CS 4203 Compiler Theory

Lecture #3

Scanning (Lexical Analysis)-Part2

Outline

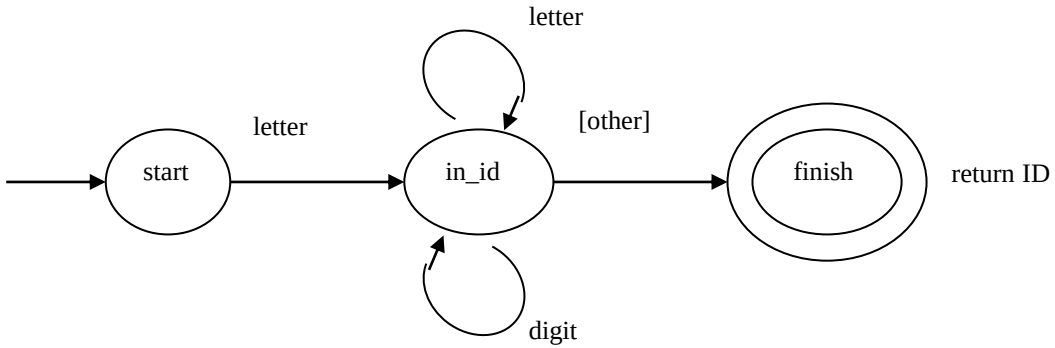
1. Remember From the Previous Lecture that …
2. Lookahead, Backtracking, and Nondeterministic Automata
3. Implementation of Finite Automata in Code
4. From Regular Expression To DFAs

1. Remember From the Previous Lecture that …

2. Lookahead, Backtracking, and Nondeterministic Automata

2.1 A Typical Action of DFA Algorithm

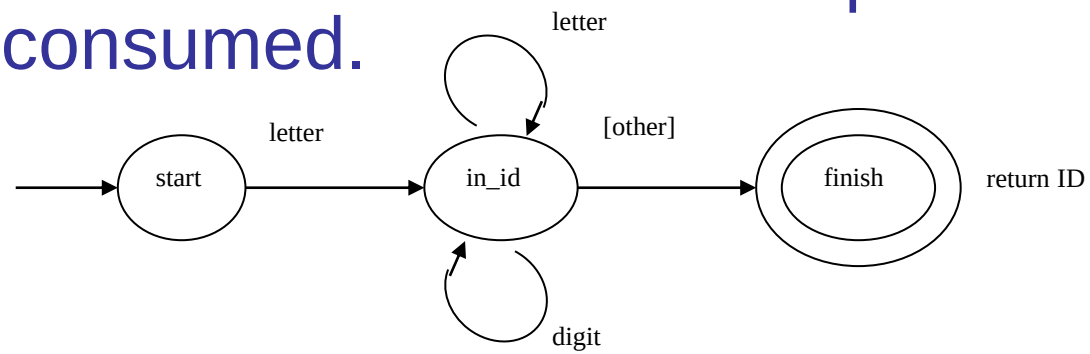
- Making a transition: move the character from the input string to a string that accumulates the characters belonging to a single token (the token string value or lexeme of the token)
- Reaching an accepting state: return the token just recognized, along with any associated attributes.
- Reaching an error state: either back up in the input (backtracking) or to generate an error token.



2. Lookahead, Backtracking, and Nondeterministic Automata

2.2 Finite automaton for an identifier with delimiter and return value

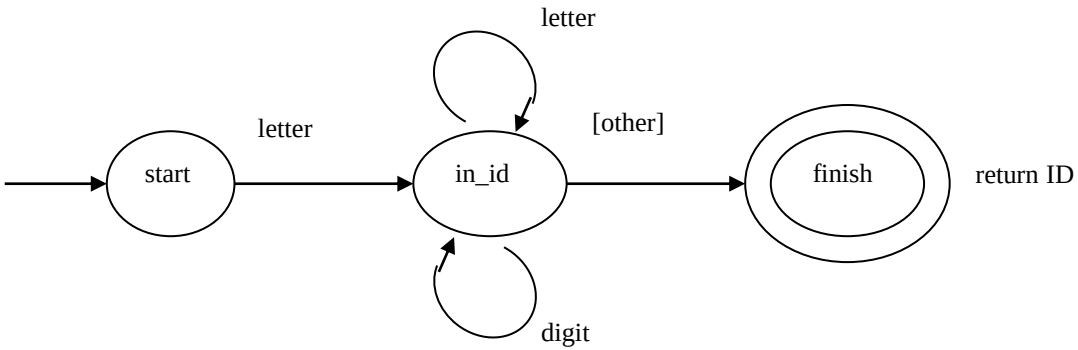
- **The error state** represents the fact that either an identifier is not to be recognized (if came from the start state) or a delimiter has been seen and we should now accept and generate an identifier-token.
- **[other]: indicate that the delimiting character** should be considered look-ahead, it should be returned to the input string and not consumed.

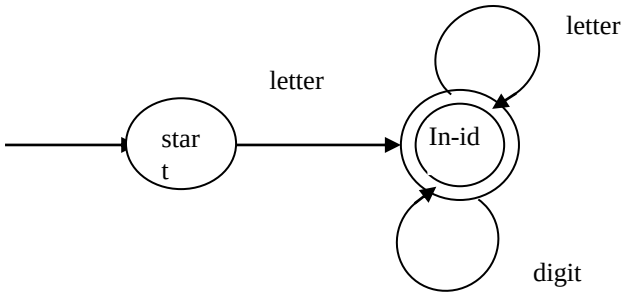


2. Lookahead, Backtracking, and Nondeterministic Automata

2.2 Finite automaton for an identifier with delimiter and return value

- This diagram also expresses the principle of longest sub-string described in Section 2.2.4: the DFA continues to match letters and digits (in state in_id) until a delimiter is found.
- By contrast the old diagram allowed the DFA to accept at any point while reading an identifier string.





2. Lookahead, Backtracking, and Nondeterministic Automata

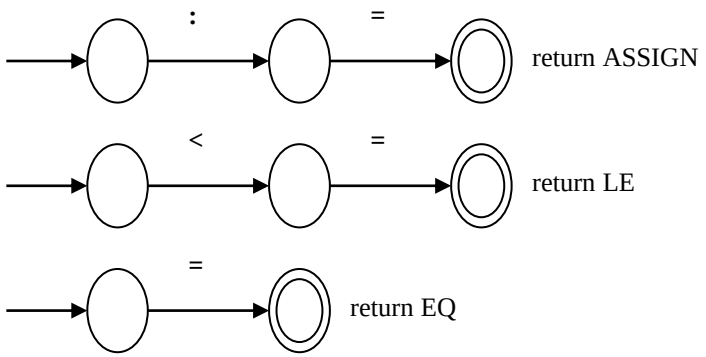
2.3 How to arrive at the start state in the first place

- (combine all the tokens into one DFA)

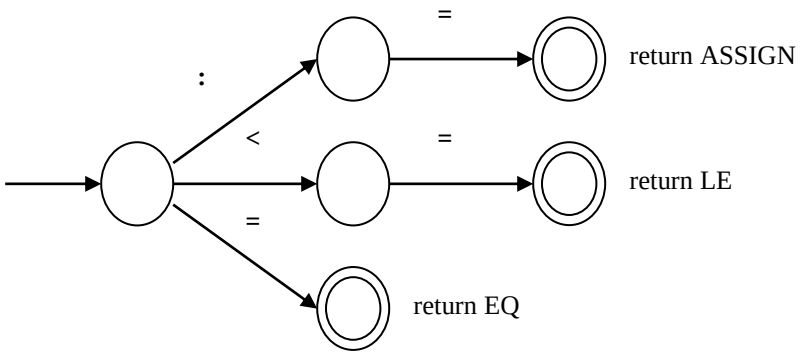
2.3 How to arrive at the start state in the first place

a) Each of these tokens begins with a different character

- Consider the tokens given by the strings : =, <=, and =
- Each of these is a fixed string, and DFAs for them can be written as right



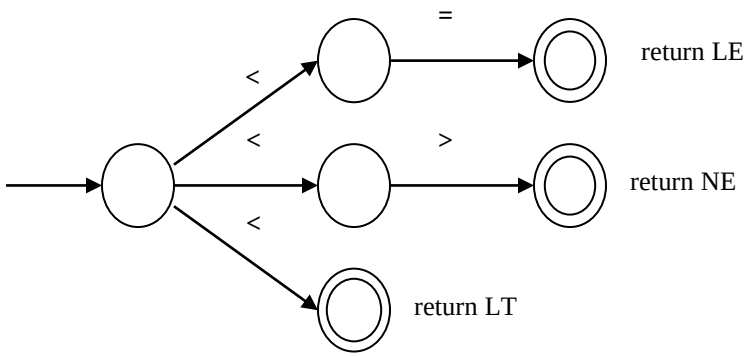
- Uniting all of their start states into a single start state to get the DFA



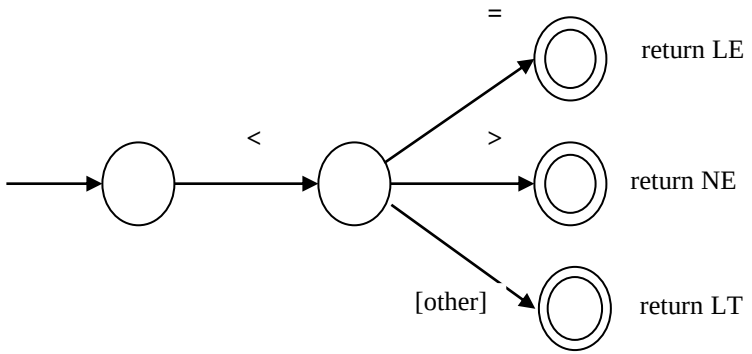
2.3 How to arrive at the start state in the first place

b) Several tokens beginning with the same character

- They cannot be simply written as the right diagram, since it is not a DFA



- The diagram can be rearranged into a DFA



2. Lookahead, Backtracking, and Nondeterministic Automata

2.4 Expand the Definition of a Finite Automaton

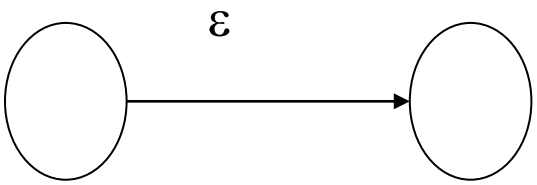
- One solution for the problem is to **expand the definition of a finite automaton**
- **More than one transition from a state** may exist for a particular character

  (NFA: non-deterministic finite automaton,)
- Developing an algorithm for systematically turning these NFA into DFAs

2. Lookahead, Backtracking, and Nondeterministic Automata

2.5 ε-transition

- A transition that may occur without consulting the input string (and without consuming any characters)

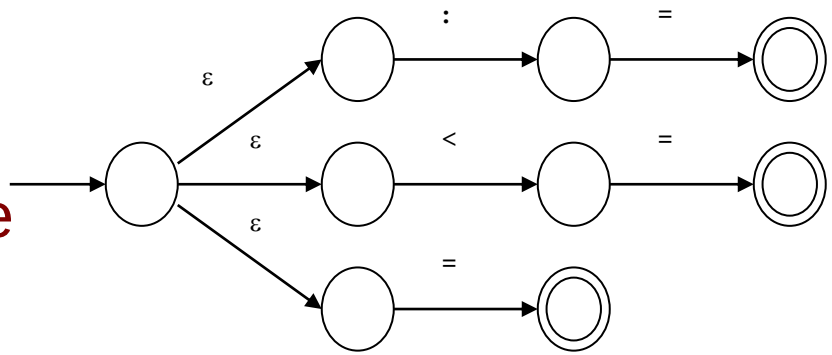


- It may be viewed as a "match" of the empty string.
- (This should not be confused with a match of the character ε in the input)

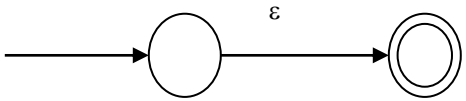
2. Lookahead, Backtracking, and Nondeterministic Automata

2.6 ε-Transitions Used in Two Ways.

- First: to **express a choice of alternatives** in a way without combining states
  - Advantage: keeping the original automata intact and only adding a new start state to connect them



- Second: to explicitly **describe a match of the empty string**.



2. Lookahead, Backtracking, and Nondeterministic Automata

2.7 Definition of NFA

- An **NFA** (non-deterministic finite automaton) *M* consists of
  - an alphabet Σ, a set of states *S*,
  - a transition function $T: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$,
  - a start state *s0* from *S*, and a set of accepting states *A* from *S*

- The language accepted by *M*, written L(*M*),
  - is defined to be the set of strings of characters *c1c2*…. *cn* with
  - each *ci* from $\Sigma \cup \{\varepsilon\}$ such that
  - there exist states *s1* in *T(s0 ,c1*), *s2* in (s*1,* c2),..., *sn* in *T(sn-1 , cn)* with *sn* an element of A.

2. Lookahead, Backtracking, and Nondeterministic Automata

2.8 Some Notes

- Any of the c*i* in *c1c2……cn* may be ε, and

  the string that is actually accepted is the string c,c2. . .*cn* with theε's removed (since the concatenation of *s* withε is *s* itself).

  Thus, **the string *c,c2.. .cn* may actually have fewer than *n* characters in it**

- The sequence of states *s1*,..., *sn* are chosen from the *sets* of states *T(sQ ,* c*1*),..., T(s*n-1,* cn), and this choice will not always be uniquely determined.

  The sequence of transitions that accepts a particular string is not determined at each step by the state and the next input character.

  Indeed, arbitrary numbers ofε's can be introduced into the string at any point, corresponding to any number ofε-transitions in the NFA.

2. Lookahead, Backtracking, and Nondeterministic Automata

2.8 Some Notes

- An NFA does not represent an algorithm.

  However, it can be simulated by an algorithm that backtracks through every non-deterministic choice.

2. Lookahead, Backtracking, and Nondeterministic Automata

2.9 Examples of NFAs

Example 2.10

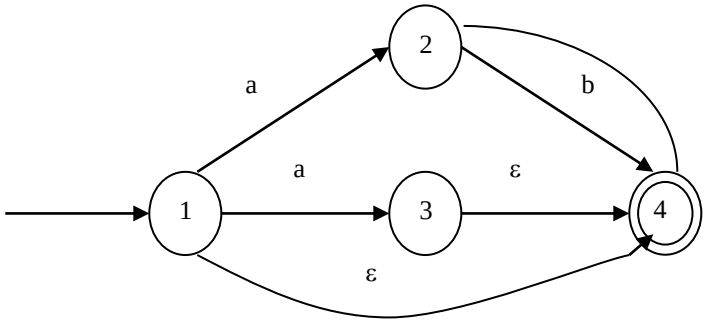
- The string **abb** can be accepted by either of the following sequences of transitions:

  a b ε b

  →1→2→4→2→4

  a ε ε b ε b

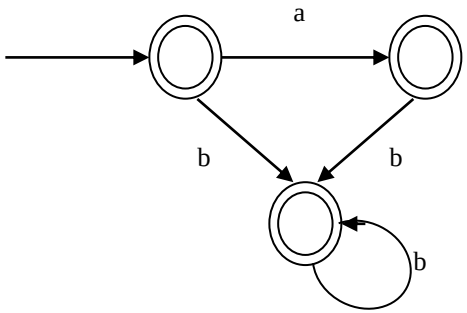
  →1→3→4→2→4→2→4



- This NFA accepts the languages as follows:

  regular expression: (a|ε)b*

  ab+|ab*|b*

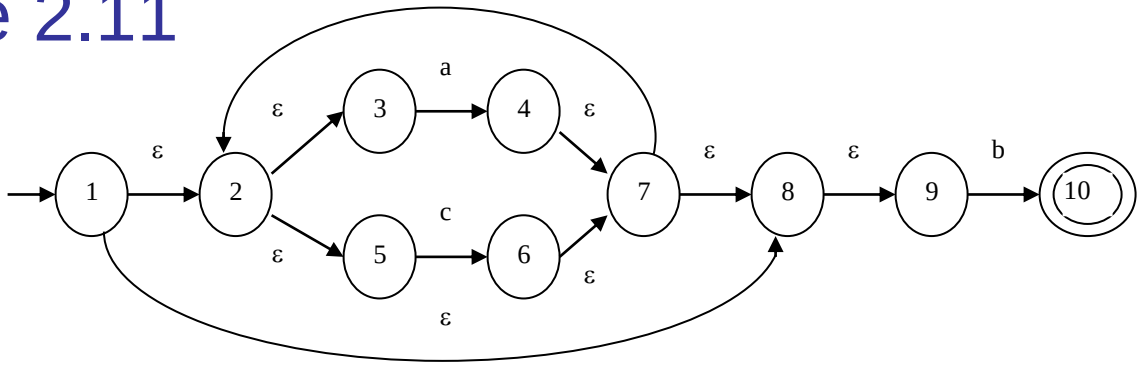


- This DFA accepts the same language.

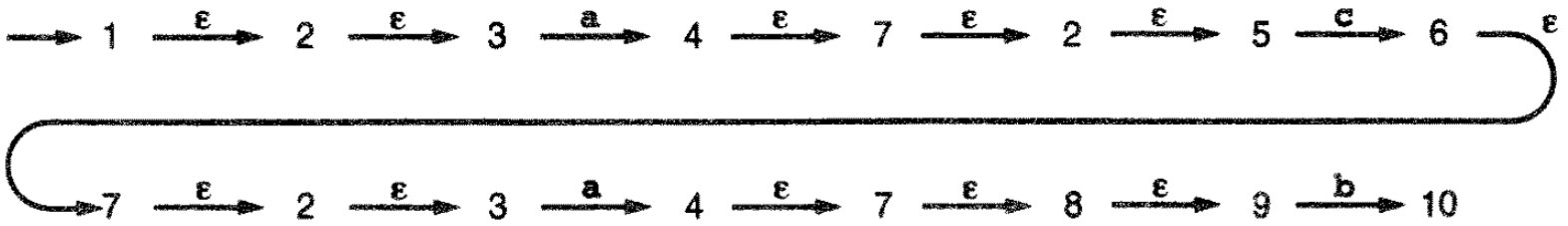
2. Lookahead, Backtracking, and Nondeterministic Automata

2.9 Examples of NFAs

Example 2.11



- It accepts the string acab by making the following transitions:

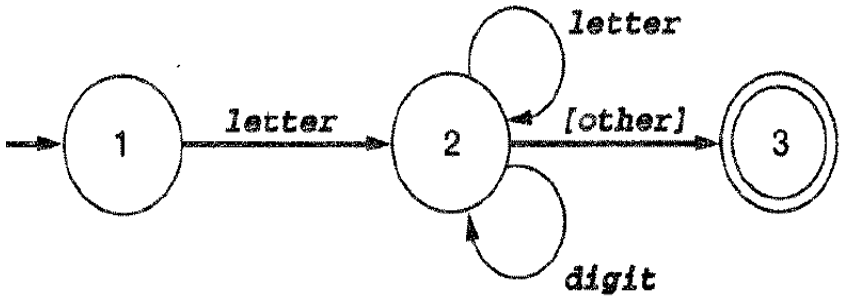


- It accepts the same language as that generated by the regular expression : (a | c) *b

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

a) The first way to simulate the DFA in the following form




```
{ starting in state 1 }
if the next character is a letter then
   advance the input;
   { now in state 2 }
   while the next character is a letter or a digit do
      advance the input; { stay in state 2 }
   end while;
   { go to state 3 without advancing the input }
   accept;
else
   { error or other cases }
end if;
```

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

a) The first way to simulate the DFA in the following form
The code for the DFA accepting identifiers:

Such code uses the position in the code (nested within tests) to maintain the state implicitly, as we have indicated by the comments. This is reasonable if there are not too many states (requiring many levels of nesting), and if loops in the DFA are small. Code like this has been used to write small scanners.

Two drawbacks:

- **It is ad hoc—that is, each DFA has to be treated slightly differently, and it is difficult to state an algorithm that will translate every DFA to code in this way.**
- **The complexity of the code increases dramatically as the number of states rises or, more specifically, as the number of different states along arbitrary paths rises.**

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

b) A better method:

- Using a **variable to maintain the current state** and
- writing the transitions as a doubly nested case statement inside a loop,
- where the first case statement tests the current state and the nested second level tests the input character.

The code of the DFA for identifier:

3. Implementation of Finite Automata in Code

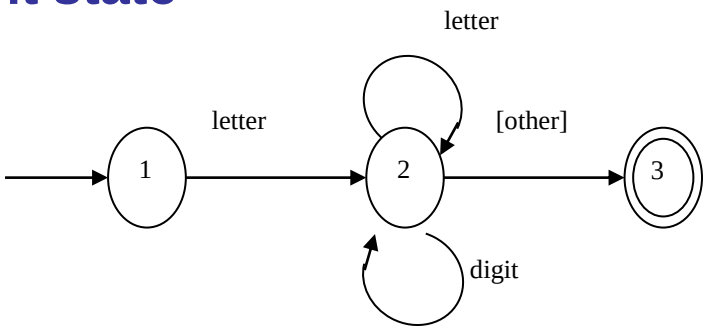
3.1 Ways to Translate a DFA or NFA into Code

b) A better method:

- Using a **variable to maintain the current state**

The code of the DFA for identifier:

```
state := 1; { start }
while state = 1 or 2 do
  case state of
  1:  case input character of
      letter : advance the input;
               state := 2;
      else state := . . . { error or other };
      end case;
  2:  case input character of
      letter, digit: advance the input;
                     state := 2; { actually unnecessary }
      else state := 3;
      end case;
  end case;
end while;
if state = 3 then accept else error ;
```



3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

b) A better method:

- Using a **variable to maintain the current state**

The code of the DFA for identifier:

Notice how this code reflects the DFA directly: transitions correspond to assigning a new state to the *state* variable and advancing the input (except in the case of the "non-consuming" transition from state 2 to state 3).

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

c) Generic code:

Express the DFA as a data structure and then write "generic" code;

A **transition table,** or two-dimensional array, indexed by state and input character that expresses the values of the transition function *T*

	Characters in the alphabet c
States s	**States representing transitions *T (s, c)***

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

c) Generic code:

The transition table of the DFA for identifier:

Input / char / state	letter	digit	other	*Accepting*
1	2			No
2	2	2	[3]	no
3				yes

Brackets indicate "noninput-consuming" transitions

This column indicates accepting states

Assume :the first state listed is the start state

3. Implementation of Finite Automata in Code

3.1 Ways to Translate a DFA or NFA into Code

c) Generic code:

Now we can write code in a form that will implement any DFA, given the appropriate data structures and entries. The following code schema assumes that the transitions are kept in a transition array *T* indexed by states and input characters; that transitions that advance the input (i.e., those not marked with brackets in the table) are given by the Boolean array *Advance*, indexed also by states and input characters; and that accepting states are given by the Boolean array *Accept*, indexed by states. Here is the code scheme:

```
state := 1;
ch := next input character;
while not Accept[state] and not error(state) do
  newstate := T[state,ch];
  if Advance[state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept[state] then accept;
```

3.1 Ways to translate a DFA or NFA into Code

c) Generic code:

Features of Table-Driven Method

Table driven: use tables to direct the progress of the algorithm.

The advantage:

- The size of the code is reduced, the same code will work for many different problems, and the code is easier to change (maintain).

The disadvantage:

- The tables can become very large, causing a significant increase in the space used by the program. Indeed, much of the space in the arrays we have just described is wasted.
- Table-driven methods often rely on table-compression methods such as sparse-array representations, although there is usually a time penalty to be paid for such compression, since table lookup becomes slower. Since scanners must be efficient, these methods are rarely used for them.

3.1 Ways to translate a DFA or NFA into Code

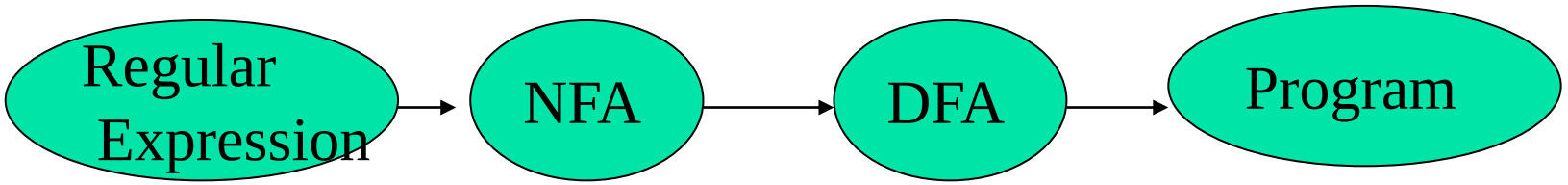
NFAs can be implemented in similar ways to DFAs, except NFAs are nondeterministic,

- there are potentially many different sequences of transitions that must be tried.
- A program that simulates an NFA must store up transitions that have not yet been tried and backtrack to them on failure.

4. From Regular Expression To DFAs

Main Purpose

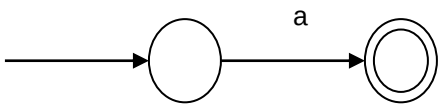
- Study an algorithm:
  - Translating a regular expression into a DFA via NFA.

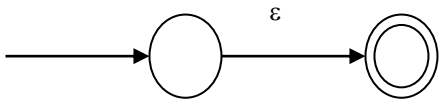


4.1 From a Regular Expression to an NFA

The Idea of Thompson's Construction

- Use ε-transitions
  - to ***"glue together"*** the machine of each piece of a regular expression
  - to form a machine that corresponds to the whole expression
- Basic regular expression
  - The NFAs for basic regular expression of the form a, ε,or φ

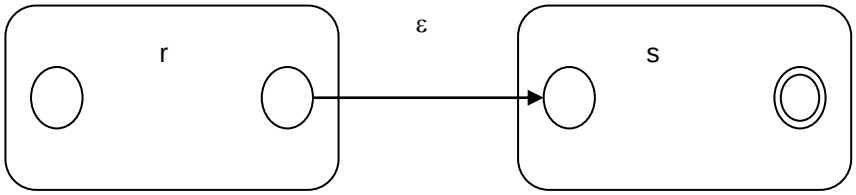




4.1 From a Regular Expression to an NFA

The Idea of Thompson's Construction

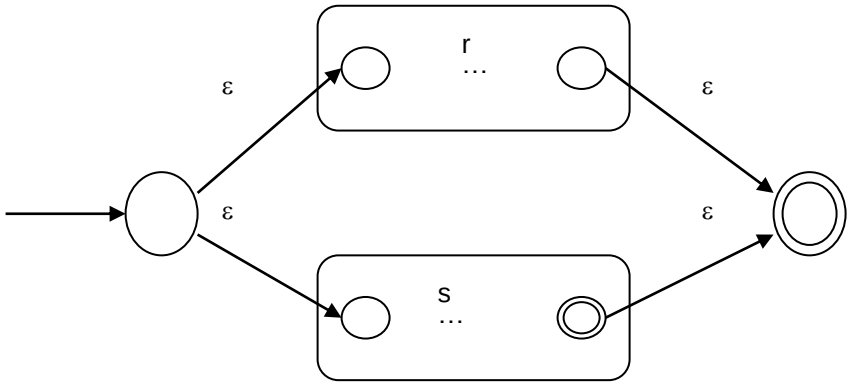
- Concatenation: to construct an NFA equal to ***rs***
  - *To* connect the accepting state of the machine of *r* to the start state of the machine of *s* by anε-transition.
  - The start state of the machine of *r* as its start state and the accepting state of the machine of *s* as its accepting state.
  - This machine accepts *L(rs) = L(r)L(s)* and so corresponds to the regular expression *rs*.



4.1 From a Regular Expression to an NFA

The Idea of Thompson's Construction

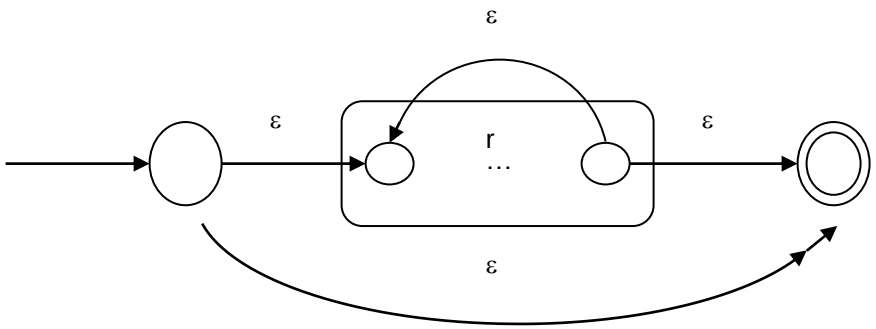
- Choice among alternatives: To construct an NFA equal to ***r* | *s***
  - To add a new start state and a new accepting state and connected them as shown usingε-transitions.
  - Clearly, this machine accepts the language *L(r*|*s)* =*L(r)*U*L (s)*, and so corresponds to the regular expression *r*|*s*.



4.1 From a Regular Expression to an NFA

The Idea of Thompson's Construction

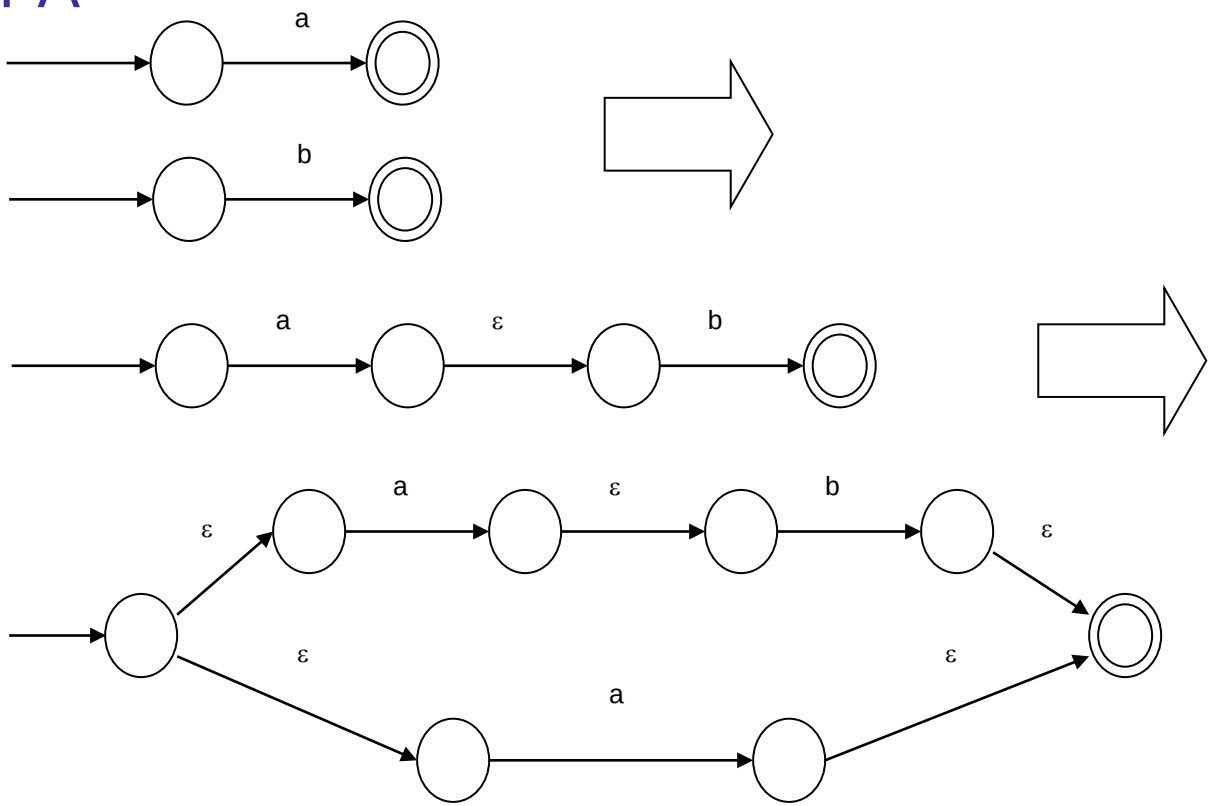
- Repetition: Given a machine that corresponds to *r,* Construct a machine that corresponds to r*
  - To add two new states, a start state and an accepting state.
  - The repetition is afforded by the newε-transition from the accepting state of the machine of *r* to its start state.
  - To draw an ε-transition from the new start state to the new accepting state.
  - This construction is not unique, simplifications are possible in the many cases.



4.1 From a Regular Expression to an NFA

Examples of NFAs Construction

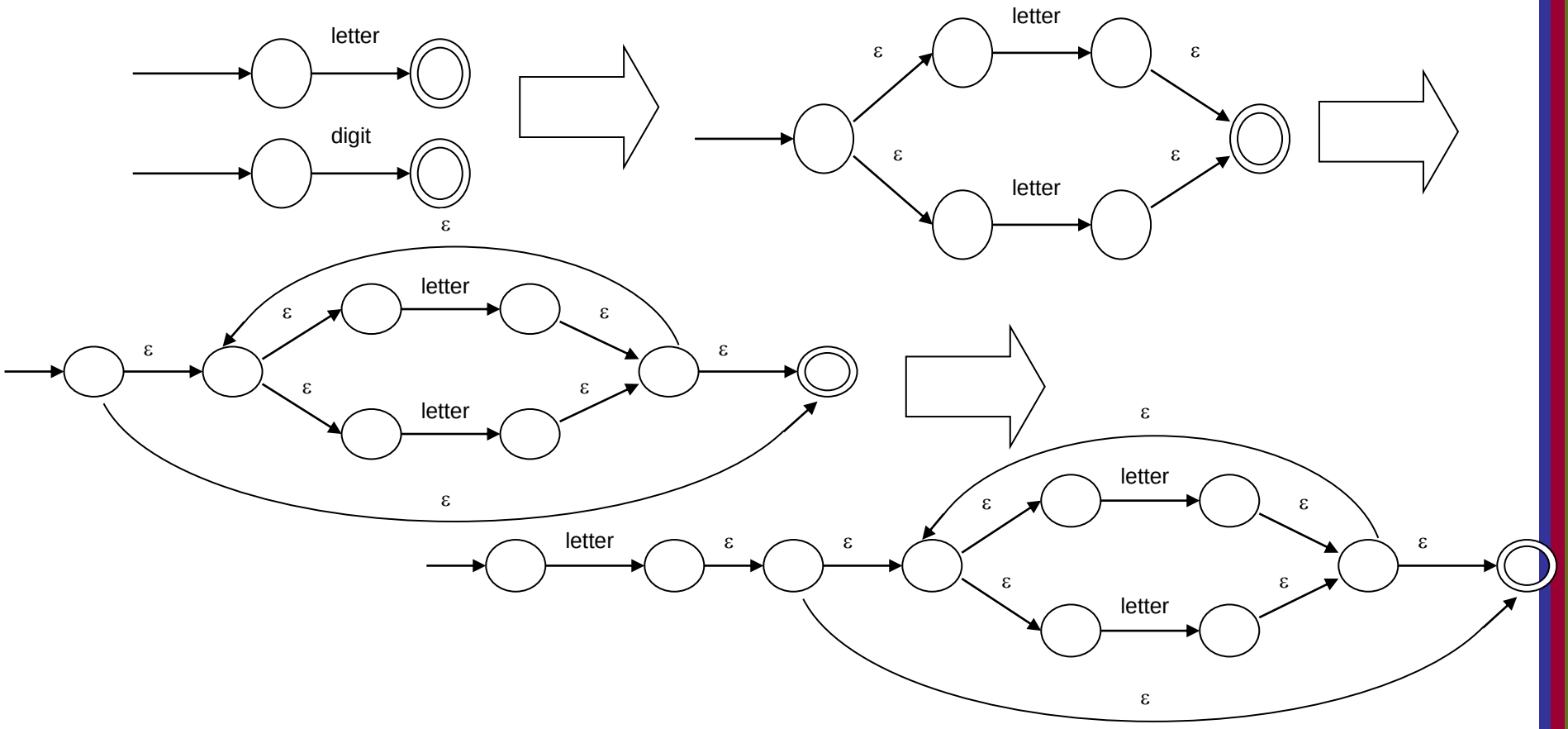
Example 1.12: Translate regular expression **ab|a** into NFA



4.1 From a Regular Expression to an NFA

Examples of NFAs Construction

Example 1.13: Translate regular expression *letter(letter|digit)** into NFA



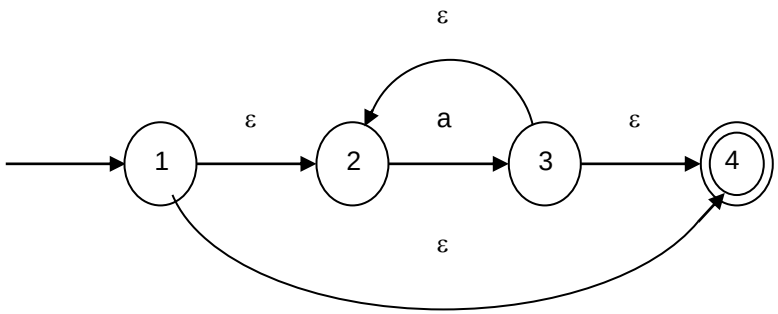
4.2 From an NFA to a DFA
Goal and Methods

- **Goal**
  - ***Given an arbitrary NFA, construct an equivalent DFA. (i.e., one that accepts precisely the same strings)***
- Some methods
  - (1) Eliminating ε-transitions
    - **ε-closure**: the set of all states reachable by ε-transitions from a state or states
  - (2) Eliminating multiple transitions from a state on a single input character.
    - Keeping track of the set of states that are reachable by matching a single character
  - Both these processes lead us to *consider sets of states instead of single states.* Thus, it is not surprising that the DFA we construct has *sets of states* of the original NFA as its states.

4.2 From an NFA to a DFA

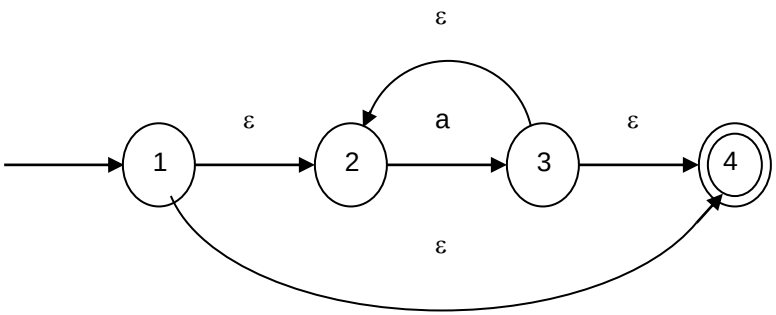
The Algorithm Called Subset Construction.

- *The ε-closure of a Set of states:*
  - The ε-closure of a single state *s* is the set of states reachable by a series of zero or more ε-transitions, and we write this set as $\overline{s}$.
- Example 2.14: regular a*



4.2 From an NFA to a DFA

The Algorithm Called Subset Construction.



$\bar{1} = \{1, 2, 4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2, 3, 4\}$, and $\bar{4} = \{4\}$.

The ε-closure of a set of states : the union of the ε-closures of each individual state.

$$\bar{S} = \bigcup_{s \text{ in } S} \bar{s}$$

$\overline{\{1,3\}} = \bar{1} \cup \bar{3} = \{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$

4.2 From an NFA to a DFA

The *Subset Construction* Algorithm

(1) Compute the ε-closure of the start state of *M;* to obtain new state $\overline{M}$.

(2) For this set, and for each subsequent set, compute transitions on characters *a* as follows.

Given a set *S* of states and a character *a* in the alphabet,

Compute the set

S'_a = { *t* | for some *s* in S there is a transition from *s* to *t* on *a* }.

Then, compute $\overline{S_a}'$, the ε-closure of S'_a.

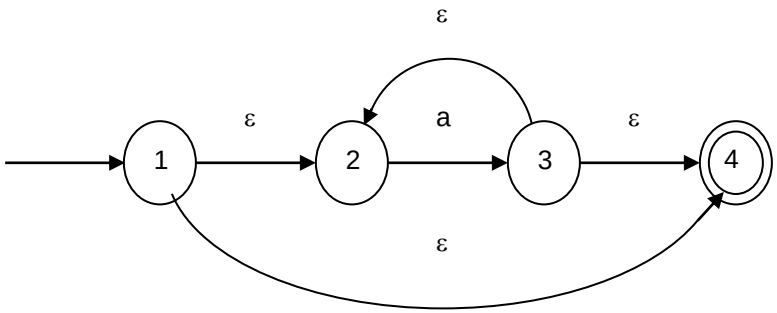
This defines a new state in the subset construction, together with a new transition S$\rightarrow$ $\overline{S_a}'$.

(3) Continue with this process until no new states or transitions are created.

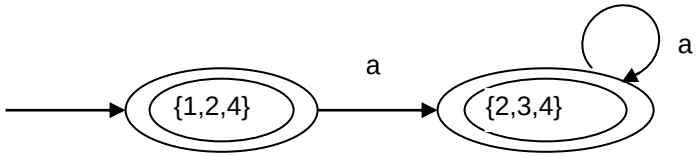
(4) Mark as accepting those states constructed in this manner that contain an accepting state of M.

4.2 From an NFA to a DFA

Examples of Subset Construction

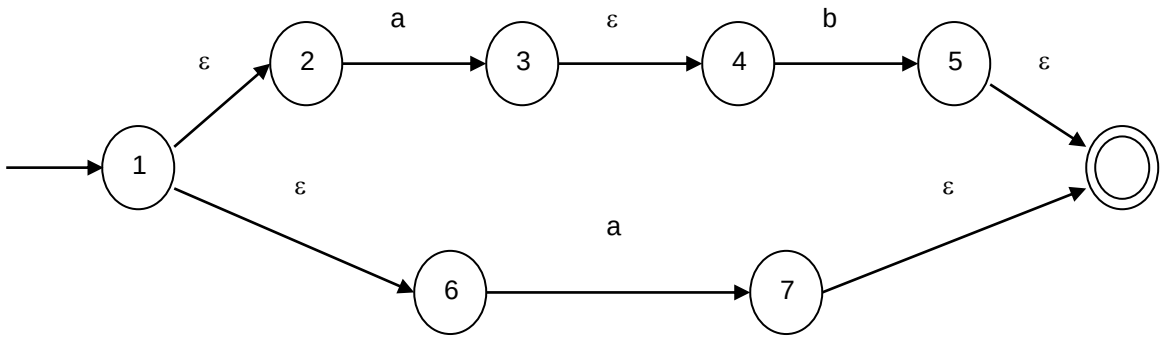


M	ε-closure of M (S)	S'_a
1	1,2,4	3
3	2,3,4	3

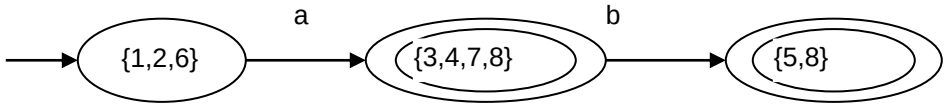


4.2 From an NFA to a DFA

Examples of Subset Construction

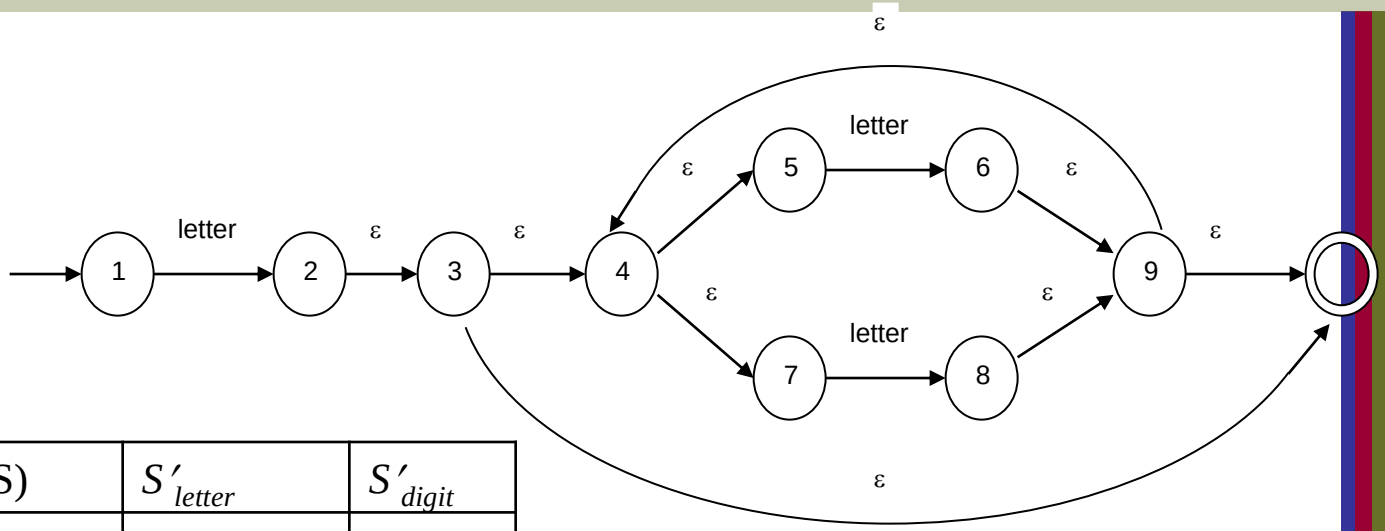


M	ε-closure of M (S)	S'_a	S'_b
1	1,2,6	3,7	
3,7	3,4,7,8		5
5	5,8		

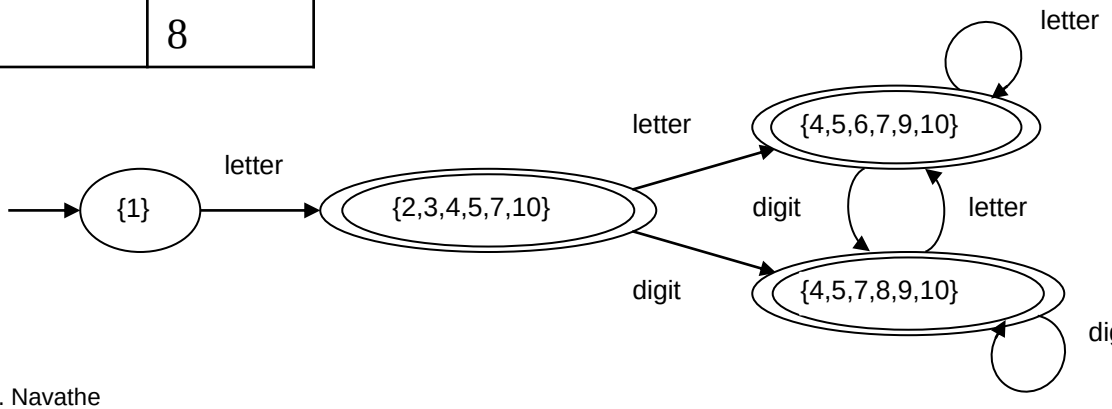


4.2 From an NFA to a DFA

Examples of Subset Construction



M	ε-closure of M (S)	S'_{letter}	S'_{digit}
1	1	2	
2	2,3,4,5,7,10	6	8
6	4,5,6,7,9,10	6	8
8	4,5,7,8,9,10	6	8



RET