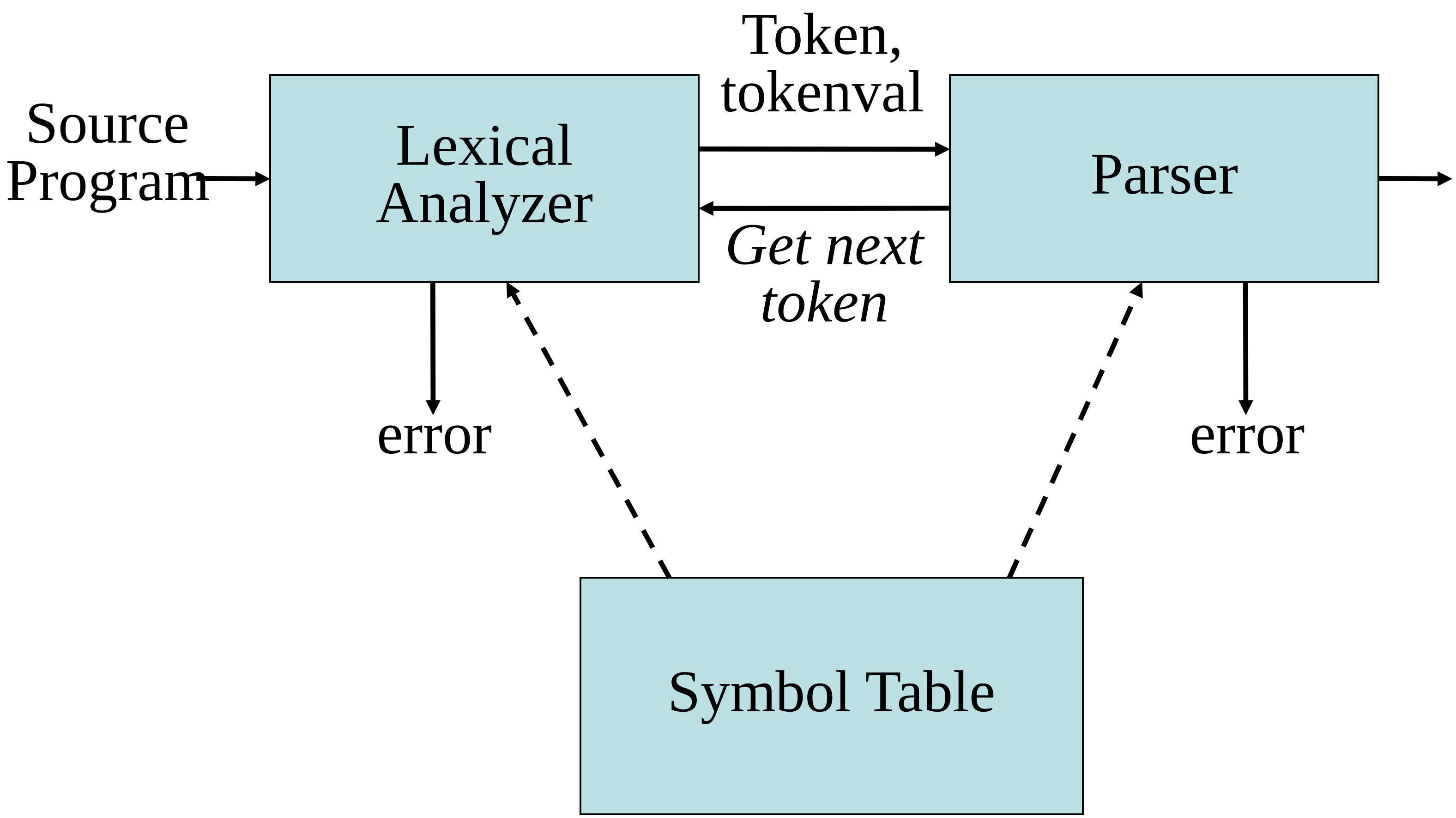
Lexical Analyzer (Scanner)

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Why to separate Lexical Analyzer And parsing

- Simplicity of design
- Improving compiler efficiency
- Enhancing compiler portability

Tokens, Lexemes, Patterns

- A *token* is a pair - a token name and an optional token value
  - For example: **id** and **num**

- *Lexemes* are sequence of characters in the source program that matches pattern for a token
  - For example: `abc` and `1233`
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: “*letter followed by letters and digits”* and “*non-empty sequence of digits*”

Example

Token	Informal description	Sample lexemes
if	Characters I ,f	If
Comparison	< , > , <= , >=	<= . !=
Number	Any numeric constant	3.14,0
literal	Anything but “ surrounded by”	“core dumped”
id	Letters followed by letters and digit	Pi,D2

Example

E = M *C**2

- <id, symbol table entry for E>
- < assign – op>
- <id, symbol table entry for M>
- < mult – op>
- <id, symbol table entry for C>
- <exp – op>
- <number , interger value 2 >

Lexical Errors

- Unterminated comments
- Length of longest identifier
- Invalid identifiers
- Invalid constants
- Invalid symbols

Lexical Errors

- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character

Specification of Token

- Any language has an alphabet which defines the characters or symbols that language can have.
- In theory of compilation regular expression are used to formalize the specification of tokens
- Example :
  - letter(letter | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - $|s|$ is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - $\varnothing$ the empty set is a language.
  - $\{\varepsilon\}$ the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$

- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

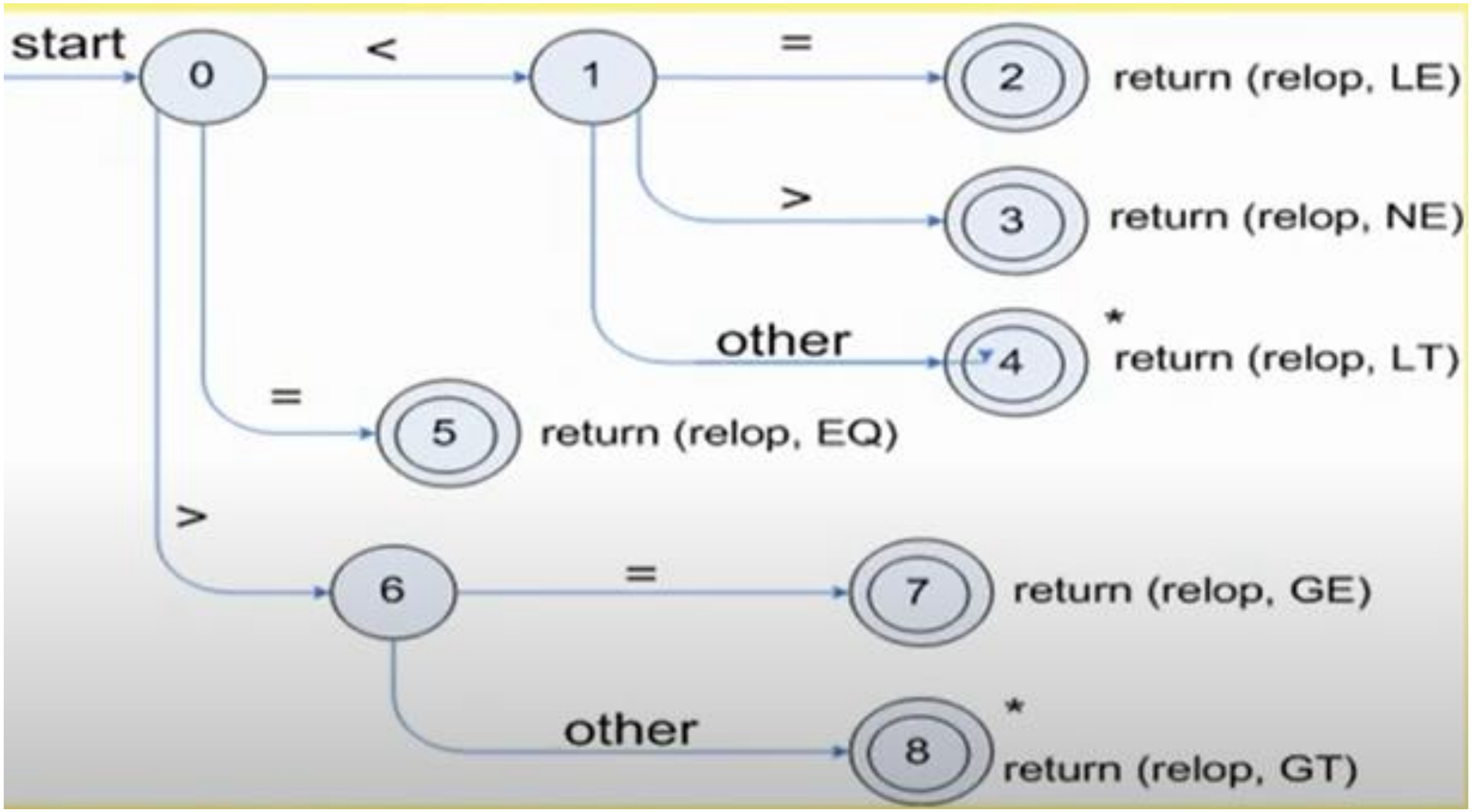
Regular Expressions (cont.)

- Ex:
  - $\Sigma = \{0,1\}$
  - 0|1 => {0,1}
  - (0|1)(0|1) => {00,01,10,11}
  - 0^* => {ε,0,00,000,0000,....}
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string
  - 0(0|1)*0 => All binary strings of length at least 2 , starting and ending with 0's
  - (0|1)*0(0|1)(0|1)(0|1) : All binary strings with at least three characters in which the 3rd last character is always 0

Recognition of tokens

- Digit - [0-9]
- Digits – digit+
- Letter – [A-Za-z_]
- Id – letter(letter|digit)*
- If – if
- Then – then
- Else – else
- Relop - < | > | <= | >= | = | < >

Recognition of tokens



Disambiguation Rules

1) **longest match rule:** from all tokens that match the input prefix, choose the one that matches the most characters
2) **rule priority:** if more than one token has the longest match, choose the one listed first

Examples:

- for8 — is it the for-keyword, the identifier "f", the identifier "fo", the identifier "for", or the identifier "for8"?
  Use rule 1: "for8" matches the most characters.
- for — is it the for-keyword, the identifier "f", the identifier "fo", or the identifier "for"?
  Use rule 1 & 2: the for-keyword and the "for" identifier have the longest match but the for-keyword is listed first.

How Scanner Generators Work

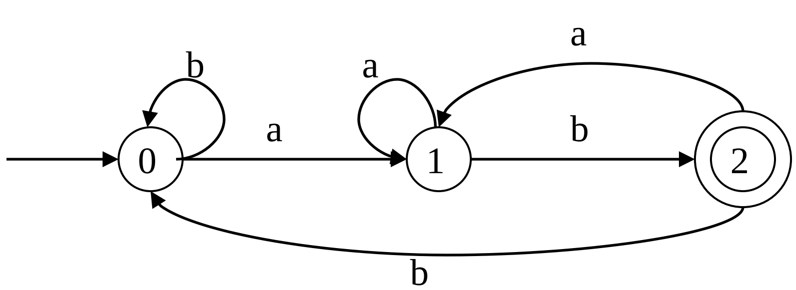
- Translate REs into a finite state machine
- Done in three steps:
  1) translate REs into a no-deterministic finite automaton (NFA)
  2) translate the NFA into a deterministic finite automaton (DFA)
  3) optimize the DFA (optional)

Finite Automata

- A *recognizer* for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
  - no state has ε- transition
  - for each symbol a and state s, there is at most one labeled edge a leaving s.
    i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by this DFA is also $(a|b)^*\ a\ b$

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F – a set of accepting states (final states)

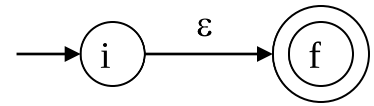
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Converting A Regular Expression into A NFA (Thomson's Construction)

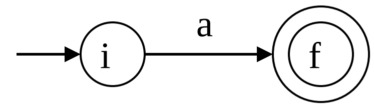
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
  To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Thomson’s Construction (cont.)

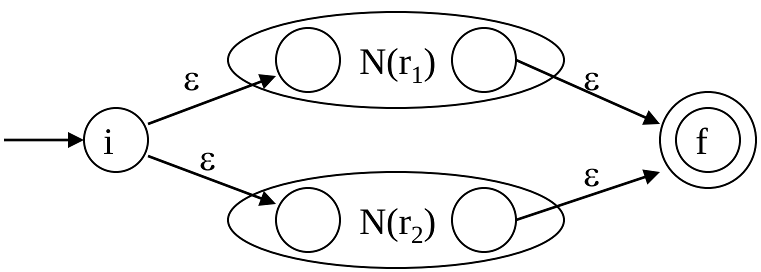
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ



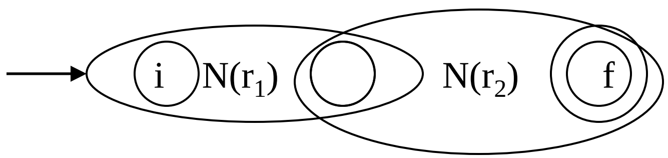
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

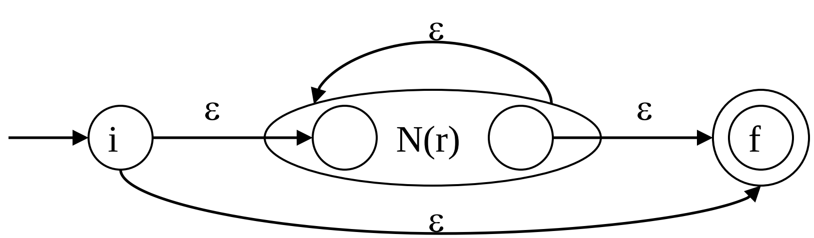
- For regular expression $r_1 r_2$



Final state of $N(r_2)$ become final state of $N(r_1r_2)$

NFA for $r_1 r_2$

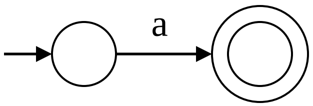
- For regular expression r^*



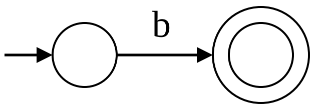
NFA for r^*

Thomson's Construction (Example - (a|b) * a)

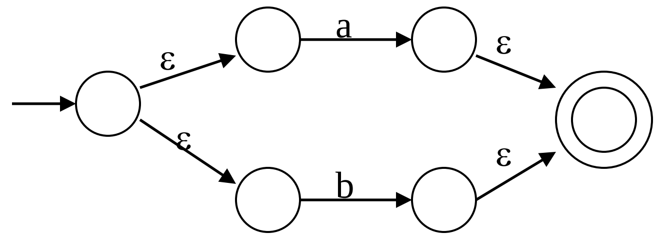
a:



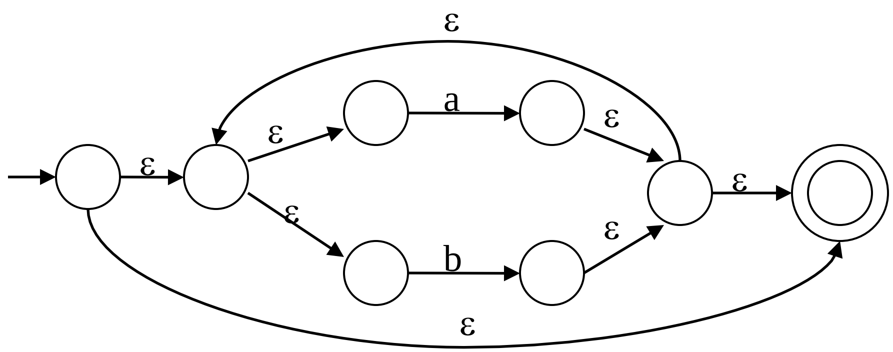
b:



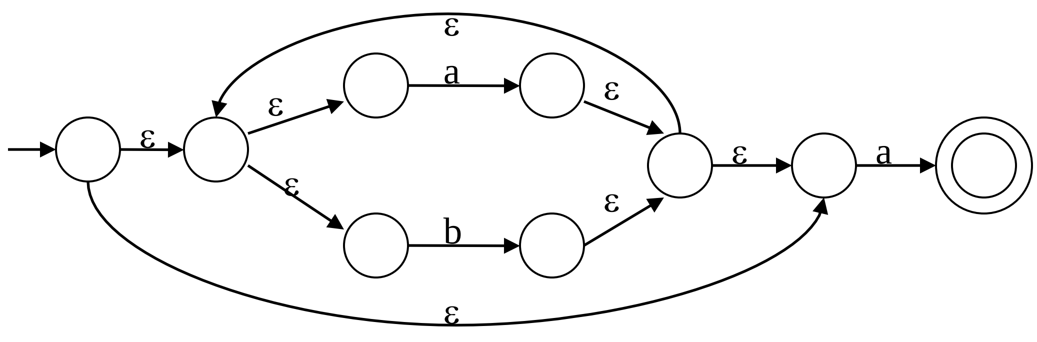
(a | b)



(a|b) *



(a|b) * a



Converting a NFA into a DFA (subset construction)

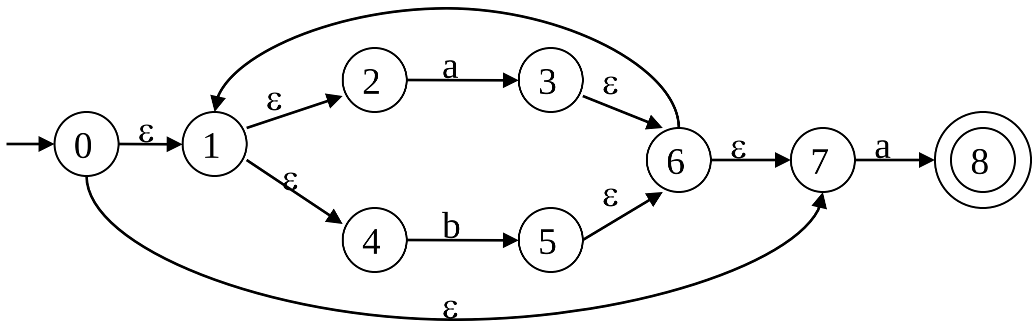
```
put  ε-closure({s0}) as an unmarked  state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
    begin
        mark S1
        for each input symbol a do
            begin
                S2 ← ε-closure(move(S1,a))
                if (S2 is not in DS) then
                    add S2 into DS as an unmarked state
                transfunc[S1,a] ← S2
            end
    end
```

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Converting a NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

$\Downarrow$ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] 🡐 S_1 transfunc[S_0,b] 🡐 S_2

$\Downarrow$ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] 🡐 S_1 transfunc[S_1,b] 🡐 S_2

$\Downarrow$ mark S_2

ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] 🡐 S_1 transfunc[S_2,b] 🡐 S_2

Converting a NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of $S_0=\{0,1,2,4,7\}$
S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1,2,3,4,6,7,8\}$