COMPILER CONSTRUCTION

Zhihua Jiang

2. Scanning (Lexical Analysis)

Contents

2.1 The Scanning Process

The Function of a Scanner

- **Reading characters** from the source code and form them into logical units called **tokens**
- Tokens are **logical entities** defined as an **enumerated type**
  - Typedef enum

    {IF, THEN, ELSE, PLUS, MINUS, NUM, ID,…}

    TokenType;

The Categories of Tokens

- RESERVED WORDS
  - Such as IF and THEN, which represent the strings of characters “if” and “then”
- SPECIAL SYMBOLS
  - Such as PLUS and MINUS, which represent the characters “+” and “-“
- OTHER TOKENS
  - Such as NUM and ID, which represent numbers and identifiers

Relationship between Tokens and its String

- The string is called **STRING VALUE** or **LEXEME** of token
- Some tokens have only **one lexeme**, such as reserved words
- Some token may have infinitely **many lexemes**, such as ID

2.2 Regular Expression

Some Relative Basic Concepts

- Regular expressions
  - represent **patterns** of strings of characters.
- A regular expression *r*
  - completely defined by **the set of strings** it matches.
  - The set is called the **language** of **r** written as *L*(**r**)
  - **r** is the character *r* used as a pattern
- The set elements
  - referred to as **symbols**
- This set of legal symbols
  - called the **alphabet** and written as the Greek symbol ∑

More About Regular Expression

2.2.1 Definition of Regular Expression

2.2.2 Extension to Regular Expression

2.2.3 Regular Expressions for Programming Language Tokens

2.2.1 Definition of Regular Expressions

Basic Regular Expressions

- The single characters from alphabet matching themselves
  - **a** matches the character *a* by writing $L(\mathbf{a})=\{ a \}$
  - ε denotes **the empty string**, by $L(\boldsymbol{\varepsilon})=\{\varepsilon\}$
  - **Φ** matches **no string** at all, by $L(\boldsymbol{\Phi})=\{ \}$

Regular Expression Operations

- **Choice among alternatives**, indicated by the meta-character |
- **Concatenation**, indicated by juxtaposition
- **Repetition or "closure",** indicated by the meta-character *

Choice Among Alternatives

- If **r** and **s** are regular expressions, then **r|s** is a regular expression which matches any string that is matched either by **r** or by **s**.
- In terms of languages, the language **r|s** is the **union of language** **r** and **s**, or $L(\mathbf{r}|\mathbf{s})=L(\mathbf{r}) \cup L(\mathbf{s})$
- A simple example, $L(\mathbf{a}|\mathbf{b})=L(\mathbf{a}) \cup L(\mathbf{b})=\{a, b\}$
- Choice can be extended to more than two alternatives. E.g., $L(\mathbf{a}|\mathbf{b}|\mathbf{c})=\{a, b, c\}$

Concatenation

- If **r** and **s** are regular expressions, then **rs** is their concatenation which matches any string that is the concatenation of two strings, the first of which matches **r** and the second of which matches **s**.
- In term of generated languages, the concatenation set of strings S_1S_2 is the set of strings of S_1 **appended by all the strings** of S_2.
- A simple example, (**a**|**b**)**c** matches *ac* and *bc*
- Concatenation can also be extended to more than two regular expressions.

Repetition

- The repetition operation of a regular expression **r**, is written **r***, which matches **any finite concatenation** of strings, each of which matches **r**.
- A simple example, **a*** matches the strings ε, *a*, *aa*, *aaa*,…
- In term of generated language, given a set *S* of strings, $S^* = S^0 \cup S^1 \cup S^2 \cup S^3 \ldots$ is an **infinite set union**, but each element in it is a **finite concatenation** of string from *S*

Precedence of Operations and Use of Parentheses

- The **standard convention:**
  - Repetition* has the highest precedence
  - Concatenation is given the next highest
  - | is given the lowest
- A simple example:
  - **a|bc*** is interpreted as **a|(b(c*))**
- Parentheses is used to indicate a different precedence

Name for regular expression

- Give a name to a **long regular expression**
  - *digit* = 0|1|2|3|4……|9
  - (0|1|2|3……|9)(0|1|2|3……|9)* *digit digit**

Definition of Regular Expression

- A regular expression is one of the following:
  (1) A basic regular expression **a**, consisting of a single legal character *a* from alphabet ∑, or meta-character **ε** or **Φ.**
  (2) The form **r|s**, where **r** and **s** are regular expressions
  (3) The form **rs**, where **r** and **s** are regular expressions
  (4) The form **r***, where **r** is a regular expression
  (5) The form **(r)**, where **r** is a regular expression

  – Parentheses do not change the language.

Examples of Regular Expressions

Example 1:

- $\sum = \{ a,b,c \}$
- the set of all strings over this alphabet that contain **exactly one *b***.
- **(a|c)*b(a|c)***

Example 2:

- $\sum = \{ a,b,c \}$
- the set of all strings that contain **at most one *b***.
- **(a|c)*|(a|c)*b(a|c)*** or **(a|c)*(b|ε)(a|c)***
- the same language may be generated by different regular expressions.

Examples of Regular Expressions

Example 3:

- $\sum = \{ a, b \}$
- the set of strings consists of **a single *b*** surrounded by **the same number of *a*'s**.
- $S = \{b, aba, aabaa, aaabaaa, \ldots\ldots\} = \{ a^n b a^n \mid n \geq 0 \}$
- This set can not be described by a regular expression.

  "regular expression can't count "

- *not all sets of strings can be generated by regular expressions*.

Examples of Regular Expressions

Example 4:

- $\sum$={ *a*,*b*,*c*}
- The strings contain **no two consecutive *b*'s**
- **((a|c)* | (b(a|c))*)*** // force *a* or *c* to come after/before every *b*
- or **((a | c) | (b(a | c)))*** // (**r***|**s***)*=(**r**|**s**)*
- or **(a | c | ba | bc)***
  - Not yet the perfect answer (why?)
  - Can not produce any string which ends/begins with *b*

Examples of Regular Expressions

Example 4:

- $\sum = \{ a, b, c \}$
- The strings contain **no two consecutive *b*'s**

The perfect regular expression

- **(a | c | ba | bc)* (b|ε) or (b|ε) (a | c | ab | cb)*** // mirror image
- **(not b |b not b)*(b|ε)** // rename not b = a|c

Examples of Regular Expressions

Example 5:

- $\sum = \{ a, b, c \}$
- **((b|c)* a(b|c)*a)* (b|c)***
- Determine a concise description of the language where the strings contain **an even number of *a*'s**

((not a)* a (not a)* a)* (not a)*

// rename not a = b|c

// but still can't know the exact number of *a*

2.2.2 Extensions to Regular Expression

List of New Operations

1) one or more repetitions

   r^{+}

2) any character

   period “. ”

3) a range of characters

   [0-9]: 0|1|…|9

   [**a-z**]: **a**|**b**|…|**z**

   [**a-zA-Z**]:**a**|**b**|…|**z**|**A**|**B**|…|**Z**

List of New Operations

4) any character not in a given set

~(a|b|c) a character neither *a* nor *b* nor *c*

5) optional sub-expressions

– **r**? the strings matched by **r** are optional

E.g., Number, Reserved word and Identifiers

Numbers

- *nat* = [0-9]$^+$
- *signedNat* = (+|-)?*nat*
- *number* = *signedNat*(.*nat*)? (E *signedNat*)?

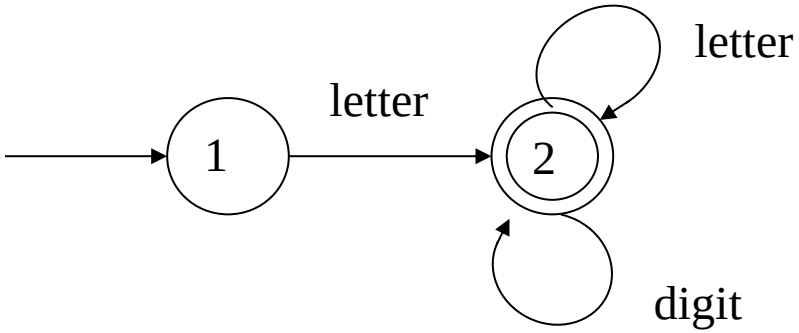
Reserved Words and Identifiers

- *reserved* = if | while | do |……
- *letter* = [a-zA-Z]
- *digit* = [0-9]
- *identifier = letter(letter|digit)**

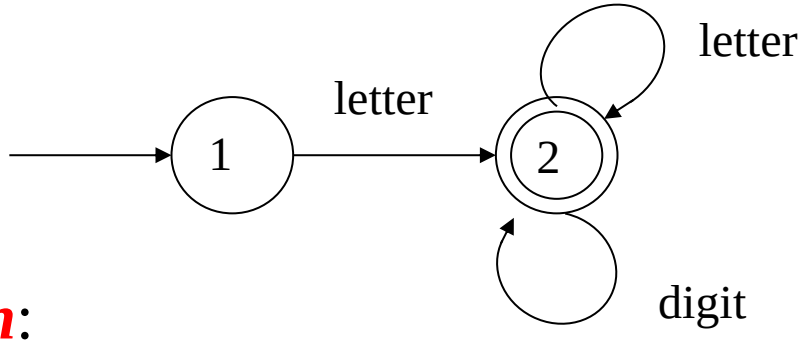
2.3 FINITE AUTOMATA

Introduction to Finite Automata

- Finite automata (finite-state machines) are a **mathematical way** of describing particular kinds of algorithms.
- A **strong relationship** between finite automata and regular expression
  - *Identifier* = *letter* (*letter* | *digit*)*



Introduction to Finite Automata



- ***Transition***:
  - Record a change from one state to another upon a match of the character or characters by which they are labeled.
- ***Start state***:
  - The recognition process begins
  - Drawing an unlabeled arrowed line to it coming “from nowhere”
- ***Accepting states***:
  - Represent the end of the recognition process.
  - Drawing a double-line border around the state in the diagram

More About Finite Automata

2.3.1 Definition of Deterministic Finite Automata

2.3.2 Lookahead, Backtracking ,and Nondeterministic Automata

2.3.3 Implementation of Finite Automata in Code

2.3.1 Definition of Deterministic Finite Automata

The Concept of DFA

DFA: Automata where the **next state is uniquely** given by the current state and the current input character.

Definition of a DFA:

A **DFA** (Deterministic Finite Automation) ***M*** consist of

(1) an alphabet $\sum$,

(2) a set of states S,

(3) a transition function $T : S \times \sum \rightarrow S$,

(4) a start state $s_0 \in S$,

(5) a set of accepting states $A \subset S$

The Concept of DFA

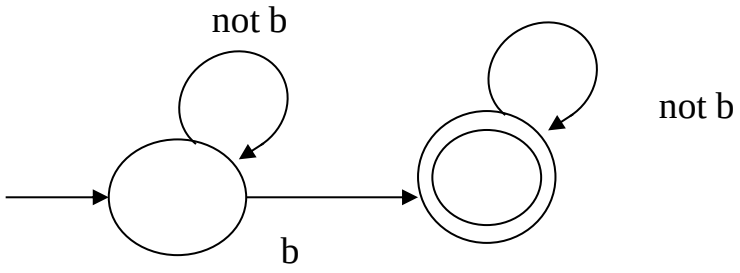
The ***language*** accepted by a DFA M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2c_3\ldots c_n$ with each $c_i \in \sum$ such that there exist states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, $s_n = T(s_{n\text{-}1}, c_n)$, with s_n an element of A (i.e. an accepting state).

Accepting state s_n means the same thing as the diagram:

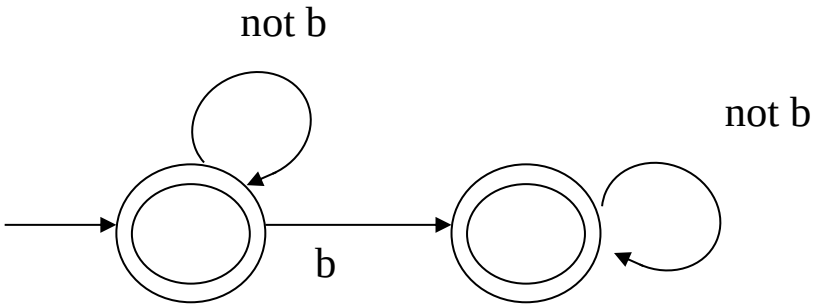
$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \rightarrow \ldots\ldots\ldots s_{n\text{-}1} \xrightarrow{c_n} s_n$$

Examples of DFA

Example 2.6: exactly accept one *b*



Example 2.7: at most one *b*



Examples of DFA

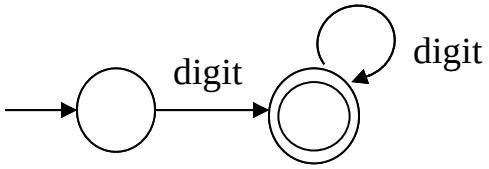
Example 2.8: digit = [0-9]

nat = digit$^+$

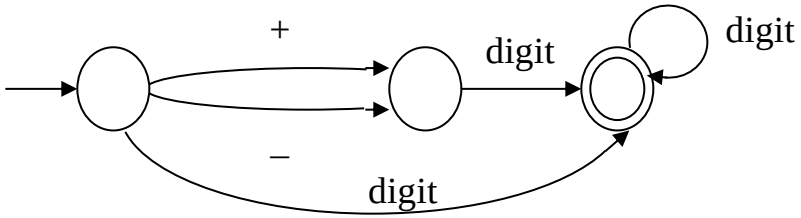
signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?

A DFA of nat:



A DFA of signedNat:



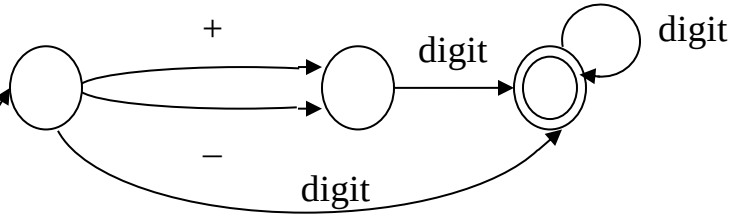
Examples of DFA

Example 2.8: digit = [0-9]

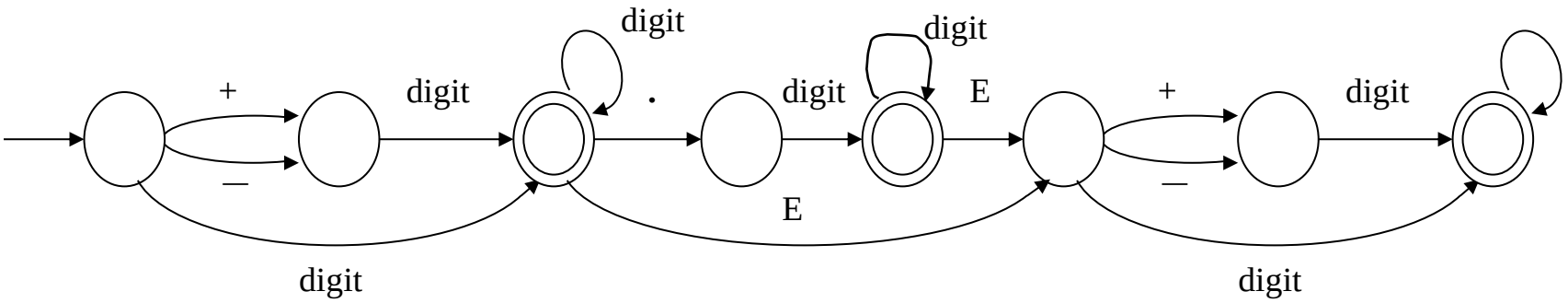
nat = digit +

signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?



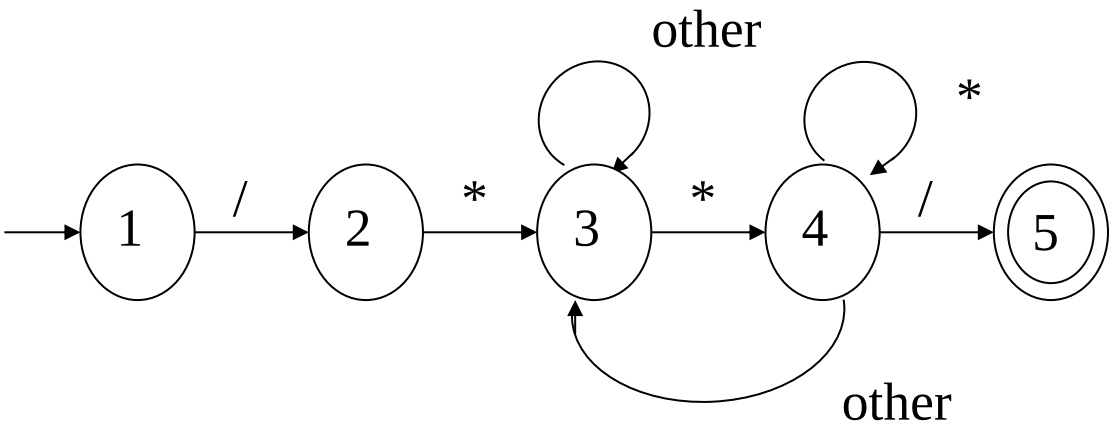
A DFA of Number:



Examples of DFA

Example 2.9 : A DFA of C Comments

(easier than write down a regular expression)



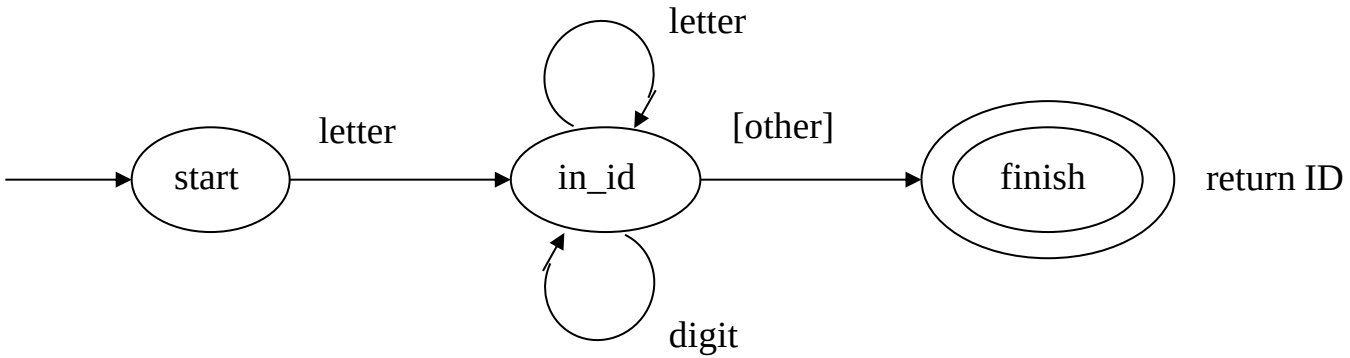
2.3.2 Lookahead, Backtracking, and Nondeterministic Automata

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
- **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.

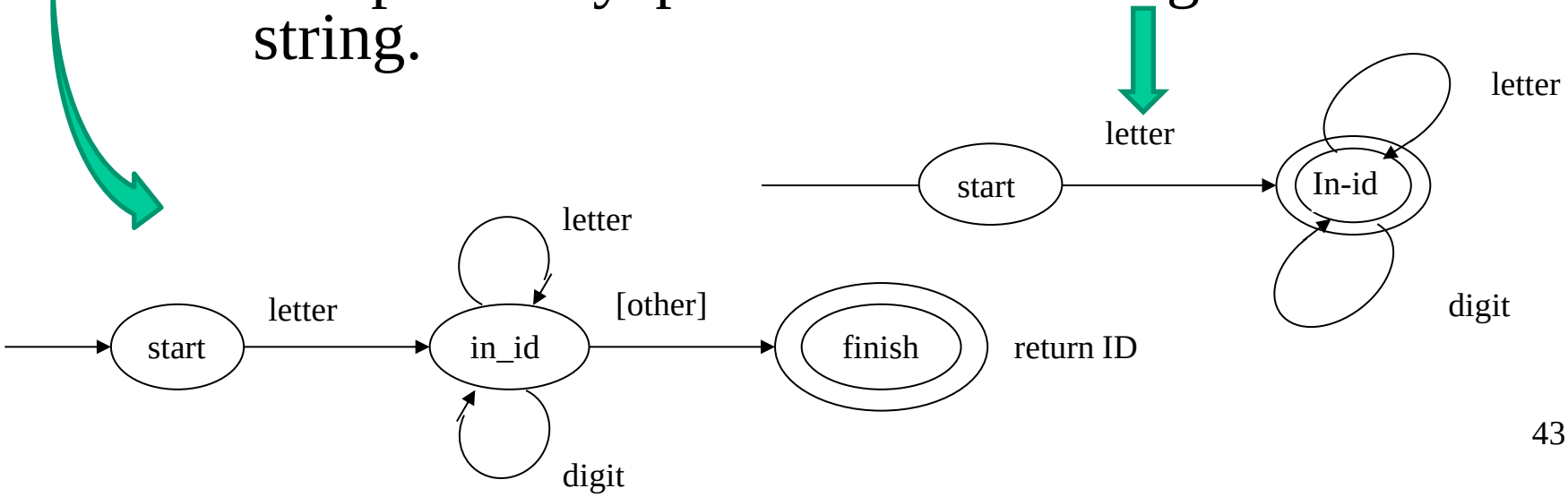
Finite automaton for an identifier with delimiter and return value

- **The error state** represents the fact that either an identifier is not to be recognized 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.



Finite automaton for an identifier with delimiter and return value

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

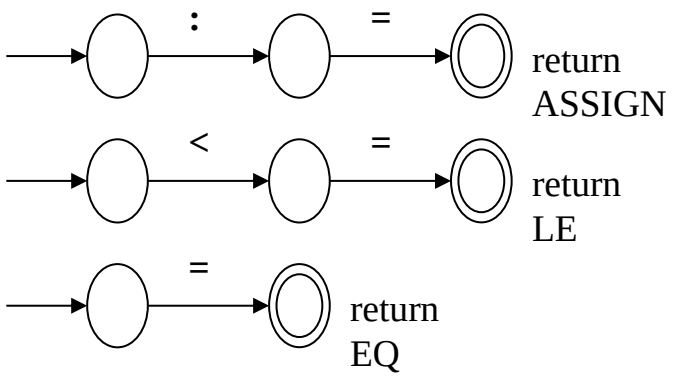


How to arrive at the start state in the first place

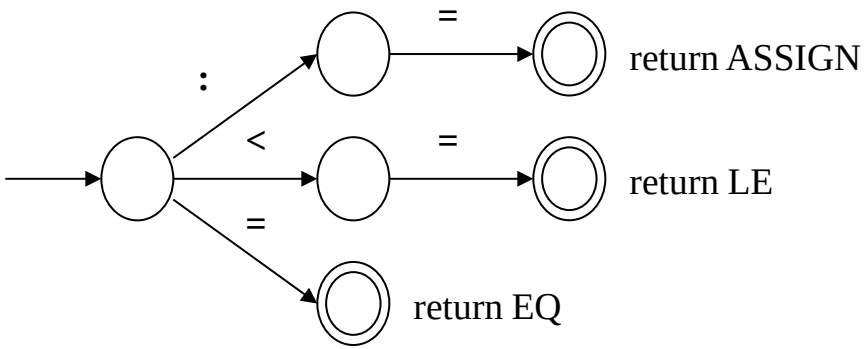
(combine all the tokens into one DFA)

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 follows

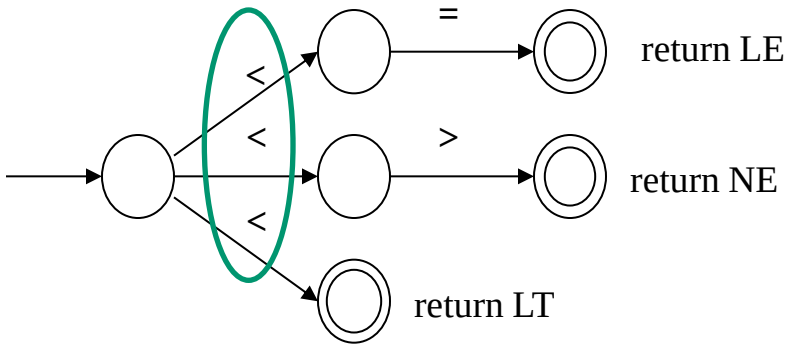


- Uniting all of their start states into a single start state to get the DFA

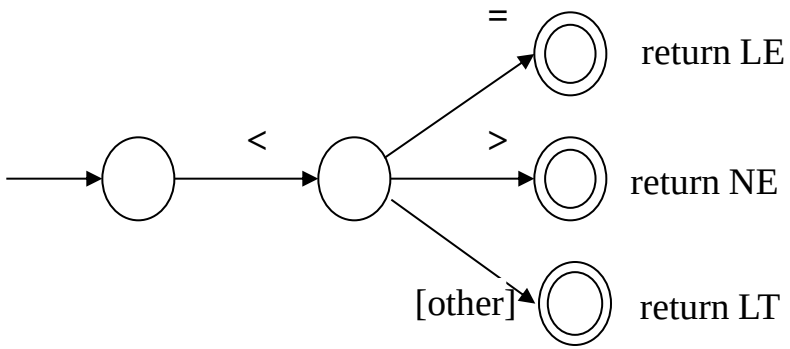


Several tokens beginning with the same character

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



- The diagram can be rearranged into a DFA



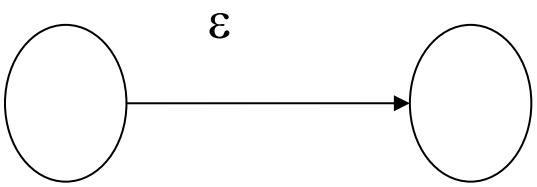
Expand the Definition of a Finite Automaton

- 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

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

ε-transition

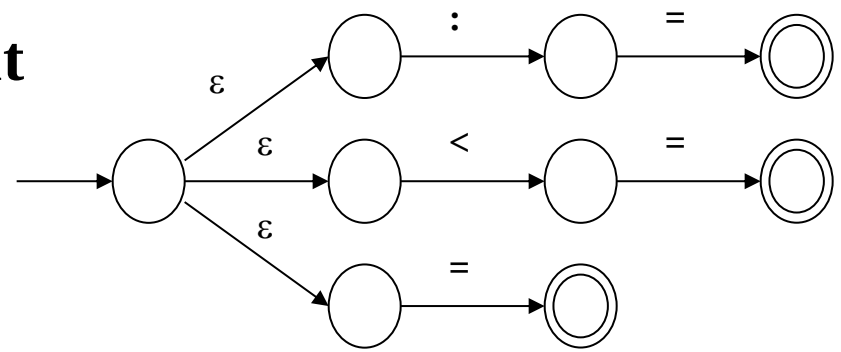
- A transition that may occur without consulting the input string (and without consuming any character)



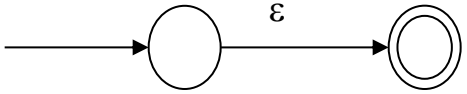
- It may be viewed as a ***"match" of the empty string***

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**.



Definition of NFA

- An **NFA (non-deterministic finite automaton)** M consists of
  - an alphabet Σ
  - a set of states
  - a transition function $T: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$
  - a start state s_0 from S
  - 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 $c_1\, c_2 \ldots. c_n$
  - each c_i from $\Sigma \cup \{\varepsilon\}$
  - there exist states s_1 in $T(s_0, c_1)$, s_2 in $T(s_1, c_2)$,..., s_n in $T(s_{n-1}, c_n)$ with s_n an element of A.

Some Notes

- Any of the c_i in $c_1\, c_2 \ldots c_n$ may be ε, **the string** $c_1\, c_2 \ldots c_n$ **may actually have fewer than *n* characters in it**

- The sequence of states s_1,..., s_n are chosen from the ***sets*** of states $T(s_0\ ,\ c_1)$,..., $T(s_{n\text{-}1}\ ,\ c_n\ )$, the sequence of transitions that accepts a particular string is not determined at each step by the state and the next input character.

- An NFA does not represent an algorithm. However, it can be simulated by an algorithm that backtracks through every non-deterministic choice.

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

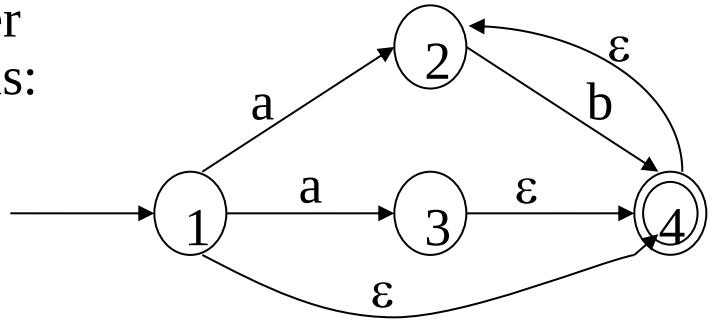
  a ε ε b ε b
  →1→3→4→2→4→2→4

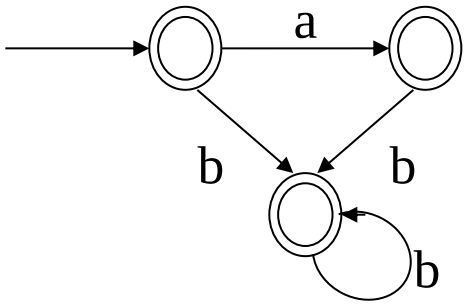
- This NFA accepts the languages as follows:

  regular expression**: (a|ε)b***

  or **ab*|b***

- A simpler DFA accepts the same language.

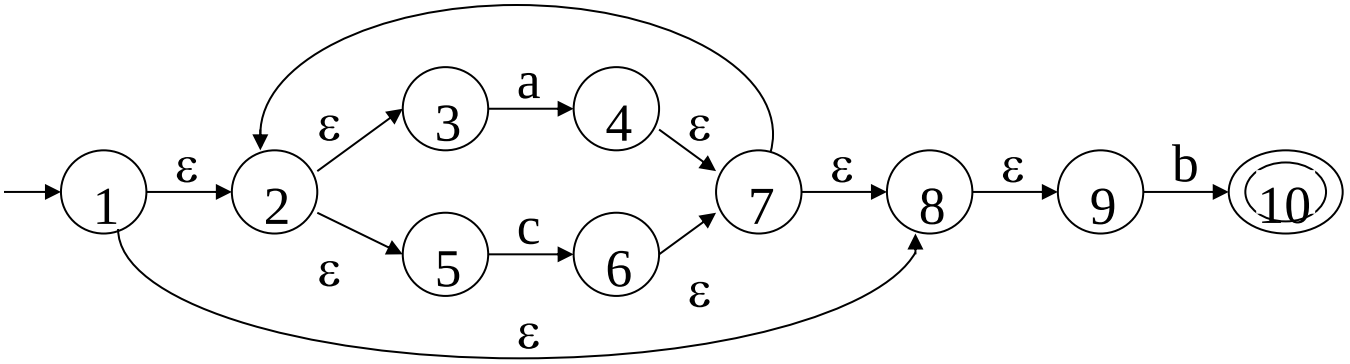




Examples of NFAs

Example 2.11

- It accepts the string *acab* by making the following transitions:
  - (1)(2)(3)a(4)(7)(2)(5)(6)c(7)(2)(3)a(4)(7)(8)(9)b(10)
- It accepts the same language as that generated by the regular expression : **(a | c)*b**

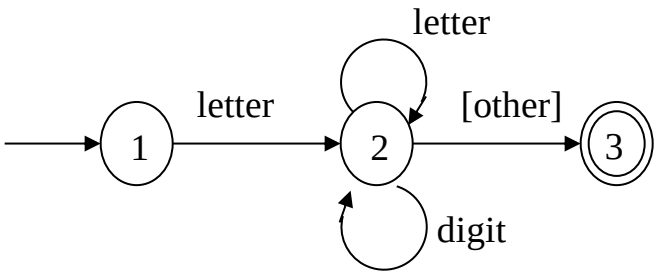


2.3.3 Implementation of Finite Automata in Code

Ways to Translate a DFA into Code

The code for the DFA accepting identifiers:

- { 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;**



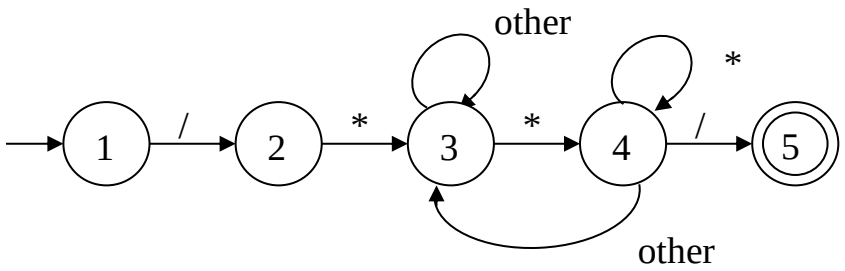
Two drawbacks:

- 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.

Ways to Translate a DFA into Code

The Code of the DFA accepting C comments:

- { state 1 }
- **if** *the next character is "/"* **then** *advance the input; (* state *2 }*
- **if** *the next character is* " * " then
- *advance the input; {* state 3 }
- *done* := **false;**
- **while not** *done* **do**
- **while** *the next input character is not* "*" do
- *advance the input;* **end while;**
- *advance the input; (* state 4 }
- **while** *the next input character is* "*" do
- *advance the input;*
- **end while;**
- **if** *the next input character is "/"* **then**
- *done* := **true; end if;**
- *advance the input;*
- **end while;**
- *accept; {* state 5 }
- **else** { other processing }
- **end if;**
- **else** { other processing }
- **end if;**



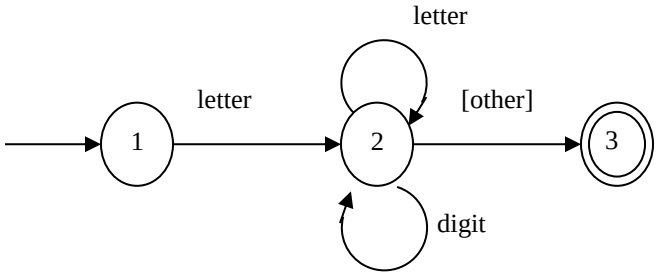
Ways to Translate a DFA into Code

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:

- 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 }

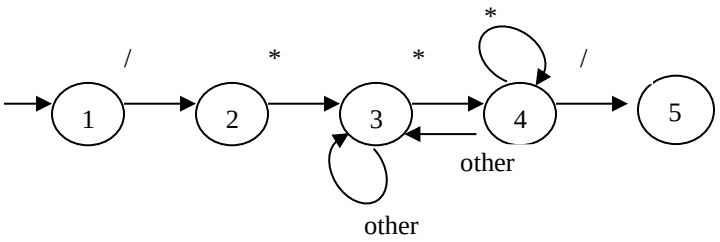


- else state := 3;
- end case;
- end case;
- end while;
- if state = 3 then accept else error;

Ways to Translate a DFA into Code

The code of the DFA for C comments

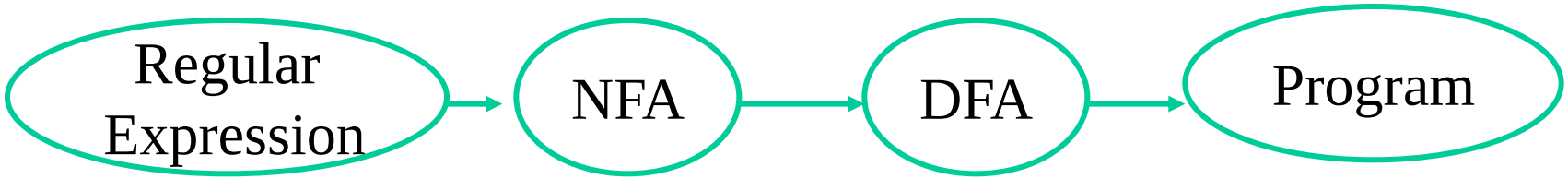
```
state := 1;  { start }
while state = 1, 2, 3 or  4 do
 case state of
 1:   case input character of
        "/" :  advance the input;
               state := 2;
        else state :=...{ error or other};
        end case;
 2:   case input character of
        "*":  advance the input;
               state :=  3;
        else state :=...{ error or other };
        end case;
 3:   case input character of
        "*":  advance the input;
               state := 4;
        else  advance the input   { and stay in state 3 };
        end  case;

 4:   case input character of
        "/“: advance the input;
             state := 5;
        "*": advance the input; { and stay in state 4 }
        else  advance the input;
             state := 3;
        end case;
      end case;
 end while;
 if state = 5 then accept else error
```



2.4 From Regular Expression To DFAs

Main Purpose

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



Contents

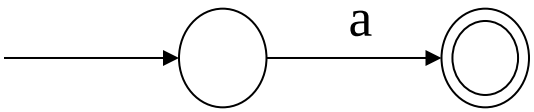
- From a Regular Expression to an NFA
- From an NFA to a DFA
- Simulating an NFA using Subset Construction
- Minimizing the Number of States in a DFA

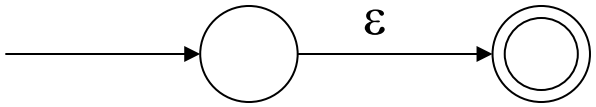
2.4.1 From a Regular Expression to an NFA

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

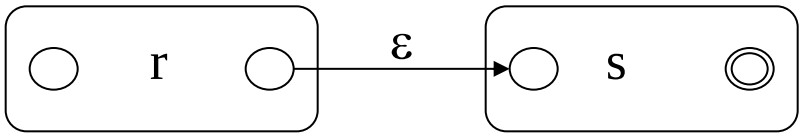
- Basic regular expression
  - The NFAs for basic regular expression of the form **a** or **ε**





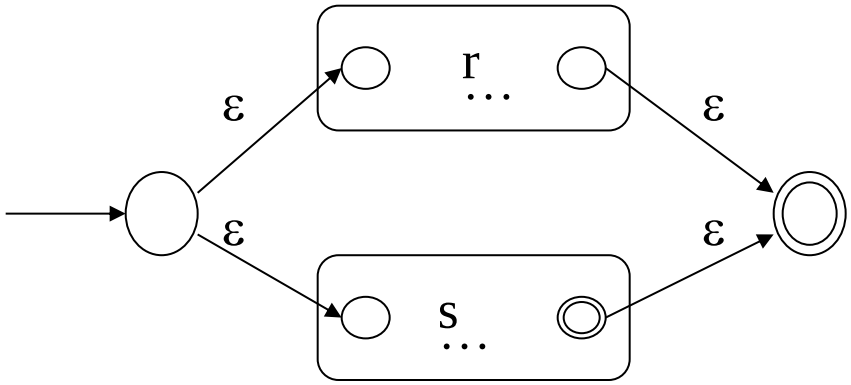
The Idea of Thompson's Construction

- 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 an ε-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 $L(\mathbf{rs}) = L(\mathbf{r})L(\mathbf{s})$ and so corresponds to the regular expression **rs**.



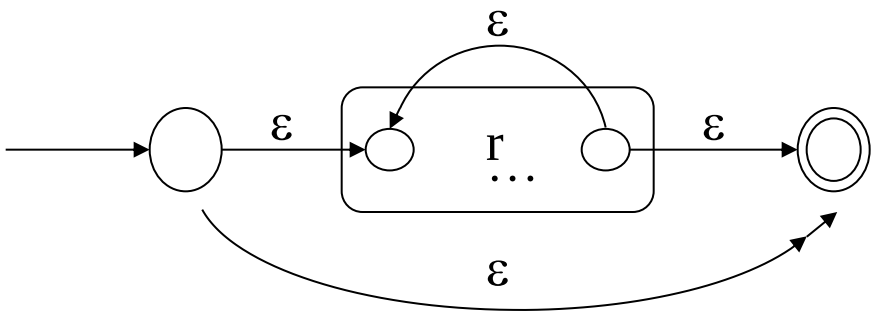
The Idea of Thompson's Construction

- Choice among alternatives: to construct an NFA equal to **r | s**
  - To add a new start state and a new accepting state and connect them as shown using ε-transitions.
  - This machine accepts the language $L(\mathbf{r}|\mathbf{s}) = L(\mathbf{r}) \cup L(\mathbf{s})$, and so corresponds to the regular expression **r|s**.



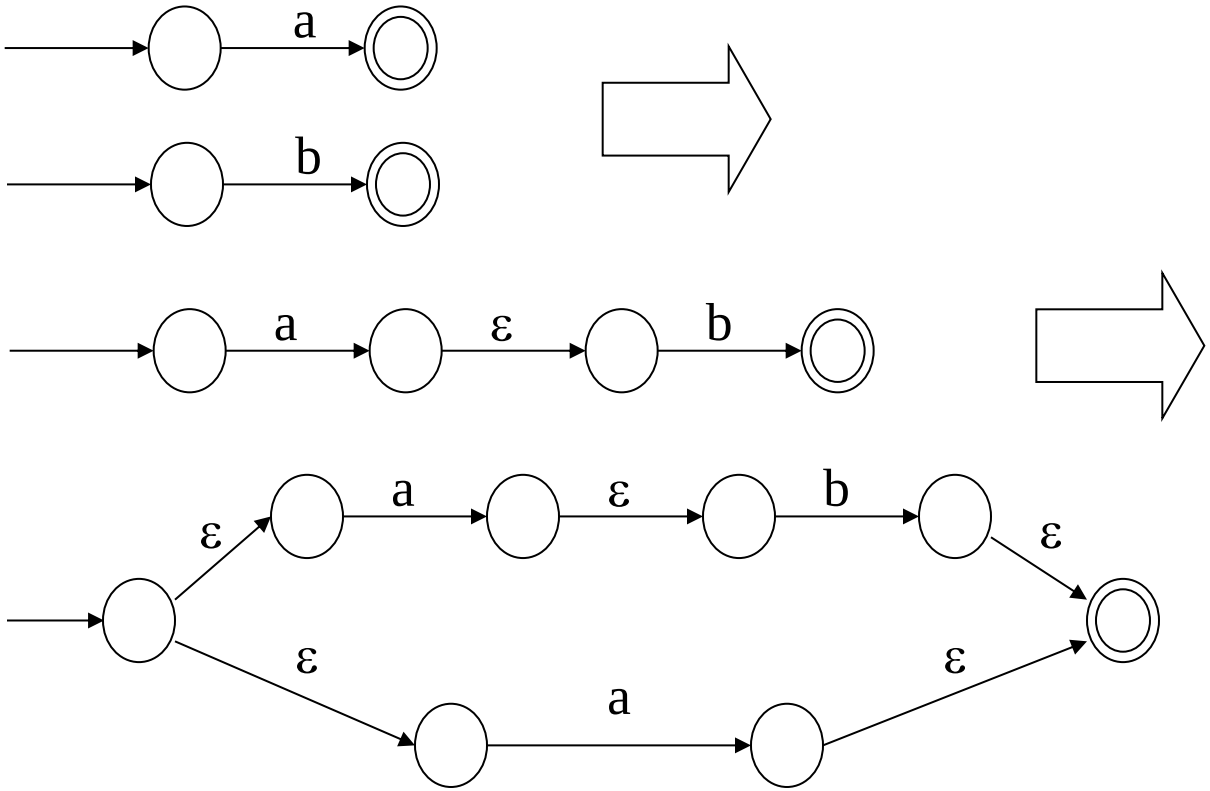
The Idea of Thompson's Construction

- 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 new ε-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 many cases.



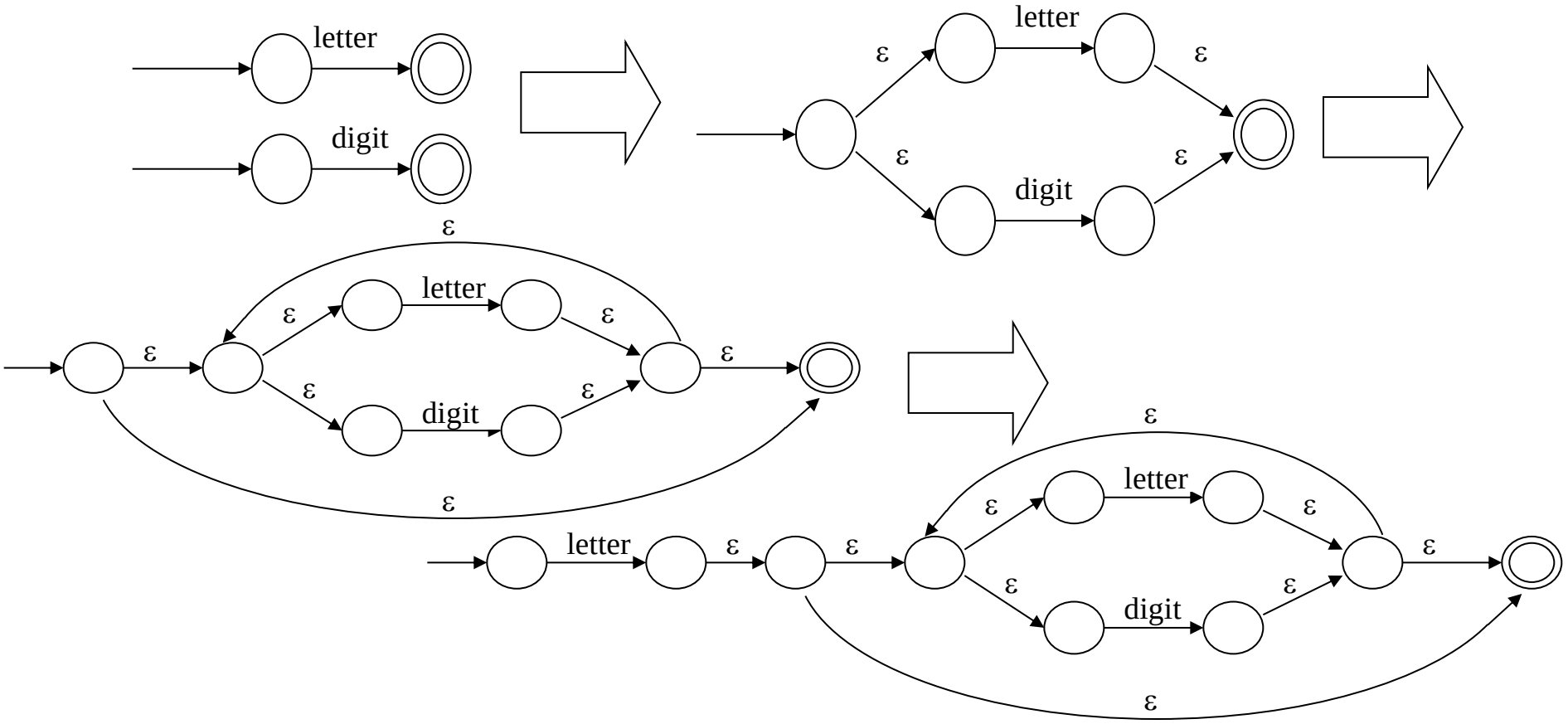
Examples of NFAs Construction

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



Examples of NFAs Construction

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



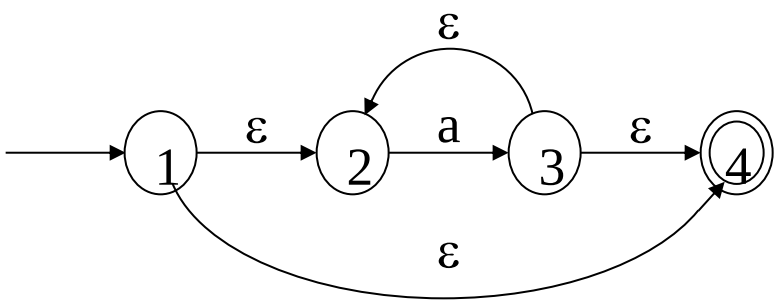
2.4.2 From an NFA to a DFA

Goal and Methods

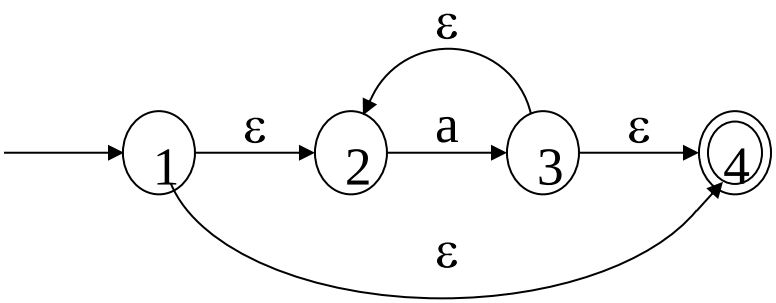
- **Goal**
  - ***Given an arbitrary NFA, construct an equivalent DFA (i.e., one that accepts precisely the same strings)***
- Some methods
  - (1) Eliminating ε-transitions
    - **ε-closure**: the set of all states reachable by ε-transitions from a state or states
  - (2) Eliminating multiple transitions 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.

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 expression **a***



The algorithm called **subset construction**.



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

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

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

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

The *Subset Construction* Algorithm

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

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

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

Compute the set

$S'_a = \{\ t \mid$ for some s in S there is a transition from s to t on $a\ \}$

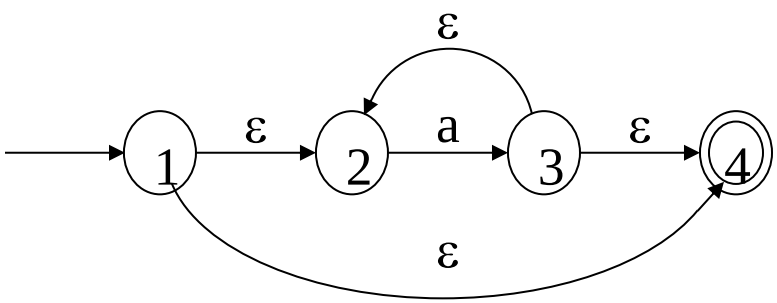
Then, **compute** $\overline{S_a}'$**, the ε-closure of** S'_a**.**

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

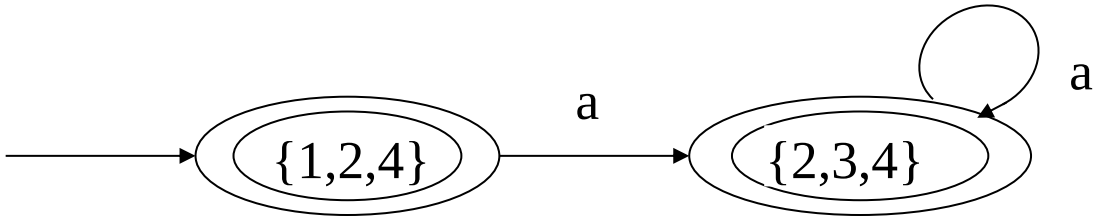
(3) Continue with this process until no new states or transitions are created.

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

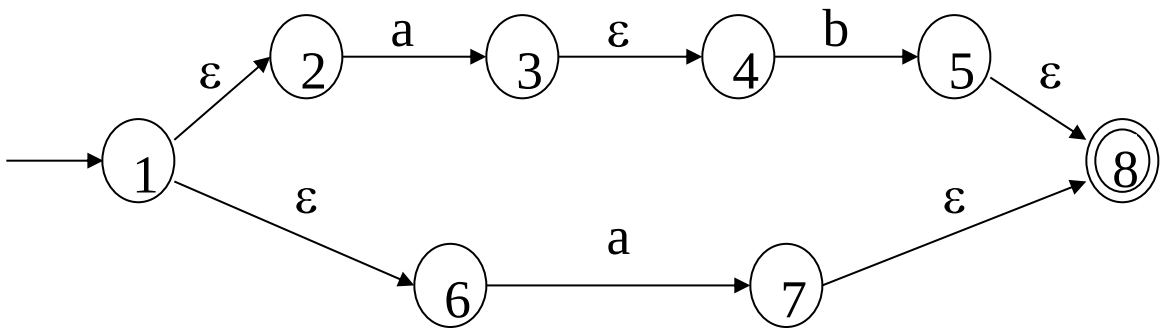
Examples of Subset Construction



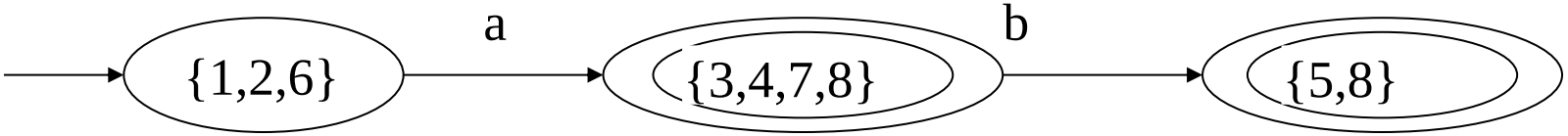
S	ε-closure of S	S'_a
1	1,2,4	3
3	2,3,4	3



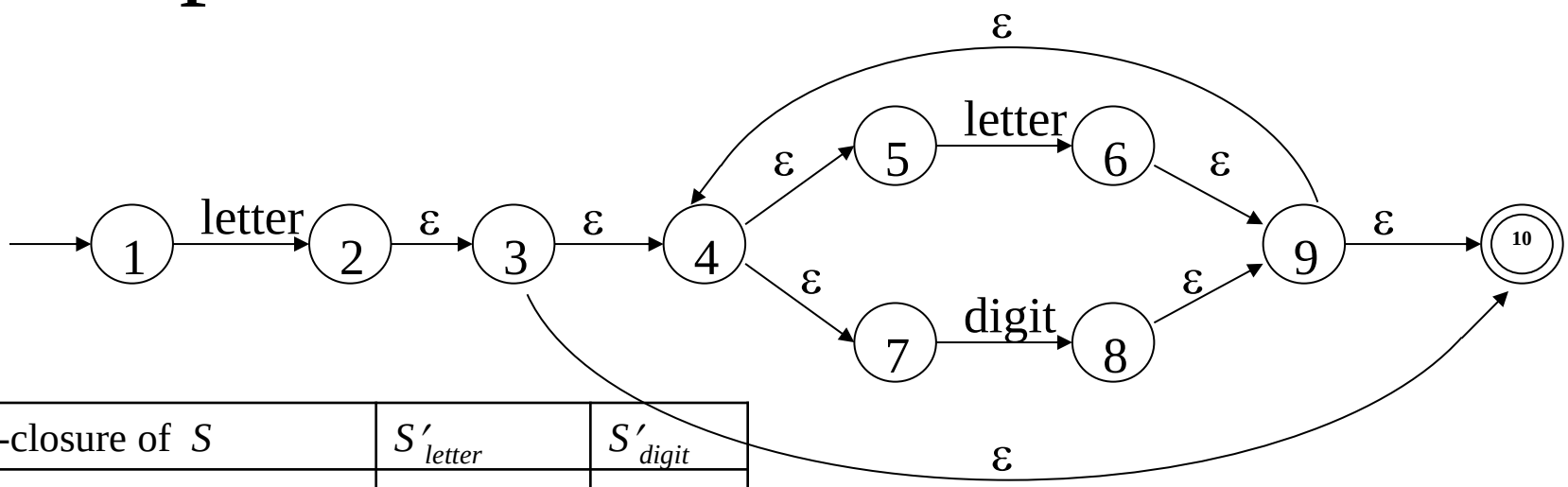
Examples of Subset Construction



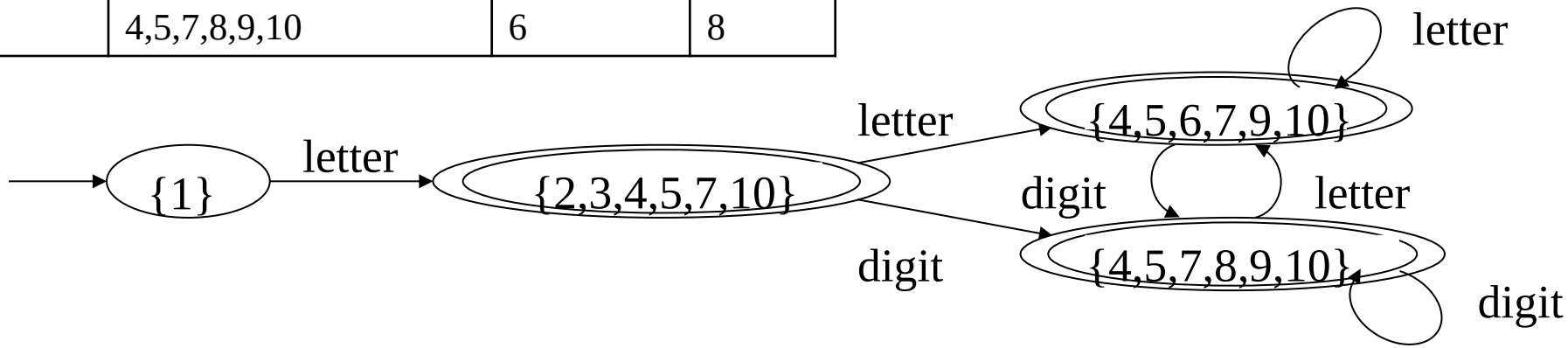
S	ε-closure of S	S'_a	S'_b
1	1,2,6	3,7	
3,7	3,4,7,8		5
5	5,8		



Examples of Subset Construction



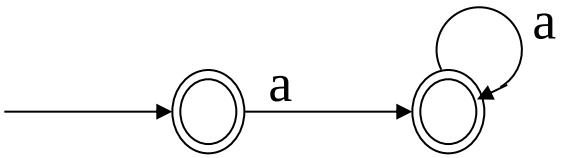
S	ε-closure of S	S'_{letter}	S'_{digit}
1	1	2	
2	2,3,4,5,7,10	6	8
6	4,5,6,7,9,10	6	8
8	4,5,7,8,9,10	6	8

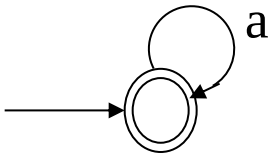


2.4.4 Minimizing the Number of States in a DFA

Why need Minimizing ?

- The process of deriving a DFA algorithmically from a regular expression has the unfortunate property that
  - the resulting DFA may be **more complex than necessary**.
- Example: the derived DFA (left) for the regular expression **a*** and an equivalent DFA (right)





An Important Result from Automata Theory for Minimizing

- **Given any DFA, there is an equivalent DFA containing a minimum number of states, and that this minimum-state DFA is unique** (except for renaming of states)

- It is also possible to directly obtain this minimum-state DFA from any given DFA.

Algorithm obtaining Mini-States DFA

1. Create two sets: one consisting of **all the accepting states** and the other consisting of **all the non-accepting states.**

2. Given this partition of the states of the original DFA, consider the transitions on **each character *a* of the alphabet**.

   (1) **If all accepting states have transitions on *a* to accepting states**, then this defines an *a*-transition from the new accepting state (the set of all the old accepting states) to itself.

   (2) **If all accepting states have transitions on *a* to non-accepting states**, then this defines an *a*-transition from the new accepting state to the new non-accepting state (the set of all the old non-accepting states).

Algorithm obtaining Mini-States DFA

(3) If there are two accepting states *s* and *t* that have transitions on *a* that land in **different sets**, then no *a*-transition can be defined for this group. Then ***a* distinguishes *s* and *t*.**

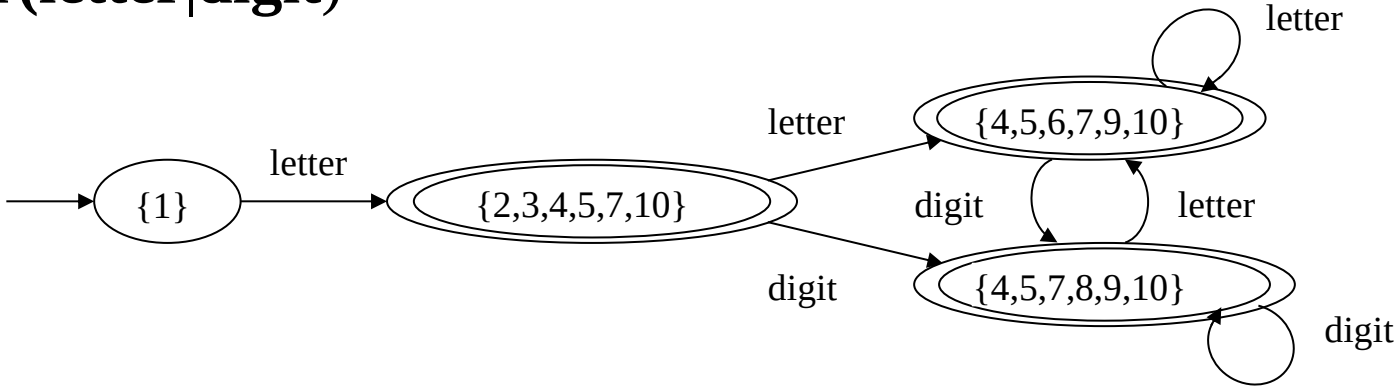
(4) If there are accepting states *s* and *t* such that *s* has an *a*-transition to another accepting state, while *t* has no *a*-transition at all (i.e., an error transition), then ***a* distinguishes *s* and *t*.**

3. **If any further sets are split, we must return and repeat the process from the beginning.** This process continues until either all sets contain only one element or until no further splitting of sets occurs.

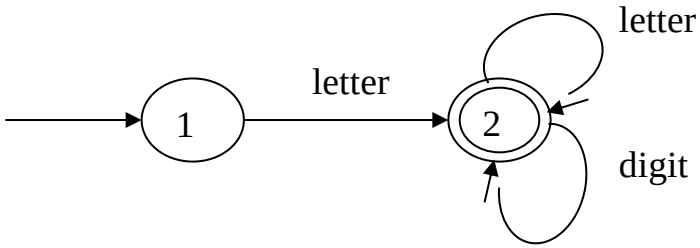
Examples of Minimizing DFA

Example 2.18: The regular expression

letter(letter|digit)*



The accepting set	{2,3,4,5,7,10},{4,5,6,7,9,10},{4,5,7,8,9,10}
The non-accepting set	{1}



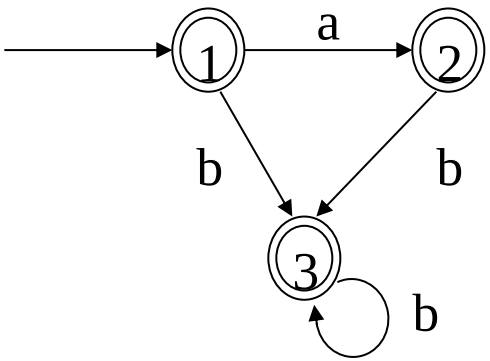
Examples of Minimizing DFA

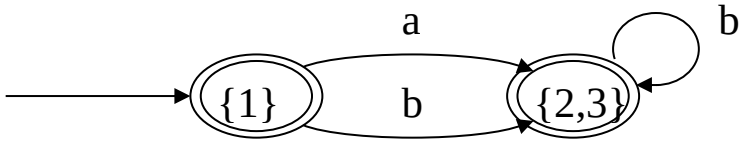
Example 2.18: the regular expression (**a**|**ε**)**b***

a distinguishes state 1 from states 2 and 3,

and we must repartition the states into the sets {1} and {2,3}

The accepting sets	{1,2,3}
The non-accepting sets	∅





Homework

1. Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$. Hint: some of these languages may include ε.

   (a) The set of all strings that do not contain the substring 00.

   (b) The set of all strings that contain at least three 1s.

   (c) The set of strings where all characters must appear in consecutive pairs (i.e. 00 or 11). Examples of strings in the language: ε, 000011, and 11. Examples of strings not in the language: 11$\underline{1}$00, 00$\underline{1}$00, and 11000$\underline{0}$).

↵

2. Convert your regular expression from (1a) to a DFA.↵

↵