

# COMPILER CONSTRUCTION

## Scanning

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017



# Chapter 3

# Theory and Practice of Scanning



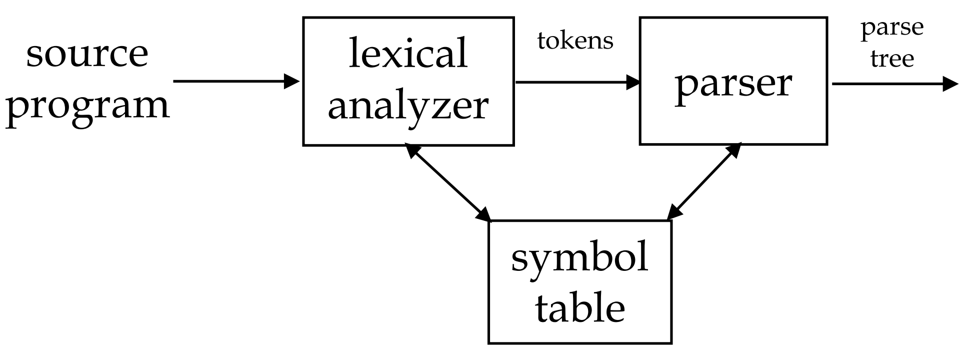
# Scanner Generator

- All scanners perform much the same function
  - except for the **tokens** to be recognized
- A **scanner generator** helps eliminate the redundant effort of building a scanner
  - One can simply specifying which tokens the scanner is to recognize, and leaves the rest to the generator
  - E.g., Lex, Flex
- Programming a scanner generator is an example of **declarative programming**
  - we do not tell a scanner generator **how** (procedural programming) to scan but simply **what** to scan

# Overview of a Scanner

- A scanner transforms a **character stream** of source program into a **token stream**
  - Also called a **lexical analyzer** or **lexer**

- Reasons to separate lexical analysis
  - Simpler design is the most import consideration
  - Compiler efficiency is improved
  - Compiler portability is enhanced





# Definition of Tokens

- **Formal definitions of tokens** helps:
  - avoid writing lexical analysers (scanners) by hand; possible errors
  - simplify **specification** and implementation
  - understand the underlying techniques and technologies
- For example, virtually all languages specify certain kinds of **rational constants**, which are often specified using decimal numerals such as 0.1 and 10.01
- Can .1 or 10. be allowed?

→C, C++, Java say **YES**

→But, Pascal and Ada say **NO**

Because *1..10* should be interpreted as a **range specifier** (1 to 10) in Pascal and Ada

If .1 and 10. are allowed, then we'll have **two constants** (i.e., 1. and .10), which will lead to immediate **grammar error** after the ``two constants''



# Regular Expression (RE)

- RE
  - Is a way to specify various sets of strings
  - Can specify the structure of the tokens used in a programming language

- In this course, we simply walk through the basic RE concepts required to build a scanner
  - as it should be covered thoroughly in Computing Theory



# Regular Expression (Cont'd)

- Regular set
  - A set of strings defined by a regular expression
- Token class
  - A regular set whose structure is defined by a RE
- Lexeme
  - A particular instance of a token class; or, instance of the token

| Token class | Sample Lexemes | Informal Description of Pattern |
|---|---|---|
| **const** | const | const |
| **if** | if | if |
| **relop** | <, <=, =, <>, >, >= | < or <= or = or <> or > or >= |
| **id** | pi, count, D2 | letter followed by letters and digits |
| **num** | 0, 3.14, 6.02E23 | any number constant |



# Regular Expression (Cont.)

- **Vocabulary**, denoted Σ
  - A finite character set
  - Normally the character set used by a computer, e.g., the **ASCII character set**, which contains **128 characters**
  - Java, however, uses the Unicode character set, including ASCII characters and a wide variety of other characters
- **Empty (or null) string**, denoted λ
  - represents an empty buffer in which no characters have yet been matched
  - represents also an optional part of a token
    - E.g., an integer literal may begin with a plus or minus; or, if it is unsigned, it may begin with λ

# Regular Expression (Meta-character)

- **Meta-character**
  - Any punctuation character or regular expression operator
  - When used as an ordinary character, **a meta-character must be quoted to avoid ambiguity**
- The six symbols are meta-characters
  **( ) ′ * + |**
  - NOTE: the symbol **′** is the Apostrophe character as used in *don′t*

- The four single-character tokens are often used in a programming language
  - ‘(’ | ‘)’ | ‘;’ | ‘,’
    (left parenthesis, right parenthesis, semicolon, and comma)
  - Parentheses are used to delimit expressions

# Regular Expression (Operations)

- Large (or infinite) sets are conveniently represented by **operations** on finite sets of characters and strings

- The three essential operations for RE

  Let $r_1$, $r_2$ be REs

  1. **Catenation**. $r_1r_2$
  2. **Alternation**. $r_1$ | $r_2$ denotes either $r_1$ or $r_2$
  3. **Kleene closure**. $r_1$* denotes zero or more occurrences of $r_1$

  | and * are the operators for Alternation and Kleene Closure, respectively



# Regular Expression (Catenation)

- **Catenation**, an operation
  - Strings are built from characters in the character set Σ via catenation

- As **characters** are catenated to a string, it grows in length
  - For example, the string `*do*' is built by first catenating *d* to λ and then catenating *o* to the string *d*
  - The null string λ, when catenated with any string s, yields s
  - That is, $s\lambda \equiv \lambda s \equiv s$

# Regular Expression (Catenation) (Cont'd)

- Catenation can be extended to **sets of strings**
  - Small finite sets are conveniently represented by listing their elements, which can be individual characters or strings of characters

- Let P and Q be sets of strings, and the symbol **∈** represents **set membership**
  - E.g., if s1 ∈ P and s2 ∈ Q, then string s1s2 ∈ (P Q)

# Regular Expression (Alternation)

- **Alternation** “ | ” can be extended to sets of strings
- Examples

1. Let P and Q be sets of strings
   The string s ∈ (P|Q) if, and only if, s ∈ P **or** s ∈ Q
2. Let D be the set of the ten single digits
   D is defined as D= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
   - Note that the textbook often uses abbreviations such **as (0 | . . . | 9)** rather than enumerate a complete list of alternatives
   - The . . . symbol is not part of the regular expression notation

# Regular Expression (Kleene Closure)

- **Kleene closure**
  - ``*'' postfix operator, representing zero or more occurrences of the *decorated* character/string
- Examples
  1. Let P be a set of strings
     P represents all strings formed by the catenation of zero or more selections (possibly repeated) from P
     Zero selections are represented by λ
  2. LC* is the set of all words composed only of **lowercase letters** and of **any length** (including the zero-length word, λ)

# Regular Expression (More Examples)

- Let Σ= {a, b}
  - The regular expression ***a* | *b*** denotes {a, b}
  - **(a | b)(a | b)** denotes {aa, ab, ba, bb}
    - i.e., the set of all 2-letter strings of a's and b's
    - Another form is *aa* | *ab* | *ba* | *bb*
  - $\boldsymbol{a}^*$ denotes the set of all strings of a's
    - i.e. {λ, a, aa, aaa, …}
  - $\boldsymbol{(a \mid b)}^*$ denotes the set of all strings of a's and b's
    - Another form is $(a^*b^*)^*$
  - ***a* | *a**b*** denotes the set containing the string a and all strings consisting of zero or more a's and followed by b's



# Formal Definition

- ∅ is a regular expression denoting the **empty set** (the set containing no strings)
- λ is a regular expression denoting the set that contains only the **empty string**
- The symbol s (s ∈ Σ) is a regular expression denoting **{s}**,
  - which refers to a **set** containing the single symbol
- If A and B are regular expressions
  then **A | B, AB, and A*** are also regular expressions
  They denote **3 operators**:

  1) **alternation**, 2) **catenation**, and 3) **Kleene closure** of the corresponding regular sets, respectively

# Formal Definition (Cont'd)

- **Additional operators**:
  - P+, sometimes called **positive closure** denotes all strings consisting of **one or more strings** in P catenated together: P*= (P+ | λ) and P+ = PP*
  - If A is a set of characters, **Not(**A**)**, denotes (**Σ - A**), all characters in Σ, but not in A
  - If *k* is a constant, then the set $A^k$ represents all strings formed by catenating k (possibly different) strings from A, i.e., $A^k$ = (AAA …), k copies of A



# Regular Expression (Another Example)

- A Java or C++ single-line comment that begins with **//** and ends with **Eol**
  - →Comment = / / (Not(**Eol**))***Eol**
  - The above RE represents that a comment **begins with two slashes** and **ends at the first end-of-line**
  - Within the comment, any sequence of characters is allowed that does not contain an end-of-line
  - This guarantees that the first end-of-line we see ends the comment

# Regular Expression (Extra Example)

- A fixed-decimal literal (for example, 23.456) can be defined as

  →Lit = D+.D+

  - **One or more digits** must be on both sides of the decimal point, so .12 and 35. are excluded

- Note: All **finite** sets are regular
  - However, some (but not all) infinite sets are regular



# Beyond Regular Expression

- All regular sets can be defined by Context-Free Grammars (CFGs)
- CFGs are more powerful descriptive mechanism than regular expressions
  - Res are quite adequate for specifying **token-level** syntax
- For every regular expression we can create an efficient device, **finite automaton**,
  - which recognizes exactly those strings that match the regular expression's pattern



# Finite Automata and Scanners

- A **finite automation** (FA) can be used to help **recognize** the tokens specified by a regular expression
  - To see if the given strings are belonging to the regular sets

- An FA consists of:
  - A finite set of ***states***
  - A finite ***vocabulary***, denoted Σ
  - A set of ***transitions*** (or moves) from one state to another, labeled with characters in Σ
  - A special state called the ***start*** state
  - A subset of the states called the ***accepting***, or *final*, states.

- An FA can also be represented graphically using a transition diagram, composed of the components shown in Fig. 3.1



# A Finite Automaton

- An FA also can be represented graphically using a **transition diagram**
  - In practice, we first build the corresponding FA based on the given RE
  - Then, the built FA is used to determine if the given string is a valid token
- We now build the FA for the RE: (abc+)+

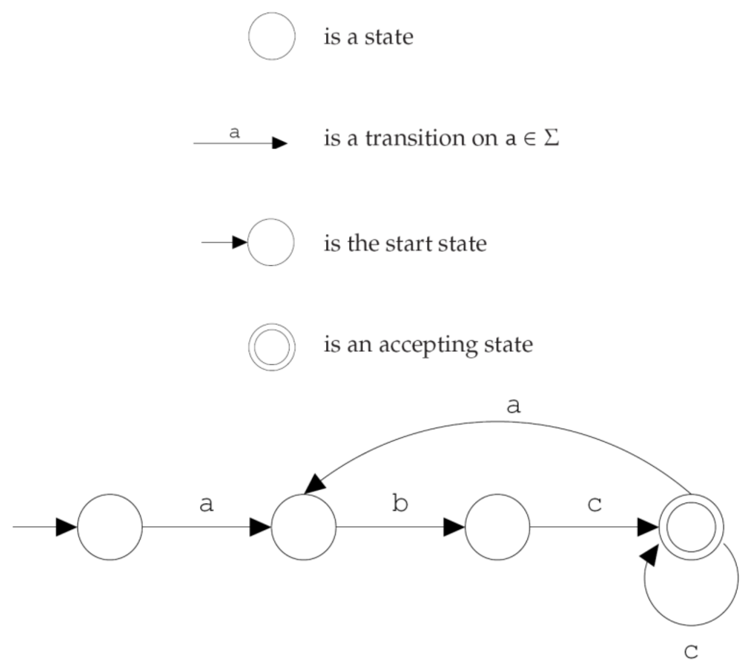
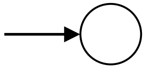



Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes $(a\,b\,c^{+})^{+}$.

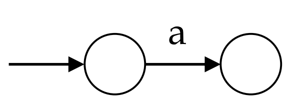
# Example: Build A Finite Automaton (Cont'd)

- Regular expression: (abc+)+
- The character to be read:
  - **None**
- Graph:
  - Add the starting state



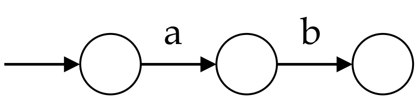
# Example: Build A Finite Automaton (Cont'd)

- Regular expression: (abc+)+
- The character to be read:
  - (a
- Graph:



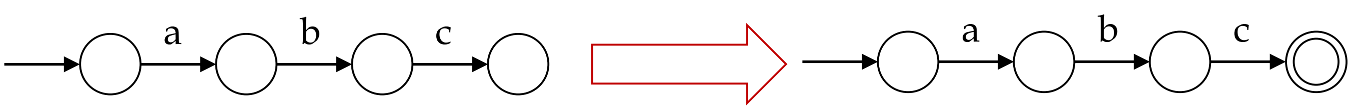
# Example: Build A Finite Automaton (Cont'd)

- Regular expression: (abc+)+
- The character to be read:
  - (ab
- Graph:



# Example: Build A Finite Automaton (Cont'd)

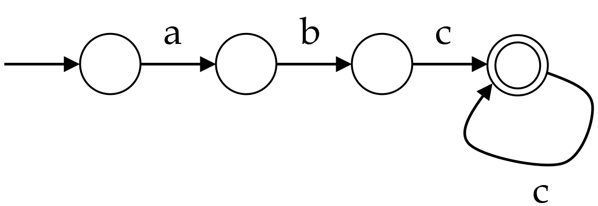
- Regular expression: (abc+)+
- The character to be read:
  - (abc
- Graph:




→ It looks like the string, ``**abc**'', is accepted by the regular expression, so we mark the current state as the **accepting state**

# Example: Build A Finite Automaton (Cont'd)

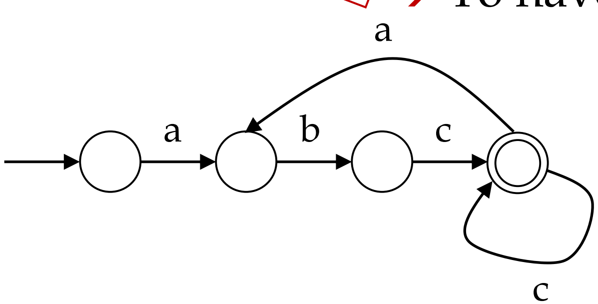
- Regular expression: (abc+)+
- The character to be read:
  - (abc+
- Graph:



# Example: Build A Finite Automaton (Cont'd)

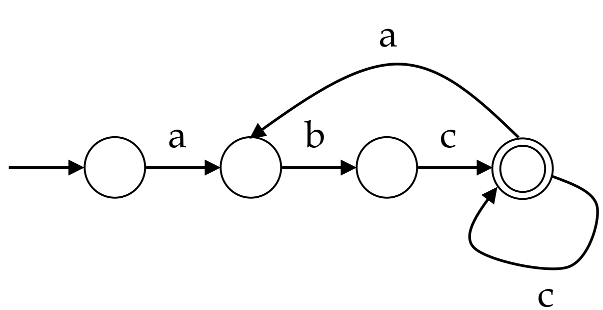
- Regular expression: (abc+)+
- The character to be read:
  - (abc+)+
- Graph:

→ To have more ``(abc+)'' string, we add the transition on ***a***



# Example: Build A Finite Automaton (Cont'd)

- Regular expression: (abc+)+
- The character to be read:
  - (abc+)+
- Graph:





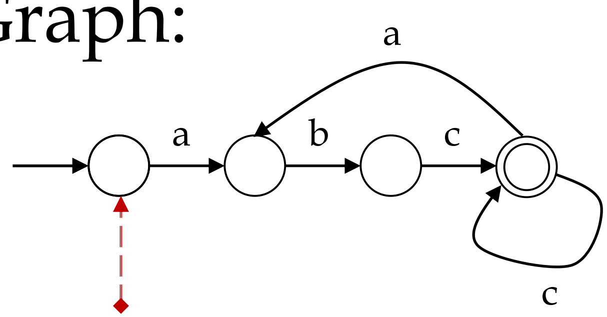
# Example: Recognize the Token

- Regular expression: (abc+)+
  1. It starts from the **start state**
  2. Does the next input character match the label on a transition from the current state?
     Yes, we move to the state pointed by the next character and set the next character as current character; otherwise, we stop
     This step runs iteratively until no next character or no suitable transition
  3. If we finish in an **accepting state**, the sequence of the characters forms a valid token; otherwise, a valid token has not been seen

# Example: Recognize the Token (Cont'd)

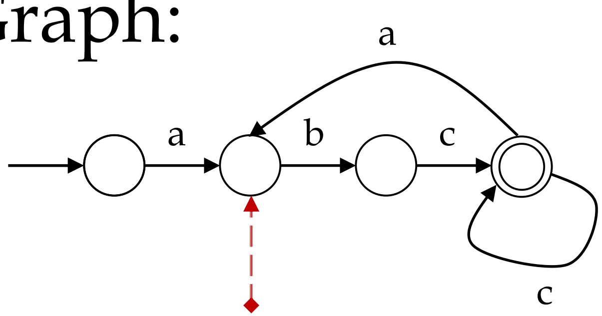
- Regular expression: (abc+)+
- **The given string:** **abc**
- The character to be read:
  - **None** (at starting state)

- Graph:



# Example: Recognize the Token (Cont'd)

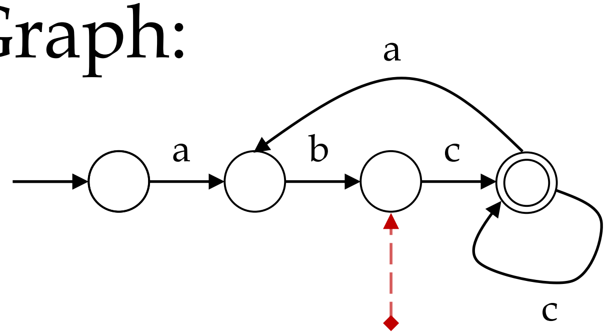
- Regular expression: (abc+)+
- **The given string: abc**
- The character to be read:
  - a
- Graph:



# Example: Recognize the Token (Cont'd)

- Regular expression: (abc+)+
- **The given string: abc**
- The character to be read:
  - ab

- Graph:

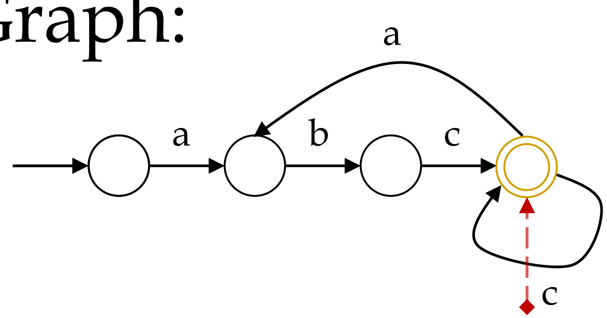


# Example: Recognize the Token (Cont'd)

- Regular expression: (abc+)+
- **The given string: abc**
- The character to be read:
  - abc

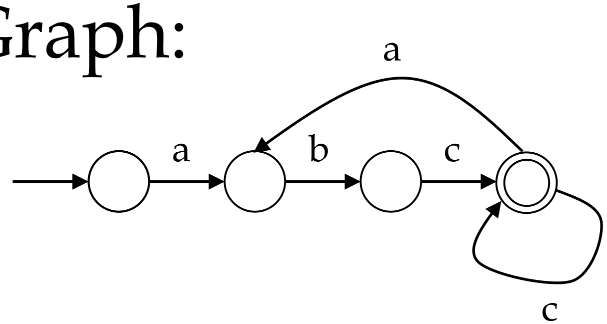
→ We have read out the characters and are at the **accepting state**, so we say the given string is a valid token.

- Graph:



# Example: Recognize the Token (Cont'd)

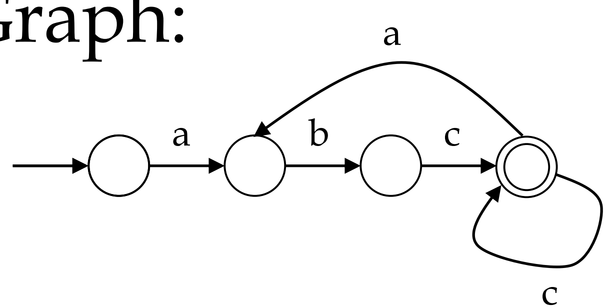
- Regular expression: (abc+)+
- **The given string:** abc
- The character to be read:
  - **None** (at starting state)
- Graph:



# Example: Recognize the Token (Cont'd)

- Regular expression: (abc+)+
- **The given string:** **abc**
- The character to be read:
  - **None** (at starting state)

- Graph:





# Deterministic Finite Automata (DFA)

- A DFA is an FA that always allows **a unique transition for a given state and character**
- DFA is simple to program and are often used to drive a scanner
- DFA is conveniently represented in a computer by a **transition table**

# Transition Table for DFA

- A transition table, *T*, is a two-dimensional array
  - indexed by a **DFA state** and a **vocabulary symbol**
  - Entries are either a **DFA state** or an **error flag** (often represented as a blank table entry)

Example

- If we are in state s and read character c,
  - then T[s, c] will be **the next state** we visit,
  - or T[s, c] will contain **an error flag** indicating that c cannot extend the current token



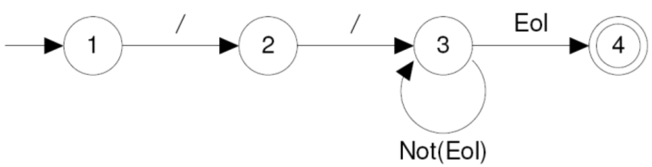
# Example: DFA for the Single-Line Comment

- The single-line comment for Java or C++ can be represented by the regular expression

  →// (Not (**EoI**) )* **EoI**

  - Fig. 3.2 is the corresponding DFA. (a) transition graph, & (b) transition table

(a)




(b)

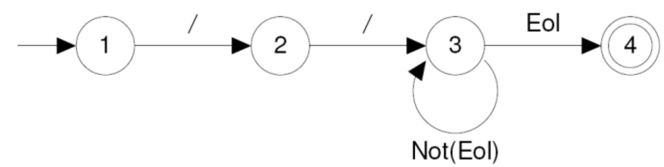
| State | Character | | | | |
|---|---|---|---|---|---|
| | / | EoI | a | b | . . . |
| 1 | 2 | | | | |
| 2 | 3 | | | | |
| 3 | 3 | 4 | 3 | 3 | 3 |
| 4 | | | | | |

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

# Example: DFA for the Single-Line Comment (Cont'd)

- T[1, /]
  - If we are at the **state 1** and
  - have the next input character `/'
  - The next state is **state 2**
    →T[1, /]=2
- T[3, d]
  - If we are at the **state 3** and
  - have the next input character `d'
  - The next state is **state 3**
    →T[3, d]=3
- T[3, **EoI**]
  - If we are at the **state 3** and
  - have the next input character **EoI**
  - The next state is **state 4**
    →T[3, **EoI**]=4

(a)




T[]

(b)

| State | Character | | | | |
|---|---|---|---|---|---|
| | / | EoI | a | b | . . . |
| 1 | 2 | | | | |
| 2 | 3 | | | | |
| 3 | 3 | 4 | 3 | 3 | 3 |
| 4 | | | | | |

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.



# Coding the DFA

- A DFA can be coded in one of two forms:
  - Table-driven
  - Explicit control
- The ***table-driven*** form (as shown in Fig. 3.3)
  - the transition table that defines the **transitions** in a DFA are **explicitly represented** in a runtime table that is “interpreted” by a driver program

```
/*    Assume CurrentChar contains the first character to be scanned    */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then   /* Return or process the valid token */
else   /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.



# Coding the DFA (Explicit Control)

- The *explicit control* form (as shown in Fig. 3.4)
  - the transition table that defines a DFA's **actions appears implicitly** as **the control logic** of the program

```
/*   Assume CurrentChar contains the first character to be scanned   */
if CurrentChar = '/'
then
    CurrentChar ← READ( )
    if CurrentChar = '/'
    then
        repeat
            CurrentChar ← READ( )
        until CurrentChar ∈ { Eol, Eof }
    else   /* Signal a lexical error */
else   /* Signal a lexical error */
if CurrentChar = Eol
then   /* Finished recognizing a comment */
else   /* Signal a lexical error */
```

Figure 3.4: Explicit control scanner.



# About Lex Scanner Implementation

- **Sec. 3.5 Lex Scanner Generator**
  - will be covered when we announce the programming homework #1
  - You could read it first by yourself
- **Sec. 3.6 Other Scanner Generators**
  - Is optional
- **Sec. 3.7 Practical Considerations of Building** Scanners
  - should be read if you want to implement a more complex and efficient scanner

# Regular Expression and Finite Automata

- REs are equivalent to FAs
- The main job of **scanner** is to
  - **transform a RE into an equivalent FA**

- To make an FA from a regular expression proceeds in two steps:
  1. It transforms the regular expression into an NFA
  2. It transforms the NFA into a DFA

RE → NFA → DFA

# Nondeterministic Finite Automaton (NFA)

- An NFA needs not make **a unique** (deterministic) **choice** of which state to visit
  - when reading a particular input
- In other words
  - FAs that contain no λ transitions and that always have unique successor states for any symbol are **deterministic**

# Example: NFAs

Fig. 3.17

–The NFA below is allowed to have a state that has two transitions coming out of it, label by the same symbol a

Fig. 3.18

–The NFA may also have transitions labeled with λ

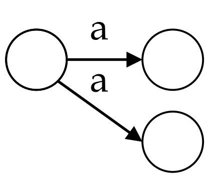



Figure 3.17: An NFA with two a transitions.

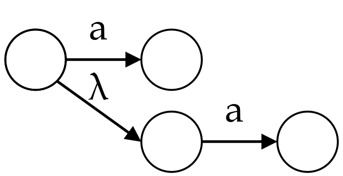



Figure 3.18: An NFA with a λ transition.

# Key Elements of REs

- Recall that a regular expression is built of:
  - the ***atomic*** regular expressions
    ***a*** (a character in Σ) and **λ** (shown in Fig. 3.19)
  - with the **three operations**:
    AB, A | B, and A*
  - Other operations, such as A+, are just abbreviation for the combinations for the above operations

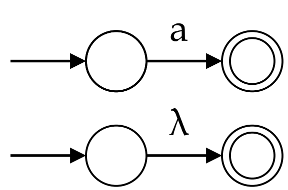



Figure 3.19: NFAs for a and λ.



# Transforming RE to NFA

- Suppose we have FAs for A and B
  - The NFA for **A | B** (shown in Fig. 3.20)
    - The states labeled A and B were the accepting states of the automata for A and B
    - we create a new accepting state for the combined FA
  - The NFA for **AB** (shown in Fig. 3.21)
    - The accepting state of the combined FA is the same as the accepting state of B
  - The NFA for **A*** is (shown in Fig. 3.22)
    - The start state is an accepting state, so λ is accepted

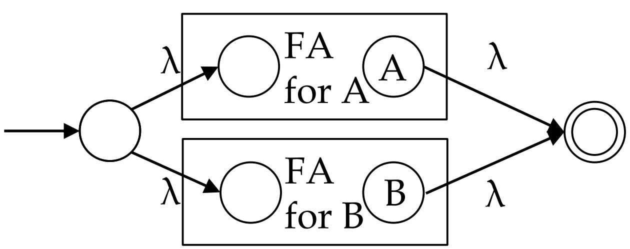



Figure 3.20: An NFA for A | B.

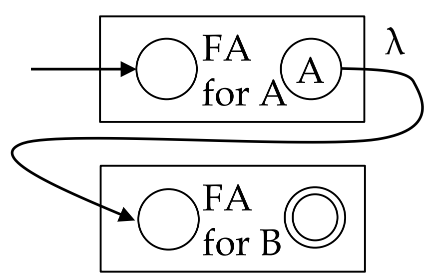



Figure 3.21: An NFA for AB.

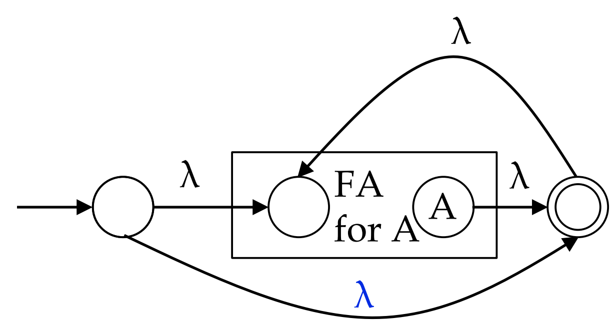



Figure 3.22: An NFA for A*.



# Example: RE to NFA

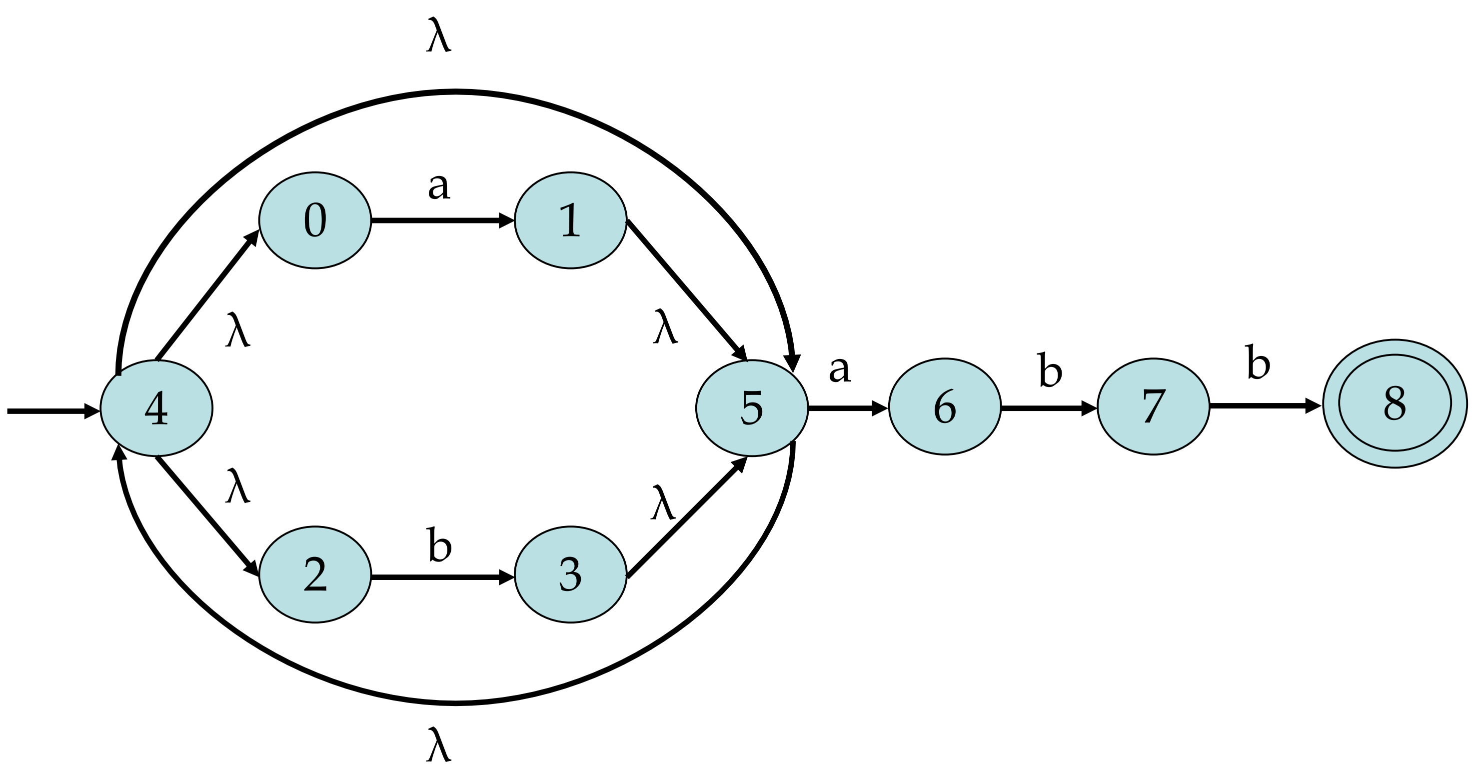
- For regular expression: **(a | b)*abb**

1. We create the NFA for a, b, a | b, (a | b)*
2. We create NFA for "abb"

# Example: RE to NFA (Cont'd)

- Regular expression: **(a | b)*abb**

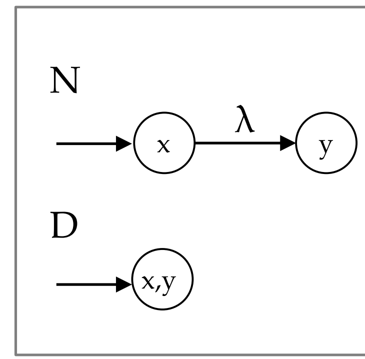
1. NFA for a, b, a | b, (a | b)*
2. NFA for "abb"
3. NFA for (a | b)*abb



# Transforming NFA to DFA

- **Subset construction algorithm**
  - transforming an **NFA N** to an equivalent **DFA D**
  - given in Fig. 3.23

Example of N and D




- Key idea
  - To construct **each state of *D*** with a ***subset* of states of *N***
  - D will be in **the state {x,y,z}** after reading a given input character, if and only if N could be in ***any* of the states x,y,or z**

# Notion

N: NFA (non-deterministic finite automata)

D: DFA (deterministic finite automata)

$s \xrightarrow{c} t$: In N under char c, state s transits to t

$S \xrightarrow{c} T$: In D, under char c, state S transits to T

S is a subset of {s | s in N}



# Start State

- The **start state of D**
  - is the set of all states to which N can transition without reading any input characters
  - that is, **the set of states reachable from the start state of N following only λ transitions**
- Algorithm **Close(S,T)**
  - computes those **states** that can be reached after only λ transitions, also known as **λ-successors**
  - shown in Fig. 3.23
- Once the start state of D is built, we begin to create successor states

# Successor States of the Start State

- Place each state S of D on a **work list** when it is created
  - For each state S on the work list and each character c in the vocabulary,
  - we compute **S's successor under c** **($S \xrightarrow{c} T$)**
    - S is identified with some set of N's states {n1, n2, . . .}
    - We find all of the possible successor states to {n1, n2, . . .} under **c** and obtain a set {m1,m2, . . .}
    - Finally, we include the **λ-successors** of {m1,m2, . . .}
  - The resulting set of NFA states is included as a state T in D, and a transition from S to T, labeled with c, is added to D
  - We continue adding states and transitions to D until all possible successors to existing states are added
    - Because each state corresponds to a finite subset of N's states, the process of adding new states to D must eventually terminate

Tip:
Step 1: Find **char successors**
Step 2: Fine **λ successors**



# Accepting State

- **An accepting state of D is any set that contains an accepting state of N**
- This reflects the convention that N accepts if there is any way it could get to its accepting state by choosing the "right" transitions

# Example: NFA to DFA

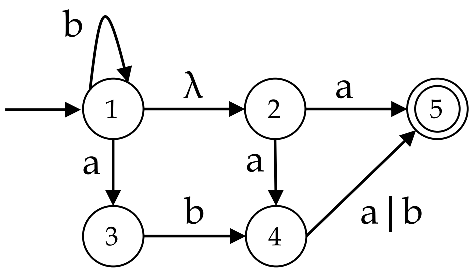



Figure 3.24: An NFA showing how subset construction operates.

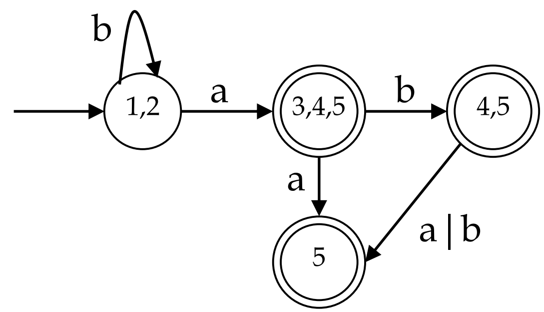



Figure 3.25: DFA created for NFA of Figure 3.24.

```
function makeDeterministic(N) returns DFA
❶ D.StartState ← recordState({N.StartState})      {1}
❷ foreach S ∈ WorkList do
       WorkList ← WorkList − {S}
❸      foreach c ∈ Σ do D.T(S, c) ← recordState(T ← ∪_{s∈S} t(s, c))
❹ D.AcceptStates ← {S ∈ D.States | S∩N.AcceptStates ≠ ∅}
end
                λ-successors
function close(S, T) return Set
    ans ← S                                        {1}
    repeat
        changed ← false
❺       foreach s ∈ ans do                         {2}
❻           foreach t ∈ T(s, λ) do
                if t ∉ ans
                then
❼                   ans ← ans ∪ {t}
                    changed ← true
    until not changed
    return (ans)
end
function recordState(S) return Set
❽ S ← close(S, T)                                  {1}
❾ If S ∉ D.States
    then
       D.States ← D.States ∪ {S}
       WorkList ← WorkList ∪ {S}
    return (S)
end
```

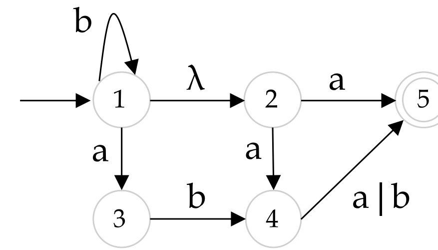



Figure 3.24: An NFA showing how subset construction operates.

**To construct the start state of DFA:**

**→ To find the set of states reachable from the start state of N following only λ transitions.**

- Start with state 1, the start state of N, and call RecordState(state 1) to find its λ-successors (Marker 1)
- RecordState() calls Close(state1, T). T includes states 2 and 3 (Marker 8), where T represents the following states of state 1 in N via any symbol.
- In Close(), set ans to state 1 (S).

  And then for state 1 in ans (Marker 5) find the successor of state 1 via λ-transition, t ∈ T(state 1, λ) (Marker 6). Add t to ans, which is state 2 (Marker 7).

  After that, return the ans set, states {1,2}, to RecordState() (Marker 8).
- Then, RecordState() will determine whether the set is in D.States (Marker 9). It is not, so it will be stored into D.States and WorkList (Marker 9).
- Now, we have constructed DFA 's start state as states {1,2}.

**Revised Fig. 3.23 Construction of a DFA $D$ from an NFA $N$**

```
function makeDeterministic(N) returns DFA
(1)  D.StartState ← recordState({N.StartState})
(2)  foreach S ∈ WorkList do
         WorkList ← WorkList − {S}
(3)      foreach c ∈ Σ do D.T(S, c) ← recordState(T ← ∪_{s∈S} t(s, c))
(4)  D.AcceptStates ← {S ∈ D.States | S∩N.AcceptStates ≠ ∅}
end
function close(S, T) return Set
    ans ← S
    repeat
        changed ← false
(5)     foreach s ∈ ans do
(6)         foreach t ∈ T(s, λ) do
                if t ∉ ans
                then
(7)                 ans ← ans ∪ {t}
                    changed ← true
    until not changed
    return (ans)
end
function recordState(S) return Set
(8) S ← close(S, T)
(9) If S ∉ D.States
    then
        D.States ← D.States ∪ {S}
        WorkList ← WorkList ∪ {S}
    return (S)
end
```

Find all of the possible successor states under c of N with the given states, including λ states.

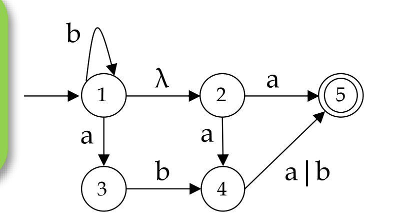



Figure 3.24: An NFA showing how subset construction operates.

Construct the successors of the start state S = {1, 2} of DFA:

for each S in WorkList (S = **{1, 2}**) do (Marker 2)

under char **"a"**

set S's successor D.T(S, c) (S is **{1, 2}** and c is ***a***) to (Marker 3):

state 1 transits to 3,

state 2 transits to 4,

state 2 transits to 5, we got T= **{3,4,5}**

recordStates (**{3,4,5}**)

add {3,4,5} to D.States and workList

under char **"b"**

set S's successor D.T(S, c) (S is **{1, 2}** and c is ***b***) to:

state 1 transits to 1, we got T=**{1}**

recordStates (**{1}**) calls close(), we got T=**{1,2}**

**{1,2}** is already in D.States, so do not add it to D.States and WorkList

**function** makeDeterministic($N$) **returns** *DFA*

❶ $D.StartState \leftarrow$ recordState($\{N.StartState\}$)

❷ **foreach** $S \in WorkList$ **do**

  $WorkList \leftarrow WorkList - \{S\}$

❸  foreach $c \in \Sigma$ **do** $D.T(S, c) \leftarrow$ recordState($T \leftarrow \bigcup_{s \in S} t(s,\ c)$)

❹ $D.AcceptStates \leftarrow \{S \in D.States \mid S \cap N.AcceptStates \neq \emptyset\}$

**end**

**function** close(S, T) **return** *Set*

  $ans \leftarrow S$

  **repeat**

   $changed \leftarrow$ **false**

❺   **foreach** $s \in ans$ **do**

❻    **foreach** $t \in T(s, \lambda)$ **do**

     **if** $t \notin ans$

     **then**

❼      $ans \leftarrow ans \cup \{t\}$

      $changed \leftarrow$ **true**

  **until not** *changed*

  **return** ($ans$)

**end**

**function** recordState($S$) **return** *Set*

❽ $S \leftarrow$ close($S$, $T$)

❾ **If** $S \notin D.States$

  **then**

   $D.States \leftarrow D.States \cup \{S\}$

   $WorkList \leftarrow WorkList \cup \{S\}$

  **return** ($S$)

**end**

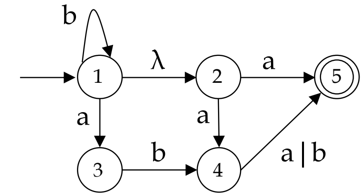



Figure 3.24: An NFA showing how subset construction operates.

Construct the successors of the start state S = {1, 2} of DFA:

for each S in WorkList (S = {1, 2}) do (Marker 2)

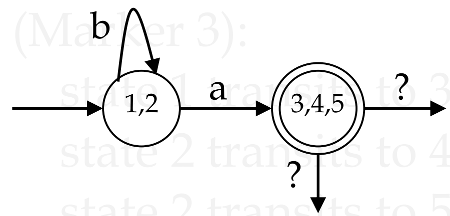
under char **"a"**

set S's successor D.T(S, c) (S is {1, 2} and c is *a*) to (Marker 3):

state 1 transits to 3,

state 2 transits to 4,

state 2 transits to 5, we got T= **{3,4,5}**




recordStates ({3,4,5})

What we have established so far.

**$\{1,2\} \xrightarrow{a} \{3,4,5\}$**

add {3,4,5} to D.States and wor

**D.T({1,2}, a)← {3,4,5}**

under char **"b"**

set S's successor D.T(S, c) (S is {1, 2} and c is *b*) to:

state 1 transits to 1, we got T={1}

recordStates ({1}) calls close(), we got T=**{1,2}**

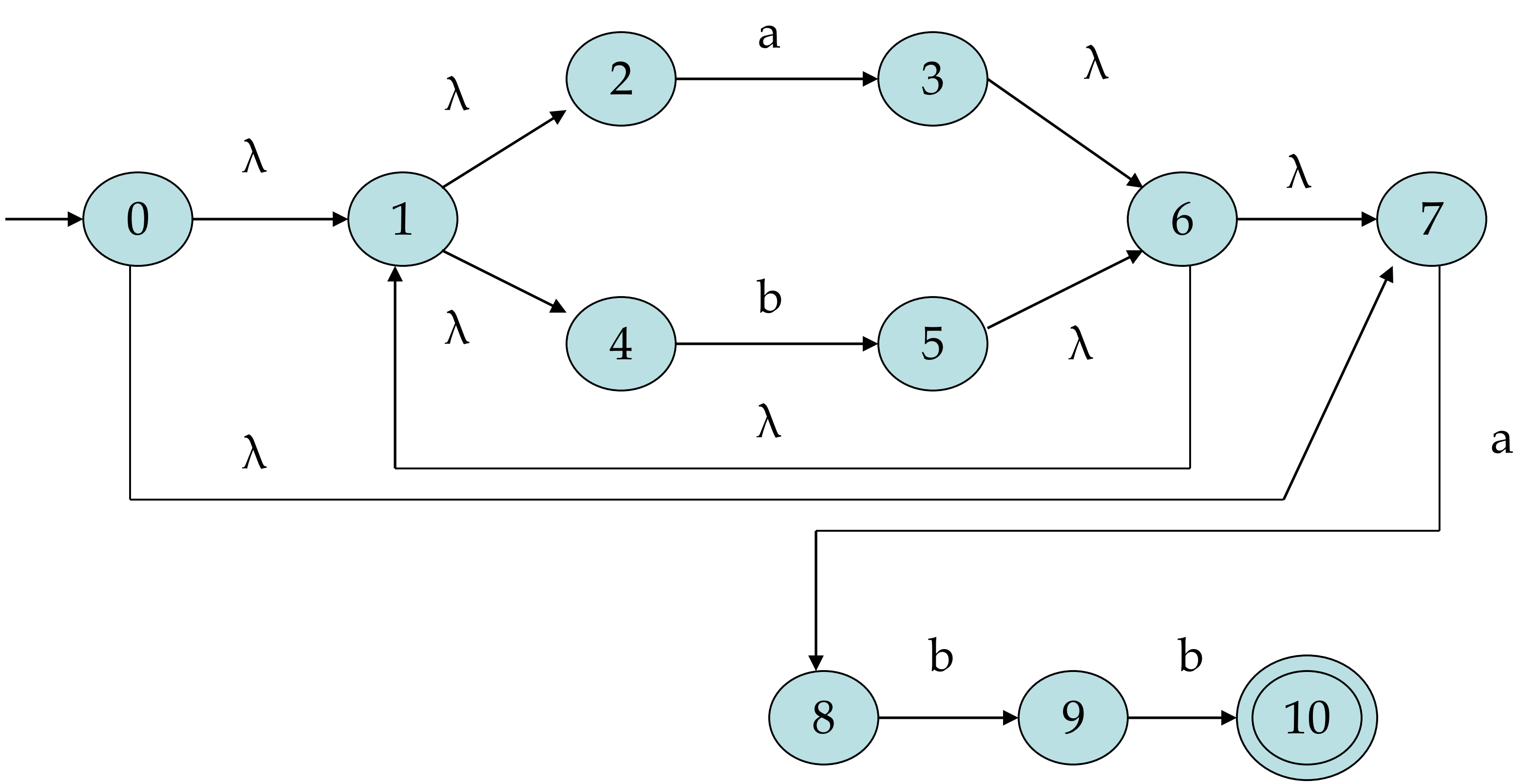
{1,2} is already in D.States, so do not add it to D.States and WorkList

**$\{1,2\} \xrightarrow{b} \{1,2\}$**

**D.T({1,2}, b)← {1,2}**

# Your Exercise

Given the NFA below, find its DFA.



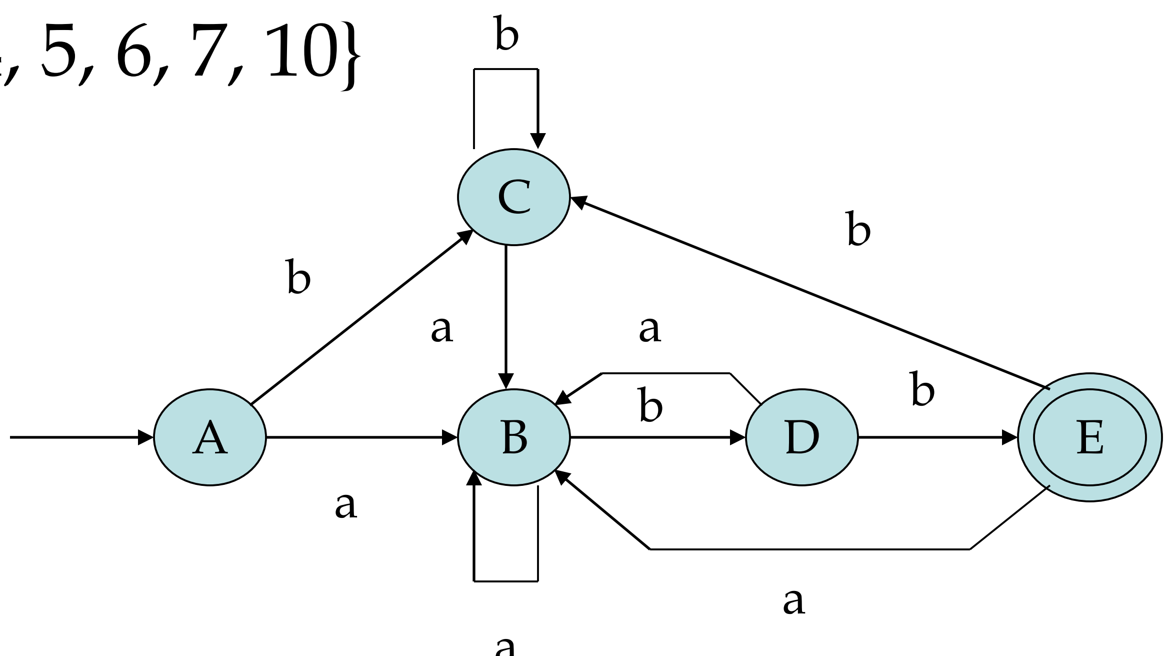
# Your Exercise (Cont'd)

The resulting DFA is:

A {0, 1, 2, 4, 7}    B {1, 2, 3, 4, 6, 7, 8}

C {1, 2, 4, 5, 6, 7}    D {1, 2, 4, 5, 6, 7, 9}

E {1, 2, 4, 5, 6, 7, 10}





# QUESTIONS?