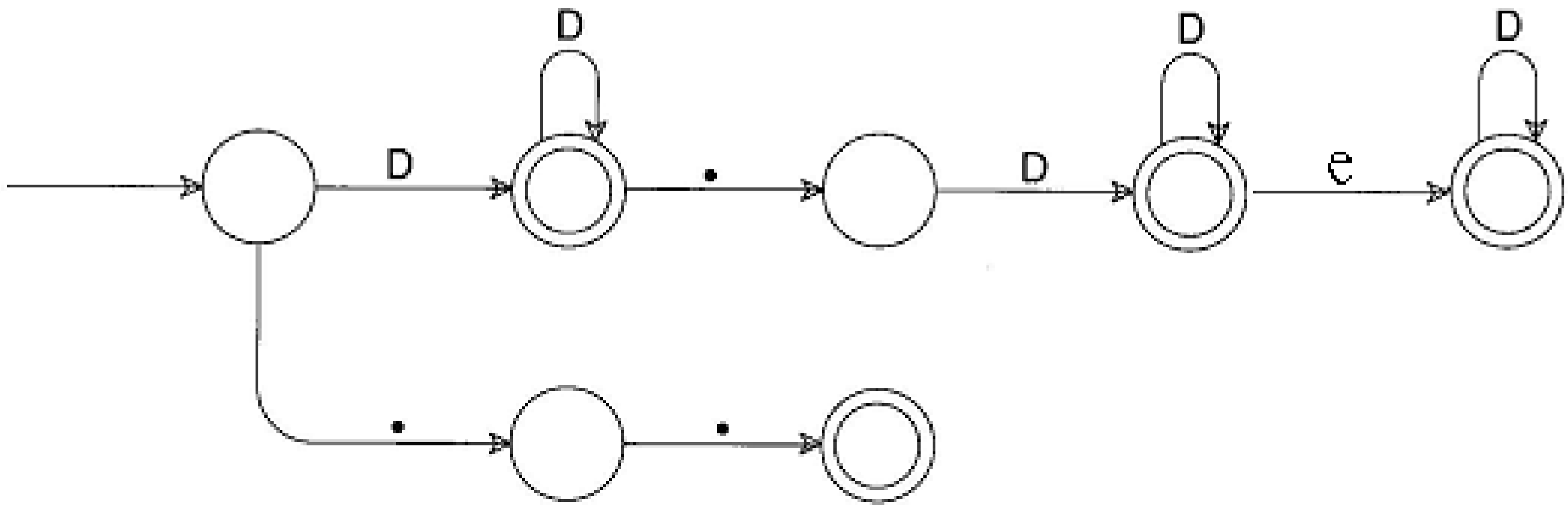
Practical Consideration (Cont'd)

- Multicharacter Lookahead (Cont'd)
  - It is easy to build a scanner that can perform general backup.
  - If we reach a situation in which we are not in final state and *cannot scan any more characters*, backup is invoked.
    - Until we reach a prefix of the scanned characters flagged as a valid token



Buffered Token	Token Flag
1	Integer Literal
12	Integer Literal
12.	Invalid
12.3	Real Literal
12.3e	Invalid
12.3e+	Invalid

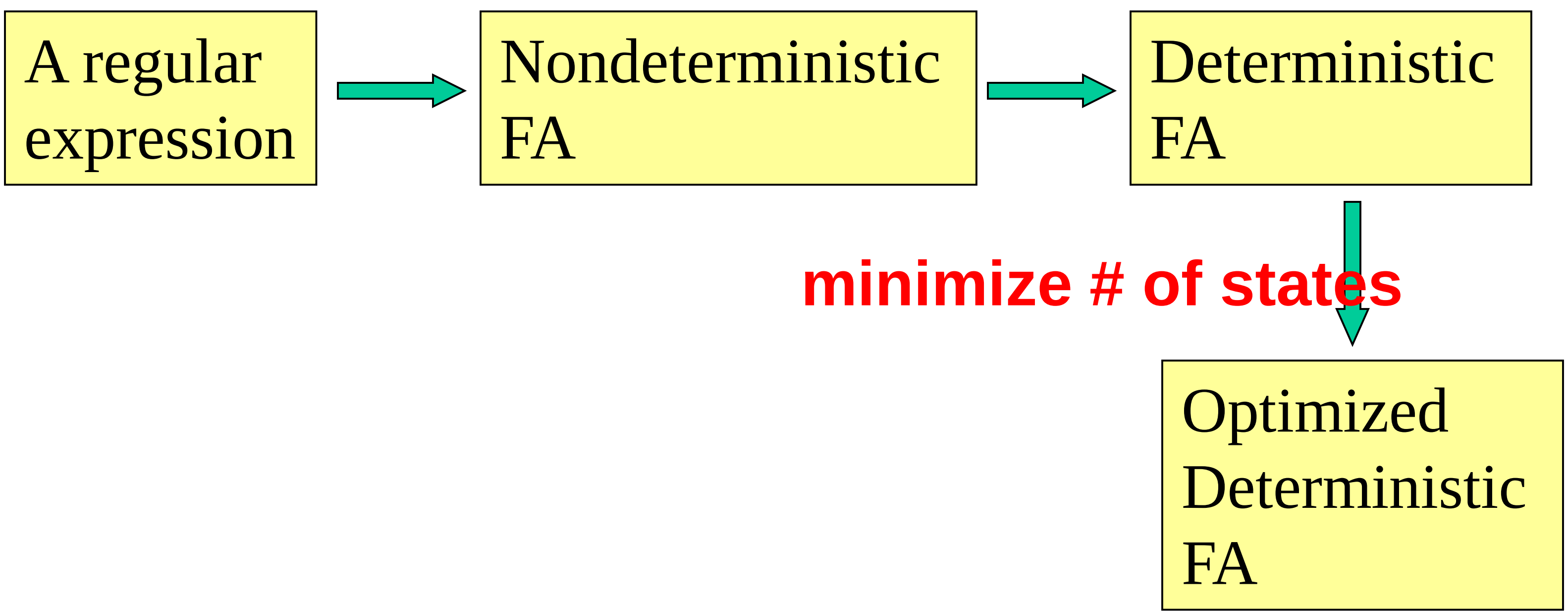
Figure 3.6 An FA That Scans Integer and Real Literals and the Subrange Operator



Buffered Token	Token Flag
1	Integer Literal
12	Integer Literal
12.	Invalid
12.3	Real Literal
12.3e	Invalid
12.3e+	Invalid

Translating Regular Expressions into Finite Automata

- Regular expressions are equivalent to FAs
- The main job of a scanner generator
  - To transform a regular expression definition into an equivalent FA



Translating Regular Expressions into Finite Automata

- A FA is nondeterministic:

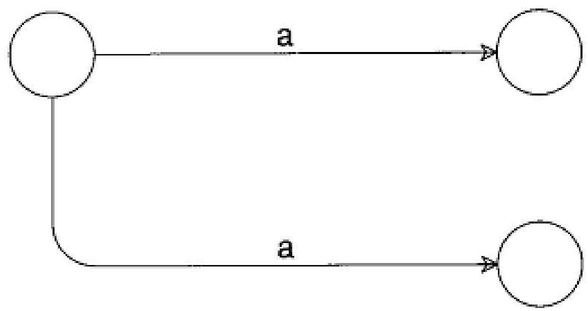


Figure 3.7 An NFA with Two a Transitions

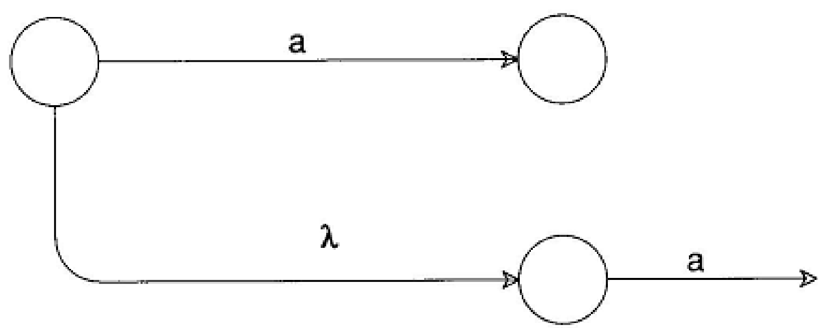


Figure 3.8 An NFA with a λ Transition

Translating Regular Expressions into Finite Automata

- We can transform any regular expression into an NFA with the following properties:
  - There is an unique final state
  - The final state has no successors
  - Every other state has either one or two successors

Translating Regular Expressions into Finite Automata

- We need to review the definition of regular expression

1. **λ (null string)**
2. **a (a char of the vocabulary)**
3. **A|B (or)**
4. **AB (cancatenation)**
5. **A* (repetition)**

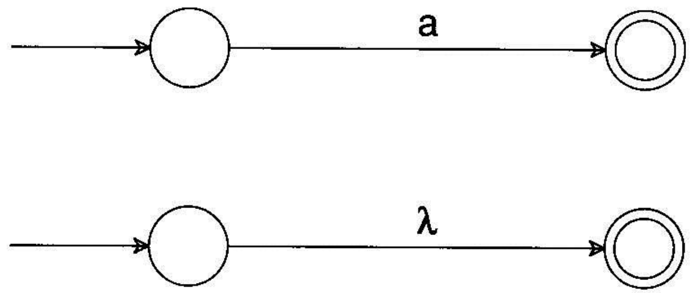


Figure 3.9 NFAs for a and λ

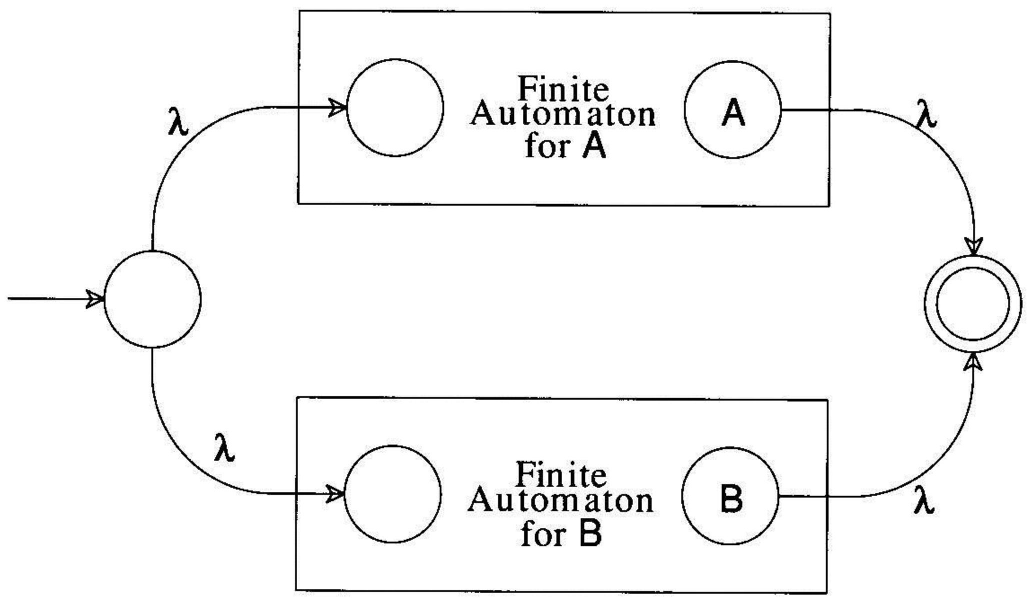


Figure 3.10 An NFA for A | B

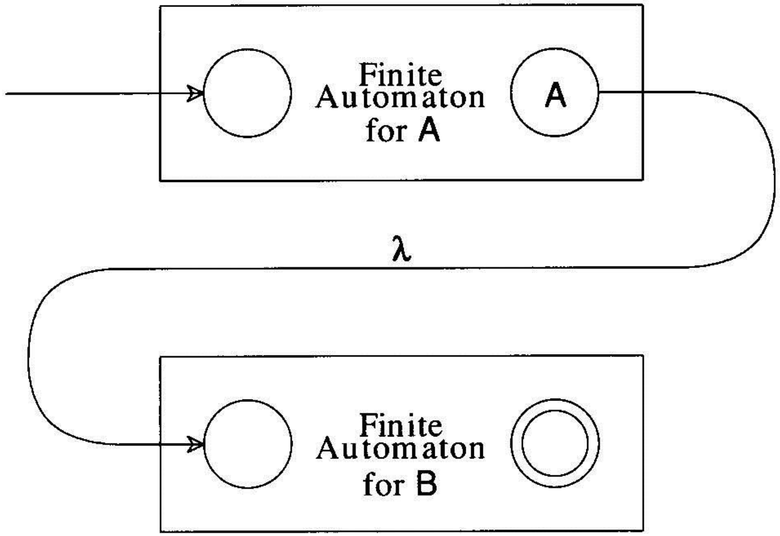


Figure 3.11 An NFA for A B

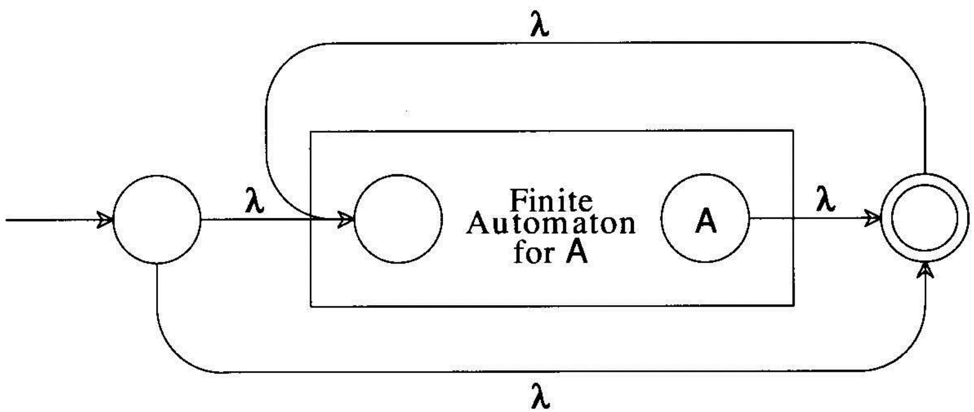
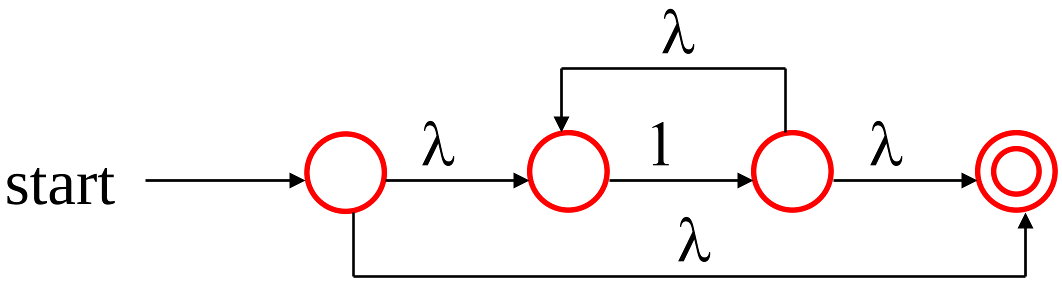


Figure 3.12 An NFA for A^*

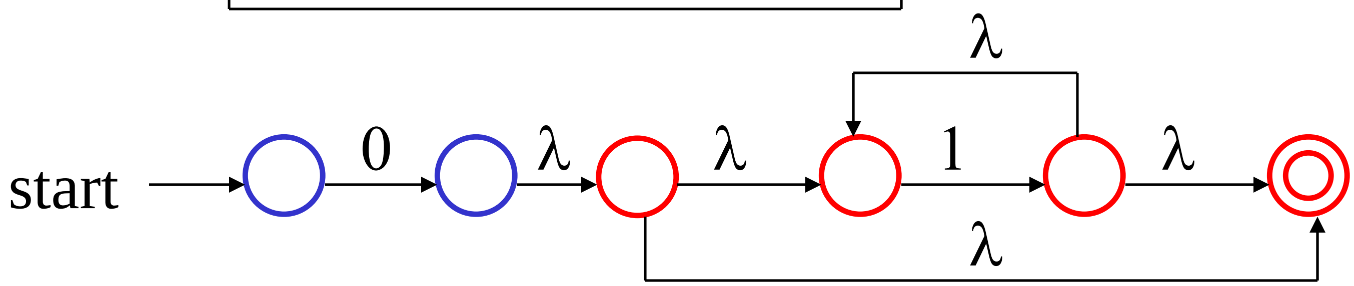
Construct an NFA for Regular Expression 01^*+1

$$\boxed{01^*+1 \Rightarrow (0(1^*))+1}$$

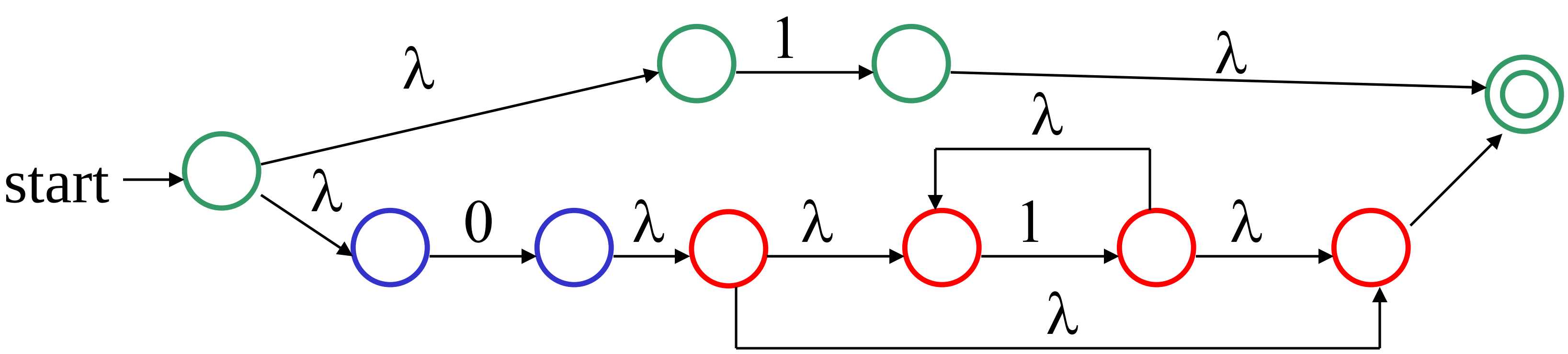
$1^* \Rightarrow$



$01^* \Rightarrow$



$01^*+1 \Rightarrow$



Creating Deterministic Automata

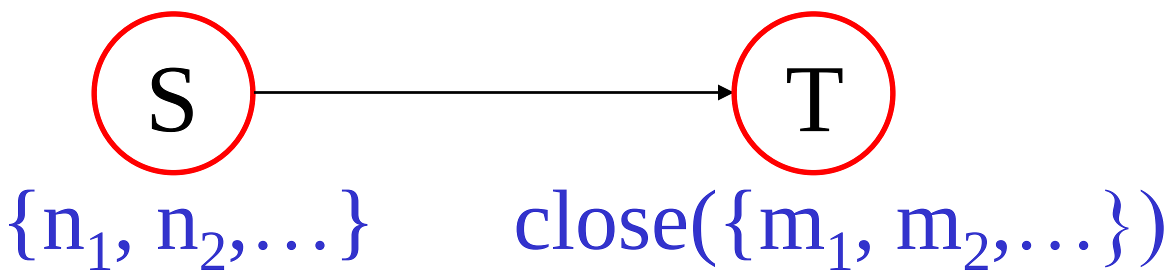
- The transformation from an NFA N to an equivalent DFA M works by what is sometimes called the ***subset construction***
  - ***Step 1:*** ***The initial state of M is the set of states reachable from the initial state of N by λ-transitions***

```
/*
 * Add to S all states reachable from it
 * using only λ transitions of N
 */
void close(set_of_fa_states *S)
{
    while (there is a state x in S
        and a state y not in S such that
        x→y using a λ transition)
            add y to S
}
```

Creating Deterministic Automata

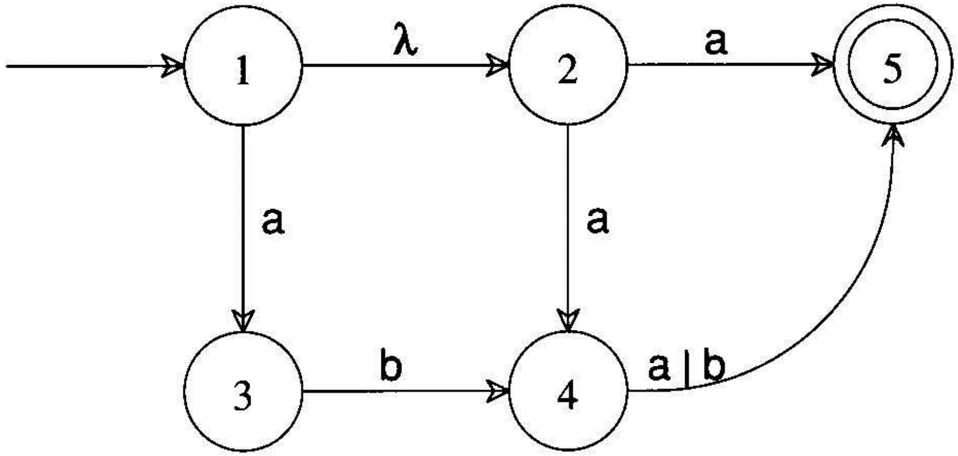
– ***Step 2:*** To create the successor states

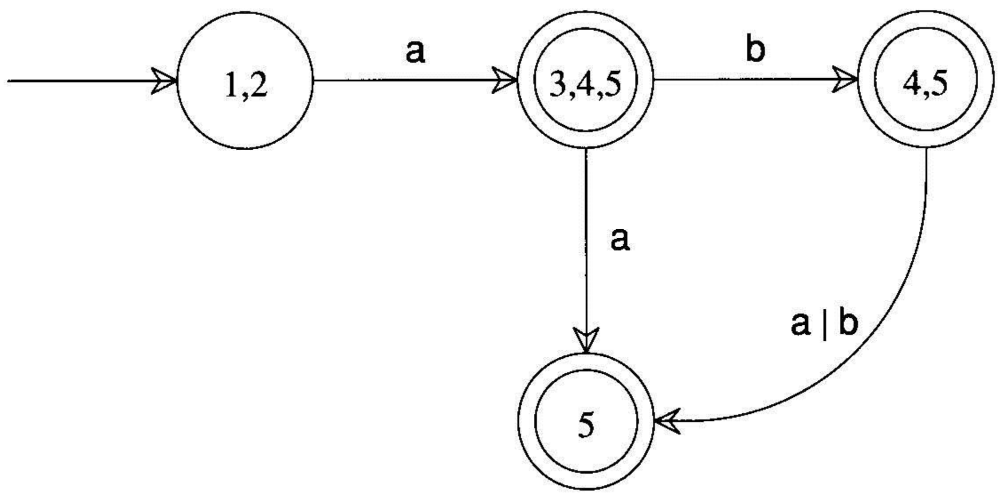
- Take any state S of M and any character c, and compute S's successor under c
  - – S is identified with some set of N's states, $\{n_1, n_2, \ldots\}$
  - – Find all possible successor states to $\{n_1, n_2, \ldots\}$ under c
    - » Obtain a set $\{m_1, m_2, \ldots\}$
  - – $T = \text{close}(\{m_1, m_2, \ldots\})$



Creating Deterministic Automata

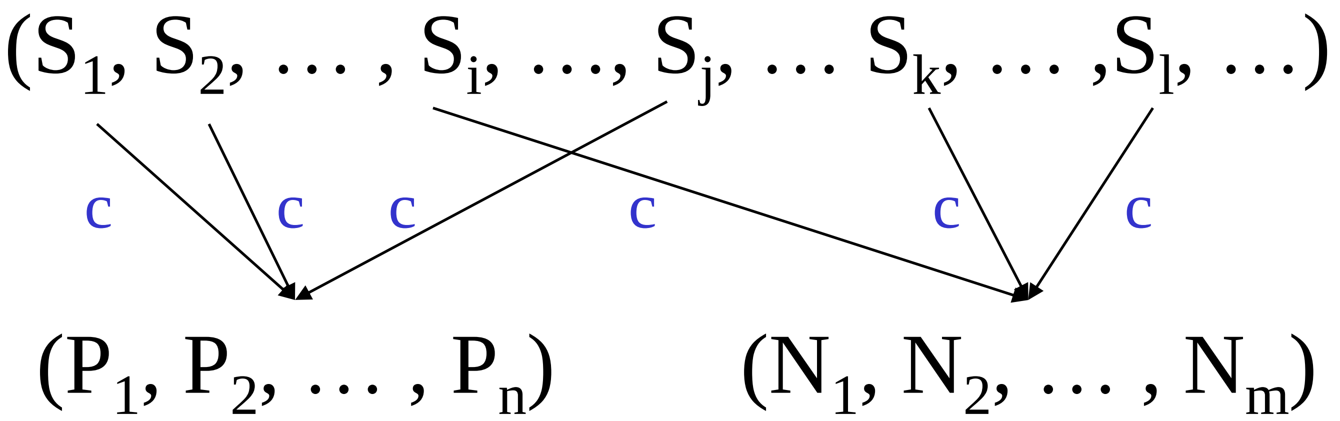
```
void make_deterministic(nondeterministic_fa N,
                        deterministic_fa *M)
{
    set_of_fa_states T;

    M->initial_state = SET_OF(N.initial_state) ;
    close(& M->initial_state);
    Add M->initial_state to M->states;
    while (states or transitions can be added)
    {
        choose S in M->states and c in Alphabet;

        T = SET_OF(y in N.states
            SUCH_THAT x -c-> y for some x in S) ;
        close(& T);
        if (T not in M->states)
            add T to M->states;

        Add the transition to M->transitions: S -c-> T ;
    }
    M->final_states =
        SET_OF(S in M->states SUCH_THAT
            N.final_state in S);
}
```


λ
a
1
2
5
a
a
3
4
b
a | b


1,2
a
3,4,5
b
4,5
a
a | b
5

Optimizing Finite Automata

- minimize number of states
  - Every DFA has a unique smallest equivalent DFA
  - Given a DFA M, we use splitting to construct the equivalent minimal DFA.
  - Initially, there are two sets, one consisting all accepting states of M, the other the remaining states.



$(S_1, S_2, \ldots, S_i, \ldots, S_j, \ldots S_k, \ldots, S_l, \ldots, S_x, S_y)$

c c c c c c

$(P_1, P_2, \ldots, P_n)$ $(N_1, N_2, \ldots, N_m)$

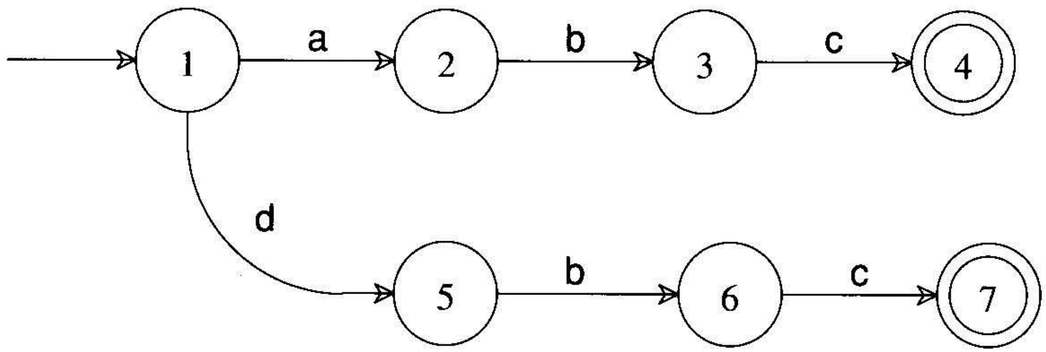
$\Downarrow$

$(S_1, S_2, S_j, \ldots)$ $(S_i, S_k, S_l, \ldots)$ $(S_x, S_y, \ldots)$

Note that S_x and S_y no transaction on c

```
void split(set_of_fa_states *ss)
{
    do {
        Let S be any merged state corresponding to
            {s₁ ,..., sₙ} and
            let c be any character;
        Let t₁ ,..., tₙ be the successor states to
            {s₁ ,..., sₙ} under c;
        if (t₁ ,..., tₙ do not all belong to the
            same merged state)
        {
            Split S into new states so that sᵢ and
            sⱼ remain in the same merged state if
            and only if tᵢ and tⱼ are in
            the same merged state;
        }
    } while (more splits are possible);
}
```

Figure 3.13 An Algorithm to Split FA States



- Initially, two sets {1, 2, 3, 5, 6}, {4, 7}.
- {1, 2, 3, 5, 6} splits {1, 2, 5}, {3, 6} on c.
- {1, 2, 5} splits {1}, {2, 5} on b.